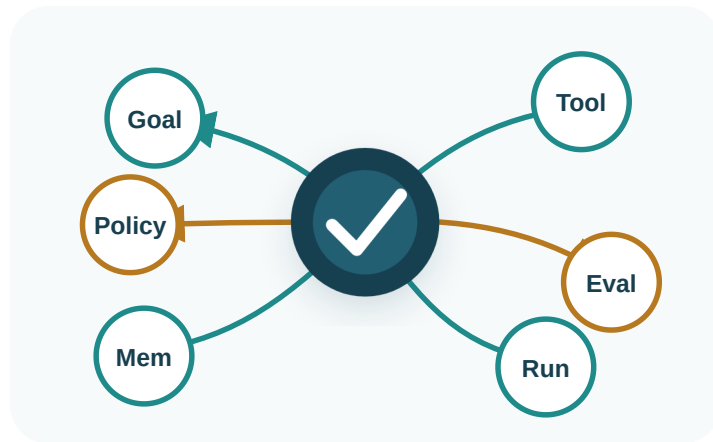




Agentic Systems Patterns

A practical reference for modern agent architecture: goals, loops, tools, skills, memory, protocols, multi-agent systems, and production runtimes.

Edition: 2026-06-16

Site: <https://gturitto.github.io/Agentic-Systems-Patterns/>

Courtesy PDF: <https://gturitto.github.io/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.pdf>

Courtesy EPUB: <https://gturitto.github.io/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.epub>

Repository: <https://github.com/GTuritto/Agentic-Systems-Patterns>

Content licensed under [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International \(CC BY-NC-SA 4.0\)](#). Code examples licensed under [MIT](#).

Table of Contents

1. Introduction
2. Start Here / How To Read This Book
3. Start Here / Logical Groups
4. Start Here / What Is An Agent?
5. Start Here / Glossary and Acronyms
6. Start Here / 10/10 Production Gate
7. Pattern Selection and Composition / Architecture Before Autonomy
8. Pattern Selection and Composition / Choosing the Right Pattern
9. Pattern Selection and Composition / Pattern Evaluation Checklist
10. Pattern Selection and Composition / From Patterns To Systems
11. Pattern Selection and Composition / Pattern Composition Playbook
12. Pattern Selection and Composition / Prompt Chaining and Gates
13. Pattern Selection and Composition / Routing and Handoffs
14. Pattern Selection and Composition / Resource-Aware Agent Design
15. Pattern Selection and Composition / Circuit Breakers, Fallbacks, and Replay
16. Pattern Selection and Composition / Source Map
17. Agent Runtime Foundations / Single Agent
18. Agent Runtime Foundations / Agent Loop
19. Agent Runtime Foundations / Goals and State
20. Agent Runtime Foundations / Tool Use
21. Agent Runtime Foundations / Structured Output
22. Engineering Practice and Frameworks / Agent Development Lifecycle
23. Engineering Practice and Frameworks / Agent Harnesses
24. Engineering Practice and Frameworks / Building a Minimal Agent Runtime
25. Engineering Practice and Frameworks / Agent Engineer Toolkit
26. Engineering Practice and Frameworks / Framework Selection
27. Engineering Practice and Frameworks / Cross-Framework Decision Matrix
28. Engineering Practice and Frameworks / Real Framework Setup Notes
29. Engineering Practice and Frameworks / Templates and Worksheets
30. Evaluation, Security, and Trust / Evaluation-Driven Agent Development
31. Evaluation, Security, and Trust / Agent Threat Model
32. Evaluation, Security, and Trust / Agent Security and Sandboxing
33. Evaluation, Security, and Trust / Agent UX and Human Trust
34. Control Loops / Planning and Execution
35. Control Loops / ReAct
36. Control Loops / Reflection
37. Control Loops / Evaluator-Optimizer
38. Control Loops / Self-Improvement
39. Control Loops / Self-Healing Workflows
40. Context, Memory, and Knowledge / Context Budgets and Working Sets
41. Context, Memory, and Knowledge / Context Engineering
42. Context, Memory, and Knowledge / Memory-Augmented Agent
43. Context, Memory, and Knowledge / Long-Term Episodic Memory
44. Context, Memory, and Knowledge / Semantic Recall and RAG
45. Context, Memory, and Knowledge / Working Memory
46. Context, Memory, and Knowledge / Knowledge-Bound Agents
47. Tools, Skills, and Protocols / Skills
48. Tools, Skills, and Protocols / Tool Capability Design
49. Tools, Skills, and Protocols / MCP-first Tool Use
50. Tools, Skills, and Protocols / A2A Agent Interoperability
51. Tools, Skills, and Protocols / Secure Agent Communication
52. Tools, Skills, and Protocols / Human Approval Gates
53. Multi-Agent Systems / Choosing Multi-Agent Topology
54. Multi-Agent Systems / Task Delegation
55. Multi-Agent Systems / Supervisor / Worker
56. Multi-Agent Systems / Debate and Consensus
57. Multi-Agent Systems / Parallel Agents
58. Multi-Agent Systems / CrewAI Flows and Crews
59. Systems Architecture / Agentic System Architecture
60. Systems Architecture / Agents As Services
61. Systems Architecture / Agentic RAG Systems
62. Systems Architecture / Open Personal Agent Architectures
63. Systems Architecture / Coding Agents
64. Systems Architecture / Computer-Use Agents
65. Systems Architecture / Domain Agent Architectures
66. Systems Architecture / Architecture Decision Records
67. Systems Architecture / Reference Architecture
68. Production Runtime / Overview
69. Production Runtime / Deployment Walkthrough
70. Production Runtime / Durable Workflows
71. Production Runtime / Observability and Evals
72. Production Runtime / Production Evaluation Feedback Loops

Table of Contents, Continued

- 73. Production Runtime / Cost Controls and Runtime Budgets
- 74. Production Runtime / Policy Enforcement
- 75. Production Runtime / Event-Triggered Agents
- 76. Production Runtime / Mastra Runtime
- 77. Hands-On Labs / Lab Guide
- 78. Hands-On Labs / Framework and Language Matrix
- 79. Hands-On Labs / Lab Production Readiness Checklist
- 80. Hands-On Labs / 01 - Tool-Using Agent
- 81. Hands-On Labs / 02 - Agent Loop and Planning
- 82. Hands-On Labs / 03 - Agentic RAG
- 83. Hands-On Labs / 04 - A2A Communication
- 84. Hands-On Labs / 05 - Multi-Agent Supervisor
- 85. Hands-On Labs / 06 - Observability and Evals
- 86. Hands-On Labs / 07 - Mastra Runtime Packaging
- 87. Hands-On Labs / 08 - CrewAI Flows and Crews
- 88. Hands-On Labs / From-Scratch Mini-Framework Track
- 89. Hands-On Labs / 09 - Minimal Agent Loop
- 90. Hands-On Labs / 10 - Tool Registry and Policy Gate
- 91. Hands-On Labs / 11 - Context, Memory, Trace, and Evals
- 92. Hands-On Labs / 12 - LangGraph State Graph
- 93. Hands-On Labs / 13 - AutoGen Transcript Evals
- 94. Hands-On Labs / Vertical Slice Examples
- 95. Capstone Projects / Capstone Projects
- 96. Capstone Projects / Support Refund Agent
- 97. Capstone Projects / Research RAG Agent
- 98. Capstone Projects / Multi-Agent Delivery Workflow
- 99. Historical Patterns / Historical Patterns
- 100. Publishing Appendix / Publishing and Releases
- 101. Publishing Appendix / Release Readiness Checklist
- 102. Publishing Appendix / Release Notes

Agentic Systems Patterns

Most agentic systems do not fail because the model is not smart enough. They fail because the architecture around the model is weak.

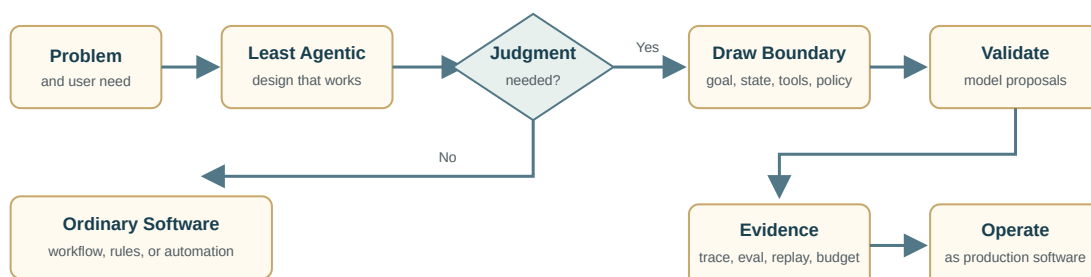
The goal is vague. The loop has no real stop condition. The tools are too powerful. State is hidden in a chat transcript. Nobody can replay what happened. The eval suite checks the final answer but not the path that produced it. Observability arrives after the first incident. Autonomy is added before the system has earned it.

This book is about fixing that.

It is written for software engineers, technical leads, architects, and builders who need to decide what an agent should own, what ordinary software should own, and what must never be left to a model. It is not a list of every agent pattern name I could find. It is a practical guide to choosing, composing, testing, securing, and operating agentic patterns without giving up engineering control.

A useful agent is not a prompt with ambition. It is software with boundaries.

Architecture Argument At A Glance

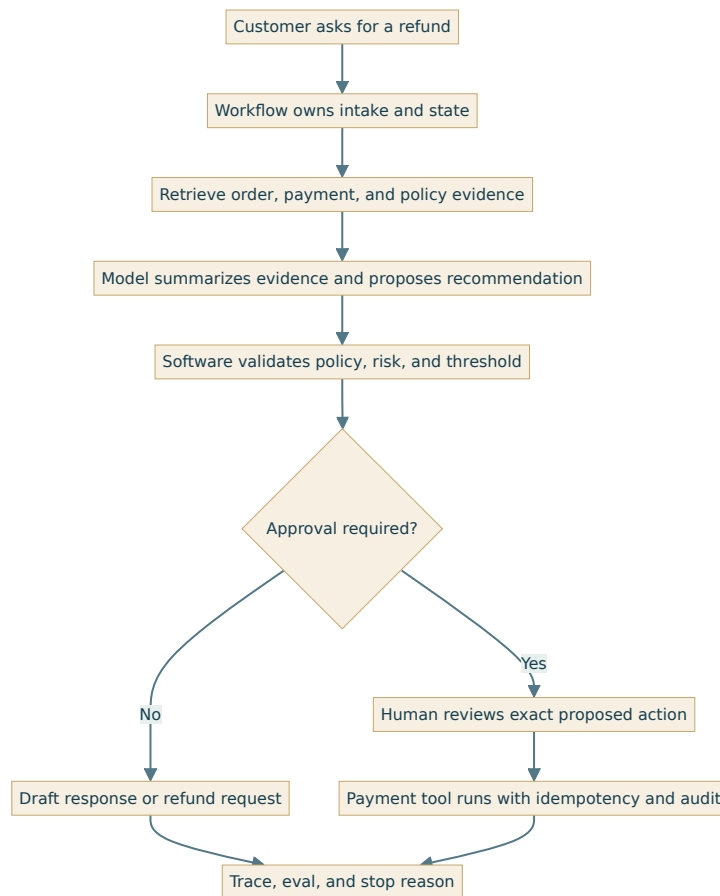


Use this diagram as the book's core test: if a design skips the boundary, validation, or evidence steps, it is not ready for production.

Running Case Study

Several chapters return to one product-shaped example: a support refund assistant. It is small enough to understand, but risky enough to force real architecture. A refund workflow touches customer data, policy evidence, payment tools, approval thresholds, audit records, and user communication. That makes it a useful test for the whole book.

The point is not to make refunds special. The point is to watch one system move from simple software, to bounded model judgment, to agentic investigation, to approval-gated side effects, to production evidence.



Use this case as a recurring question: what should the model decide, what must software own, and what evidence would let an operator replay the run after something goes wrong?

Reader Contract

Every chapter should help you make or review an engineering decision. A strong chapter in this book should give you at least one of these:

- a boundary you can draw;
- a pattern you can choose or reject;
- a contract you can implement;
- a failure mode you can test;
- a checklist you can use in review;
- an artifact you can reuse in a design, lab, capstone, or release.

If a chapter only teaches vocabulary, it should connect that vocabulary to a decision. If a chapter shows code, it should name the production controls still missing. If a chapter recommends a pattern, it should also say when not to use it.

That is the quality bar for the online book: readers should leave with decisions, evidence, and reusable artifacts, not just familiarity with agent terminology.

The Argument

My argument is simple:

1. Start with the least agentic architecture that can meet the requirement.
2. Add model judgment only where deterministic software is not enough.
3. Treat model outputs as proposals until software validates them.
4. Keep goals, state, tools, memory, and policy outside the model where they can be inspected.
5. Evaluate trajectories, not just final answers.
6. Operate agents like production systems, with traces, budgets, retries, approvals, and incident review.

Agentic design is not about making everything autonomous. It is about deciding exactly where autonomy helps, where it creates risk, and where ordinary software should stay in charge.

What You Should Be Able To Do

After reading the core chapters, you should be able to:

1. Decide whether a problem needs a prompt chain, workflow, single agent, or multi-agent system.
2. Draw the boundary between model judgment and deterministic control.
3. Define the goal, state, tools, memory, policy, budget, and stop conditions for an agentic run.
4. Review a tool surface for excess authority before exposing it to a model.
5. Design evals that test trajectories, not just final answers.
6. Explain how a system will trace, replay, approve, retry, roll back, and recover.
7. Turn a demo into a production design with explicit runtime controls.

If a chapter does not help you make one of those decisions, it is not doing enough work.

What This Book Covers

The book follows the decisions engineers make when they build real systems:

- foundations: single agents, loops, goals, state, tools, structured outputs, and context;
- pattern selection: when to use chains, routing, workflows, agents, or multi-agent systems;
- engineering practice: lifecycle, framework choice, security, evaluation, and user trust;
- control loops: planning, ReAct, reflection, evaluator-optimizer, and recovery loops;
- memory and knowledge: working memory, episodic memory, retrieval, and evidence boundaries;
- tools, skills, and protocols: tool contracts, MCP, A2A, secure communication, and approval gates;
- multi-agent systems: delegation, supervision, debate, parallel execution, and framework-shaped systems;
- systems architecture: how patterns compose into deployable products;
- production runtime: durable workflows, observability, evaluation, policy, events, and operations.

The pattern chapters are intentionally consistent. They are meant to be scanned during design work. The surrounding chapters carry the argument: why a pattern belongs in a system, what it costs, and how to know whether it is working.

What This Book Is Not

This is not a prompt engineering cookbook, a vendor comparison report, or a claim that agents should replace normal software. It does not assume that more autonomy is better. It also does not treat framework defaults as architecture.

Frameworks are useful, but they do not remove the need to decide where state lives, which tools are allowed, how approvals work, what gets traced, or how failures become regression tests. Those are product and architecture decisions.

How To Use It

If you are new to agentic systems, start with [How To Read This Book](#), then read [What Is An Agent?](#), [Architecture Before Autonomy](#), [Choosing the Right Pattern](#), and [From Patterns To Systems](#).

If you are reviewing a production design, start with the selection and engineering chapters before reading individual pattern pages.

If you are implementing, use the hands-on labs after you understand the architecture. The examples are deliberately small. The production notes show what must change before those examples become systems that can handle state, policy, evals, and observability.

Because this is an online book, use it as both a guided text and a reference. Follow the reading paths when you are learning the material. Use the sidebar, search, pattern pages, checklists, diagrams, labs, and capstones when you are designing or reviewing a real system.

License

Source code and runnable examples are licensed under the MIT License. Book/reference content, diagrams, worksheets, and generated publishing artifacts are licensed under [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International](#) ([CC-BY-NC-SA-4.0](#)).

How To Read This Book

This book has two jobs.

First, it makes an argument: agentic systems need architecture before autonomy. Second, it gives you a pattern reference you can use during design work.

You do not need to read every chapter in order. But do read the selection chapters before adding loops, tools, memory, or multiple agents to a system.

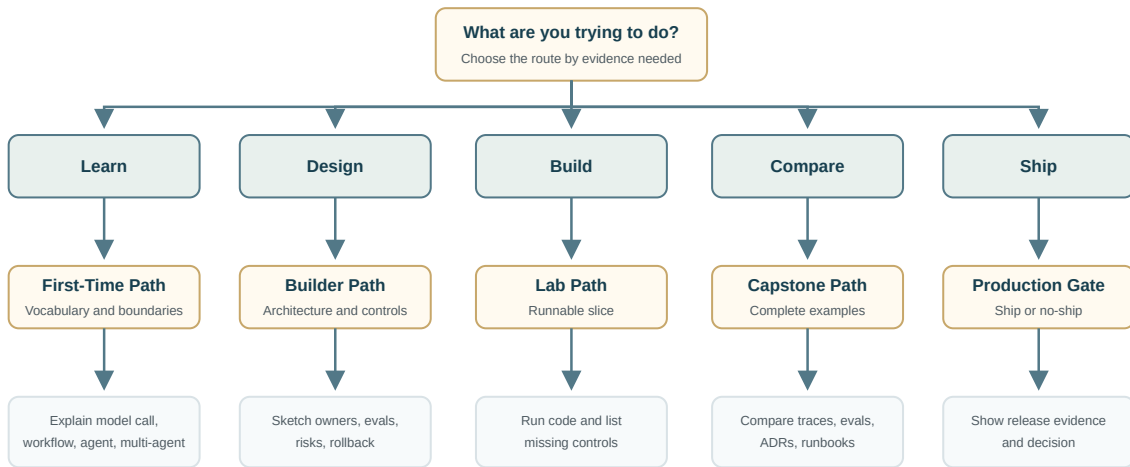
Choose Your Path

Use the path that matches the decision in front of you.

If you are...	Start with	Goal
New to agentic systems	First-Time Path	Learn the core vocabulary and boundaries.
Designing or reviewing a system	Builder Path	Choose patterns, controls, and production responsibilities.
Running code	Lab Path	Build small examples and connect them back to architecture.
Looking for product-shaped examples	Capstone Path	See patterns combined into complete systems.
Learning visually or onboarding a team	Visual Architecture Route	Use diagrams to understand boundaries, ownership, and production controls.
Checking one pattern during work	Reference Path	Scan use cases, failure modes, evals, and production checklists.
Choosing a section to read	Logical Groups	Understand why the book is grouped this way and where each group pays off.
Stuck on terminology	Glossary and Acronyms	Decode agent, eval, protocol, security, and production terms.
Preparing a release	10/10 Production Gate	Check whether the system is reviewable, testable, observable, and reversible.

The fastest useful route is not the shortest list of pages. It is the path that gets you to a design decision with enough context to avoid a bad abstraction.

Reading Path Decision Flow



The path is finished only when it produces the evidence on the right.

Time And Difficulty Guide

Use this table before choosing a route. The estimates assume a working software engineer who can skim familiar material and slow down for unfamiliar patterns, code, or production controls.

Path	Best For	Difficulty	Time	Evidence You Should Produce
First-Time Path	Learning the vocabulary and boundaries.	Beginner to intermediate	2-4 hours	A short explanation of agent, workflow, tool, state, memory, eval, and stop condition.
Builder Path	Designing or reviewing a real system.	Intermediate to advanced	1-2 days	Architecture sketch, selected pattern, rejected alternatives, owners, evals, and rollback path.
Lab Path	Running examples and seeing implementation boundaries.	Intermediate	1-3 days	Passing tests, trace output, missing-controls list, and one hardening task per lab.
Capstone Path	Comparing complete product-shaped systems.	Advanced	4-8 hours	Gap analysis against traces, evals, ADRs, runbooks, and release controls.
Reference Path	Checking one pattern during design work.	Intermediate	10-30 minutes per chapter	Fit/avoid decision, failure mode, eval case, and production checklist.
Release Path	Deciding whether a system can ship.	Advanced	2-6 hours	Completed scorecard, release evidence record,

Path	Best For	Difficulty	Time	Evidence You Should Produce
				known limits, and ship/no-ship decision.

Do not optimize for speed. Optimize for evidence. A fast read that does not change the design is not useful.

Path Exit Criteria

Use these checks to decide whether a path has paid off. If you cannot answer the exit question, continue with the linked path before choosing a pattern or shipping a system.

Path	You are done when you can...	If not, continue with...
First-Time Path	Explain the difference between a model call, prompt chain, workflow, single agent, and multi-agent system.	Pattern Selection and Composition
Builder Path	Draw the system boundary, name the owner of state, tools, memory, policy, evals, and rollback, and defend why each pattern belongs.	Reference Architecture
Lab Path	Run a small implementation, identify the missing production controls, and name the next hardening step.	Lab Production Readiness Checklist
Capstone Path	Compare a complete example with your own system and list the gaps in traces, evals, ADRs, runbooks, and rollout controls.	Capstone Projects
Reference Path	Decide whether the pattern fits, what it costs, how it fails, and which eval proves it works.	Choosing the Right Pattern
Release Path	Show evidence that the system is reviewable, testable, observable, reversible, and owned.	10/10 Production Gate

Do not treat a path as complete because the pages were read. Treat it as complete when it changes the design decision in front of you.

Logical Groups

The sidebar is organized as a design path, not a flat catalog.

Group	Purpose	Read When
Start Here	Orient yourself, choose a reader path, and define what counts as an agent.	You are new to the book or need the shortest entry point.

Group	Purpose	Read When
Pattern Selection and Composition	Decide whether to use a prompt, chain, router, loop, multi-agent system, or reliability pattern.	You are choosing or reviewing an architecture.
Agent Runtime Foundations	Learn the core runtime primitives: single agent, loop, state, tools, and structured output.	You need the vocabulary and control boundaries.
Engineering Practice and Frameworks	Move from demo to engineered system with lifecycle, harnesses, framework choices, and worksheets.	You are planning implementation.
Evaluation, Security, and Trust	Add eval gates, threat modeling, sandboxing, and human-trust controls.	The system affects users, data, or external actions.
Control Loops	Add planning, reflection, evaluator-optimizer, self-improvement, and self-healing patterns.	The agent needs iterative control.
Context, Memory, and Knowledge	Design working sets, context packets, memory, semantic recall, RAG, and knowledge boundaries.	The system depends on evidence or remembered state.
Tools, Skills, and Protocols	Design tool capabilities, skills, MCP, A2A, approvals, and secure communication.	The agent needs external capabilities.
Multi-Agent Systems	Choose and operate topologies for delegation, supervision, debate, and parallel work.	One agent is no longer enough.
Systems Architecture	Compose full systems: services, RAG systems, coding agents, computer-use agents, domains, ADRs, and references.	You are designing the whole system boundary.
Production Runtime	Run agents with durability, observability, feedback loops, policy, budgets, events, and rollout controls.	You are preparing for production.
Hands-On Labs	Build the patterns in small exercises and framework tracks.	You want implementation practice.
Capstone Projects	Study complete product-shaped examples with traces, evals, ADRs, runbooks, and rollback.	You want to see patterns combined.

Recommended First Pass

Start here if you want the book to read like a book, not a catalog:

- Reader: engineer or technical lead who wants the full argument.
- Outcome: understand the book's architecture-first position before using the catalog.
- Approximate depth: medium-long; best read across multiple sessions.

1. [Introduction](#)
2. [Logical Groups](#)
3. [What Is An Agent?](#)
4. [Architecture Before Autonomy](#)
5. [Choosing the Right Pattern](#)
6. [From Patterns To Systems](#)
7. [Agent Development Lifecycle](#)
8. [Agent Harnesses](#)
9. [Building a Minimal Agent Runtime](#)
10. [Cross-Framework Decision Matrix](#)
11. [Real Framework Setup Notes](#)
12. [Templates and Worksheets](#)
13. [Evaluation-Driven Agent Development](#)
14. [Agent Threat Model](#)
15. [Tool Capability Design](#)
16. [Agentic System Architecture](#)
17. [Agents As Services](#)
18. [Choosing Multi-Agent Topology](#)
19. [Coding Agents](#)
20. [Production Runtime Overview](#)
21. [Deployment Walkthrough](#)
22. [Capstone Projects](#)
23. [Support Refund Agent Capstone](#)
24. [Research RAG Agent Capstone](#)
25. [Multi-Agent Delivery Workflow Capstone](#)
26. [Production Evaluation Feedback Loops](#)
27. [Cost Controls and Runtime Budgets](#)
28. [Reference Architecture](#)
29. [10/10 Production Gate](#)

This path gives you the thesis before the catalog details.

First-Time Path

Start here if you are new to agentic systems:

- Reader: software engineer learning the vocabulary.
- Outcome: know what makes an agent different from a model call, chain, or workflow.
- Approximate depth: short-medium; read before the labs.

1. [Introduction](#)
2. [Logical Groups](#)
3. [What Is An Agent?](#)
4. [Glossary and Acronyms](#)
5. [Single Agent](#)

6. [Agent Loop](#)
7. [Goals and State](#)
8. [Tool Use](#)
9. [Tool Capability Design](#)
10. [Context Budgets and Working Sets](#)
11. [Context Engineering](#)
12. [Choosing the Right Pattern](#)
13. [Hands-On Labs](#)

This path gives you the core vocabulary before the production runtime chapters.

Builder Path

Use this path when you are designing or reviewing a system:

- Reader: builder, staff engineer, architect, reviewer, or technical lead.
 - Outcome: produce a defensible design with explicit state, tools, memory, policy, evals, and runtime boundaries.
 - Approximate depth: medium-long; use alongside a real design.
1. [Choosing the Right Pattern](#)
 2. [Resource-Aware Agent Design](#)
 3. [Agent Development Lifecycle](#)
 4. [Agent Harnesses](#)
 5. [Building a Minimal Agent Runtime](#)
 6. [Cross-Framework Decision Matrix](#)
 7. [Real Framework Setup Notes](#)
 8. [Templates and Worksheets](#)
 9. [Evaluation-Driven Agent Development](#)
 10. [Agent Threat Model](#)
 11. [Tool Capability Design](#)
 12. [Agent Security and Sandboxing](#)
 13. [Agents As Services](#)
 14. [Choosing Multi-Agent Topology](#)
 15. [Reference Architecture](#)
 16. [Production Runtime Overview](#)
 17. [Deployment Walkthrough](#)
 18. [Capstone Projects](#)
 19. [Support Refund Agent Capstone](#)
 20. [Research RAG Agent Capstone](#)
 21. [Multi-Agent Delivery Workflow Capstone](#)
 22. [Observability and Evals](#)
 23. [Production Evaluation Feedback Loops](#)
 24. [Cost Controls and Runtime Budgets](#)
 25. [10/10 Production Gate](#)

This path is best for architecture work, design reviews, and production readiness checks.

Lab Path

Use this path when you want to run code:

- Reader: implementation-focused engineer.
- Outcome: run small examples, inspect boundaries, and understand what must change before production.
- Approximate depth: hands-on; expect setup and debugging time.

1. [Lab Framework and Language Matrix](#)
2. [Real Framework Setup Notes](#)
3. [Lab Production Readiness Checklist](#)
4. [Deployment Walkthrough](#)
5. [Lab 01 - Tool-Using Agent](#)
6. [Lab 02 - Agent Loop and Planning](#)
7. [Lab 03 - Agentic RAG](#)
8. [Lab 04 - A2A Communication](#)
9. [Lab 05 - Multi-Agent Supervisor](#)
10. [Lab 06 - Observability and Evals](#)
11. [Lab 07 - Mastra Runtime Packaging](#)
12. [Lab 08 - CrewAI Flows and Crews](#)
13. [From-Scratch Mini-Framework Track](#)
14. [Lab 09 - Minimal Agent Loop](#)
15. [Lab 10 - Tool Registry and Policy Gate](#)
16. [Lab 11 - Context, Memory, Trace, and Evals](#)
17. [Lab 12 - LangGraph State Graph](#)
18. [Lab 13 - AutoGen Transcript Evals](#)
19. [Capstone Projects](#)
20. [Support Refund Agent Capstone](#)
21. [Research RAG Agent Capstone](#)
22. [Multi-Agent Delivery Workflow Capstone](#)
23. [10/10 Production Gate](#)

Each lab links back to the pattern chapters and downloadable source bundles. The labs intentionally move between Python, TypeScript, framework-neutral code, LangChain/LangGraph-style retrieval, LangGraph-style state graphs, AutoGen-style supervision and transcript evals, Mastra-style runtime packaging, CrewAI-style flow orchestration, protocol-first A2A code, and test-based evals so you can see the architecture beneath the API.

Capstone Path

Use this path when you want complete product-shaped examples:

- Reader: engineer who has finished the core chapters or labs.
- Outcome: connect patterns, traces, evals, ADRs, runbooks, and rollout controls into complete systems.
- Approximate depth: medium; best used as design-review material.

1. [Capstone Projects](#)
2. [Support Refund Agent Capstone](#)
3. [Research RAG Agent Capstone](#)
4. [Multi-Agent Delivery Workflow Capstone](#)
5. [Deployment Walkthrough](#)
6. [Templates and Worksheets](#)

This path is best after the labs. It shows how patterns, frameworks, evals, traces, ADRs, and runbooks fit together.

Reference Path

Use the sidebar or search when you need a specific pattern. Each generated pattern chapter follows the same structure:

- Reader: engineer making a local pattern choice.
- Outcome: decide whether a pattern fits, what it costs, and how to test it.
- Approximate depth: fast lookup; jump between related chapters.

Pattern pages include:

- when to use it
- when to avoid it
- architecture
- system shape
- core protocol
- implementation notes
- failure modes
- evaluation strategy
- production checklist
- source code and downloads

The repeated structure makes chapters easy to scan during design work. Do not read those pages like a novel. Use the authored chapters for the argument and the pattern pages for decisions.

Logical Groups

This book is grouped by the order in which engineering decisions usually appear. Start with the problem and pattern choice, then add runtime primitives, engineering practice, risk controls, architecture, production operations, labs, and complete examples.

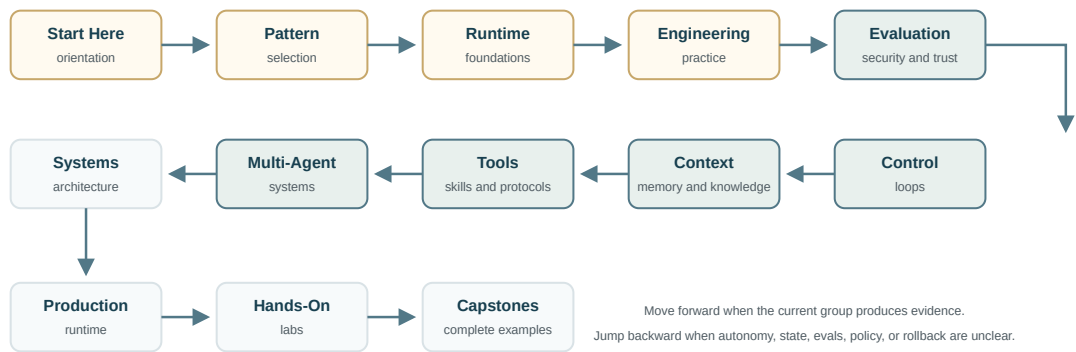
Use this page when the sidebar feels large. Each group answers a different question and gives the reader a clear place to start.

Group Map

Group	Main Question	Reader Payoff
Start Here	What is this book, and what counts as an agent?	You get the vocabulary, reader paths, glossary, and production bar.
Pattern Selection and Composition	Which pattern should this system use?	You avoid adding autonomy where a prompt, chain, router, or workflow is enough.
Agent Runtime Foundations	What primitives make an agent run?	You understand loops, state, tools, structured output, context, and control boundaries.
Engineering Practice and Frameworks	How do we build this like software, not a demo?	You get lifecycle, harness, framework, setup, and worksheet guidance.
Evaluation, Security, and Trust	How do we know this is safe and useful?	You connect evals, threat models, sandboxing, and user trust to release decisions.
Control Loops	When should the system plan, reflect, optimize, or heal?	You choose iterative control without hiding failure or cost.
Context, Memory, and Knowledge	What should the agent know, remember, or retrieve?	You design context packets, memory, RAG, and knowledge boundaries deliberately.
Tools, Skills, and Protocols	What can the agent do outside the model?	You define tool contracts, skills, approvals, MCP, A2A, and communication safety.
Multi-Agent Systems	When is one agent no longer enough?	You compare delegation, supervision, debate, parallelism, and crew-style workflows.
Systems Architecture	How do these patterns become a full system?	You see services, RAG systems, coding agents, personal agents, domain architectures, ADRs, and references.

Group	Main Question	Reader Payoff
Production Runtime	How does the system survive real operation?	You add durability, observability, feedback loops, budgets, policy, events, rollout, and rollback.
Hands-On Labs	How do these ideas look in code?	You build small vertical slices across Python, TypeScript, custom runtimes, and common agent frameworks.
Capstone Projects	What does a complete agentic system look like?	You study product-shaped systems with traces, evals, ADRs, runbooks, and rollback plans.
Historical Patterns	Which older terms still matter?	You can compare current architecture language with older pattern names.
Publishing Appendix	How is the book maintained and released?	You get release, checklist, and publishing notes for the online book.

Design Pipeline



Read left to right when you want the book to build one design argument. Jump to the group with the missing evidence when you are reviewing a real system.

Problem Map

Use this map when you know the problem but not the right group. Start with the section listed, then follow its related chapters.

If Your Problem Is...	Start With	Leave With
The team wants to add an agent but has not justified autonomy.	Pattern Selection and Composition	A smaller design or a clear reason for agentic behavior.
The implementation hides state in prompts or chat history.	Agent Runtime Foundations	Explicit goals, state, tools, structured output, and stop conditions.

If Your Problem Is...	Start With	Leave With
A demo works, but nobody knows how to test or operate it.	Engineering Practice and Frameworks	A lifecycle, harness, framework boundary, and worksheet trail.
The system can affect users, data, money, or external systems.	Evaluation, Security, and Trust	Evals, threat model, sandbox, approval, and trust controls.
The agent keeps looping, retrying, or self-correcting without clear evidence.	Control Loops	A loop design with stop conditions, budgets, and reviewable failure states.
Answers depend on documents, memory, or context assembly.	Context, Memory, and Knowledge	Source policy, context packet, retrieval, memory, and freshness rules.
The agent needs tools, approvals, MCP, A2A, or secure messages.	Tools, Skills, and Protocols	Narrow tool contracts, protocol envelopes, permissions, and audit records.
One agent is overloaded or needs specialist collaborators.	Multi-Agent Systems	A topology, role boundary, transcript, merge policy, and stop rule.
Several patterns must become one deployable product.	Systems Architecture	A system boundary, ADRs, service shape, and reference architecture.
The design is near production.	Production Runtime	Durability, observability, policy, budgets, rollout, and rollback evidence.
You want proof in code before committing to a design.	Hands-On Labs	A small runnable slice and a list of missing production controls.
You want to compare against complete examples.	Capstone Projects	A gap list for traces, evals, ADRs, runbooks, and release controls.

Recommended Order

Read these groups in order when you want the book to build one argument:

1. Start Here
2. Pattern Selection and Composition
3. Agent Runtime Foundations
4. Engineering Practice and Frameworks
5. Evaluation, Security, and Trust
6. Control Loops
7. Context, Memory, and Knowledge
8. Tools, Skills, and Protocols

9. Multi-Agent Systems
10. Systems Architecture
11. Production Runtime
12. Hands-On Labs
13. Capstone Projects

This order keeps the reader inside one design argument: choose the pattern, define the runtime, add loop control only where needed, give the agent evidence and capabilities, decide whether one agent is enough, then turn the design into an operated system.

Visual Architecture Route

Use this route when you want the diagrams to carry the first pass. It is useful for design reviews, team onboarding, and PDF reading.

Step	Chapter	What The Diagram Should Clarify
1	Agent Loop	How state, actions, observations, and stop conditions form a run.
2	Planning and Execution	Who owns the plan, validation, execution, and progress.
3	Memory-Augmented Agent	How retrieval, context injection, storage, and correction stay governed.
4	Tool Capability Design	Where permissions, policy, tool calls, and audit records sit.
5	Secure Agent Communication	Which authority gates protect agent-to-agent or remote-tool exchange.
6	Choosing Multi-Agent Topology	When to delegate, supervise, debate, parallelize, or stay single-agent.
7	Agentic System Architecture	How the patterns compose into services, state, policy, evals, and operations.
8	Production Runtime Overview	What must exist around the agent before production use.
9	Event-Triggered Agents	How unattended triggers stay idempotent, observable, replayable, and safe.

After this route, use the pattern chapters as a reference. Each diagram should answer a concrete ownership question, not just decorate the page.

Group Contracts

Each group should give the reader one durable artifact or decision. Use this table to know when to continue, pause, or jump.

Group	Enter When	Exit With
Start Here	You need orientation, vocabulary, or the production bar.	A reading path and a shared definition of agentic system.
Pattern Selection and Composition	You are deciding whether autonomy belongs in the design.	A selected pattern, rejected alternatives, and acceptance criteria.
Agent Runtime Foundations	You need to make the run explicit.	Goal, state, tool, output, context, and stop-condition contracts.
Engineering Practice and Frameworks	You are turning a design into software.	Harness, framework choice, worksheet, and review trail.
Evaluation, Security, and Trust	The system affects real users, data, money, or decisions.	Eval set, threat model, sandbox boundary, and trust controls.
Control Loops	The system must plan, revise, evaluate, or recover.	Loop policy, budget, evaluator, failure states, and stop rules.
Context, Memory, and Knowledge	The answer depends on evidence, history, or retrieval.	Context packet, source policy, memory rule, and freshness boundary.
Tools, Skills, and Protocols	The agent needs external capabilities or agent-to-agent exchange.	Tool contracts, permission envelope, approval gate, and audit record.
Multi-Agent Systems	One agent is overloaded or multiple roles improve the work.	Topology, role contracts, merge policy, transcript, and escalation path.
Systems Architecture	Patterns must become a deployable product.	Service boundary, ADRs, data/control flow, and reference architecture.
Production Runtime	The system is close to release or already operating.	Durability, traces, eval feedback, budgets, rollout, and rollback plan.
Hands-On Labs	You need proof in code.	Runnable slice, test evidence, and missing-production-control list.
Capstone Projects	You want an end-to-end comparison point.	Gap analysis against a product-shaped system.

Handoff Rules

Move forward only when the current group has produced evidence. A pattern choice without acceptance criteria is not ready for implementation. A runtime design without state and stop conditions is not ready for tools. A tool surface without approval and audit boundaries is not ready for production. A lab without a missing-controls list is not proof of readiness.

Jump backward when evidence is missing:

Missing Evidence	Return To
The team cannot explain why the system needs autonomy.	Pattern Selection and Composition
State, tools, memory, or policy live only inside prompts.	Agent Runtime Foundations
There is no eval set or threat model.	Evaluation, Security, and Trust
The system has no trace, replay, budget, or rollback path.	Production Runtime
The architecture has no written decision record.	Systems Architecture

These handoffs keep the book from becoming a tour of concepts. Every section should either produce a decision, expose a gap, or send the reader to the section that can close it.

Value Test

Each group should help a reader make a better engineering decision:

- choose a smaller design when autonomy is unnecessary
- make state, tools, memory, and context visible
- add evals before relying on behavior
- separate model judgment from deterministic control
- make risk reviewable by another engineer
- connect examples to production responsibilities

If a group does not help with one of those outcomes, it should be revised, merged, or removed.

Takeaway

The book is not organized by trend names. It is organized by decisions: choose the pattern, define the runtime, control risk, compose the system, operate it, and prove it with examples.

What Is An Agent?

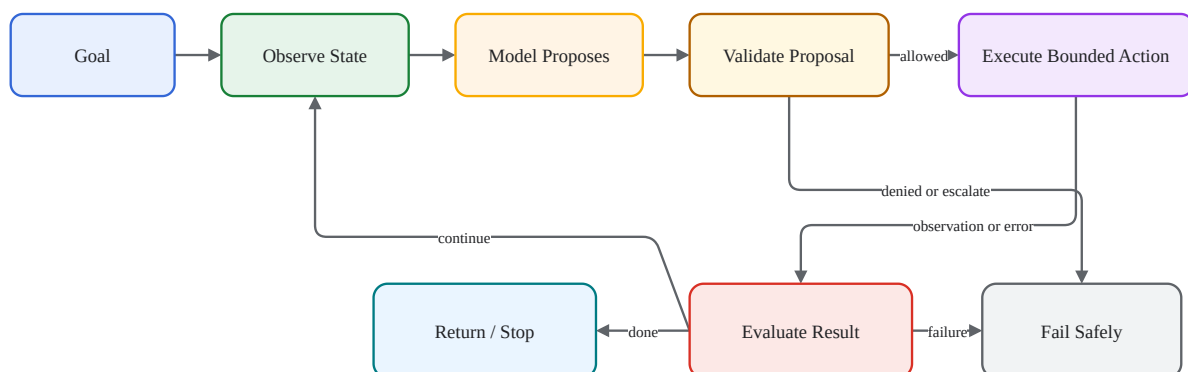
An agent is not magic. It is a loop with access to a model, some state, a few decisions, and sometimes tools. The loop is the part that matters; everything else is packaging.

A framework may call it a graph, a crew, a swarm, a harness, an assistant, or a runtime. Strip away the names and you usually find the same shape underneath: observe, decide, act, observe again, and eventually stop.

This chapter owns the mechanical definition of an agent. It does not define a framework, product interface, memory system, or production runtime. Those layers surround the loop, but they do not replace it.

Read this before the pattern catalog if you want the plain model first: an agent is a controlled loop, not a personality, a chat window, or a product category. After this, [Architecture Before Autonomy](#) explains when that loop is worth using.

Agent Loop



The Minimal Agent

The smallest useful agent needs a goal or input, instructions, context, access to a model, a decision step, state, a rule for what happens next, and a stop condition. Tools are optional, but they are usually what makes the agent worth building.

In plain terms, that is:

```

input
-> build context
-> call model
-> parse decision
-> answer, call a tool, ask for help, or stop
-> update state
-> repeat if needed
  
```

Most of agent engineering is making each of those arrows explicit and safe. The diagram is easy. The discipline is in deciding what the model is allowed to do at each step, and what code keeps for itself.

The smallest useful loop can be sketched like this:

```
type Decision =
  | { kind: 'answer'; text: string }
  | { kind: 'tool'; name: 'lookup_order'; input: { orderId: string } }
  | { kind: 'ask_human'; question: string }
  | { kind: 'stop'; reason: string };

interface AgentState {
  goal: string;
  steps: number;
  observations: unknown[];
  maxSteps: number;
}

function validateDecision(decision: Decision): Decision {
  if (decision.kind !== 'tool') return decision;
  if (!decision.input.orderId.trim()) {
    return { kind: 'stop', reason: 'invalid_tool_input' };
  }
  return decision;
}

async function runAgent(state: AgentState): Promise<Decision> {
  while (state.steps < state.maxSteps) {
    const context = buildWorkingSet(state);
    const proposal = await callModelForDecision(context);
    const decision = validateDecision(proposal);

    if (decision.kind === 'answer' || decision.kind === 'ask_human') {
      return decision;
    }

    if (decision.kind === 'tool') {
      const result = await callApprovedTool(decision.name, decision.input, {
        idempotencyKey: `step:${state.steps}`,
      });
      state.observations.push(result);
      state.steps += 1;
      continue;
    }
  }
}
```

```

    return decision;
}

return { kind: 'stop', reason: 'step_budget_exhausted' };
}

```

The important part is not the syntax. The model proposes a decision, software validates it before execution, state records the observation, and the loop stops for an explicit reason. The full runnable version appears in [Agent Loop](#).

A Model Call Is Not Yet An Agent

A model call takes input and returns text or structured output. For summarization, extraction, classification, or rewriting, that is often all you need.

An agent adds control flow around the call. It can decide whether more work is needed, whether to use a tool, whether to ask a human, whether to retry, or whether to stop. The difference is not intelligence. It is runtime shape.

System	What It Can Do	Main Risk
Model call	Produce one response from provided context.	No action or recovery.
Prompt chain	Move through known steps.	Brittle gates and unnecessary latency.
Workflow with LLM steps	Let code own the path while models handle judgment.	Overfitting to a fixed process.
Agent loop	Choose the next step from observations.	Loops, tool misuse, and hidden state drift.

So be careful with the word. Not every model-backed feature is an agent. Reserve the term for systems that can make at least some runtime decisions about what to do next.

A Quick Classification Test

When a team says “agent”, classify the system before debating the framework.

Example	Better Name	Why
A button that rewrites a paragraph once.	Model call	There is no loop, tool use, or runtime decision.

Example	Better Name	Why
A three-step support reply generator: classify, retrieve policy, draft response.	Prompt chain or workflow	The path is known before execution.
A refund flow where code checks policy, the model summarizes evidence, and approval gates the payment.	Workflow with LLM steps	Code owns the path and side effects.
A research assistant that searches, reads results, decides whether more retrieval is needed, and stops when evidence is sufficient.	Agent loop	The next step depends on observations from the run.
A delivery system with planner, dispatcher, driver-communication worker, and escalation policy.	Multi-agent system or workflow with agents	Work is split across roles with separate context and authority.

The classification is not about prestige. A model call is not worse than an agent. A workflow is not less modern than a loop. The right name tells you what must be tested, traced, secured, and operated.

Running Example: Refund Support

The support refund assistant used across this book starts as a workflow, not an agent. The known path is clear: classify the request, load the order, retrieve the current refund policy, summarize evidence, validate the recommendation, and request approval when money can move. Code should own that path.

The agentic part appears only when the workflow hits bounded uncertainty. Maybe the order data conflicts with the delivery record. Maybe the customer claims a policy exception. Maybe the evidence is missing and the system must decide whether to search another source, ask a clarifying question, or stop. That narrow investigation can justify a loop.

Part of the refund system	Better shape	Owner
Intake, account lookup, and policy version selection	Deterministic workflow	Application code
Evidence summary and recommendation draft	Model call or structured-output step	Model proposes, code validates
Missing or conflicting evidence investigation	Agent loop with max steps	Loop runtime
Refund threshold, exception policy, and approval requirement	Policy gate	Software
Payment execution	Tool call after approval	Tool gateway

This is the discipline behind the word “agent.” Do not ask whether the refund assistant sounds autonomous. Ask which runtime decision needs autonomy, which side effects need authority, and which stop reason the system will record.

Decision Trees Come Before Autonomy

Many useful agents are mostly decision trees with model calls inside them. That is not a weakness, and it is often the right design:

```
if request is out of scope:
    refuse or redirect
else if account data is missing:
    ask a question
else if policy evidence is missing:
    retrieve documents
else if action is high risk:
    request approval
else:
    call the model to draft the recommendation
```

This is still an agentic system as long as the model is making bounded judgments inside the flow. What keeps it safe is that code owns the policy, the state, and the high-risk transitions. Add autonomy where the decision tree gets too large to maintain, or where the next step genuinely depends on something the run discovers along the way. Adding it anywhere else just buys you risk you did not need.

Tools Make The Loop Consequential

Without tools, an agent can only produce text. With tools, it can search, calculate, retrieve, write files, browse, call APIs, run code, send messages, update tickets, or change customer data. That is where the system becomes useful, and it is the same place it becomes dangerous.

Never treat a tool call as “the model did it.” The model proposed the call; software executed it. That boundary is where most of your safety lives, so it has to be real: tool names are known ahead of time, arguments are typed, permissions are checked before execution, timeouts exist, results are recorded, side effects are auditable, and risky actions wait for approval. The more powerful the tool, the less trust you should leave sitting in the prompt.

State Makes The Loop Coherent

An agent needs state because the loop needs a memory of its own run. At a minimum, state should be able to answer what the active goal is, what context was used, what has already happened, which tools were called, what evidence was found, what errors occurred, how much budget remains, and why the run stopped.

Conversation history does not cover this. History is evidence of what was said; it is not a model of what the system knows and has done. Real state is what lets the system resume after an interruption, avoid repeating work, explain a decision after the fact, and turn a production incident into an eval case.

Stop Conditions Are Part Of The Agent

A loop without a stop condition is not an agent. It is a cost generator. Common stop conditions include the success criteria being met, a required field being missing, a non-recoverable tool failure, a policy block, a pending human approval, a max-iteration limit, an exhausted cost or latency budget, or a user cancelling the run.

Whatever ends the run, the stop reason should be explicit. “Done” is not enough. A system worth operating records whether the agent completed, refused, escalated, failed, timed out, or stopped because a budget ran out, because every one of those needs a different response from the people watching it.

What Frameworks Add

Frameworks do not remove the loop. They package it, and they save real work: graph execution, routing, tool registries, model adapters, memory stores, skills, subagents, durable checkpoints, streaming, human-in-the-loop interrupts, tracing, and deployment hooks. None of that is wasted effort, and rebuilding these primitives by hand is rarely a good use of a team’s time.

But the framework does not change the question you have to answer: what does the loop observe, what can it decide, what can it do, and when does it stop? If your team cannot answer that without first naming a framework, it does not understand its own agent yet.

What Makes An Agent Good

A good agent is not the one with the most autonomy. It is the one whose autonomy is useful and bounded. In practice that means clear goals, narrow tools, explicit state, typed outputs, visible decisions, bounded loops, evals that actually catch regressions, safe failure modes, and human escalation for the risky cases. Bad agents hide all of this inside a prompt and hope the model behaves.

The design rule follows from everything above: if you cannot draw the loop, the tools, the state, and the stop condition, you do not understand the agent yet, and you are not ready to give it autonomy.

Before calling something an agent, write down four things:

1. The runtime decision it is allowed to make.
2. The state it reads and writes outside the model context.
3. The tools or side effects it can request.
4. The exact stop reasons the system records.

If any of those are missing, you probably have a model-backed feature, chain, or workflow that needs clearer boundaries before it needs more autonomy.

The next question is not how to add more autonomy. It is whether the task needs a loop at all. Continue with [Architecture Before Autonomy](#), then return to [Agent Loop](#) when runtime decisions are justified.

Related Chapters

- [Architecture Before Autonomy](#)
- [Single Agent](#)
- [Agent Loop](#)
- [Goals and State](#)
- [Tool Use](#)
- [Choosing the Right Pattern](#)

Glossary and Acronyms

Use this chapter when a term blocks progress. The definitions are short on purpose. Each term explains what the word means in this book and why it matters when you design, test, or operate an agentic system.

Read the glossary as a control vocabulary. In this book, terms are useful only when they help a team decide ownership, risk, evidence, or release readiness.

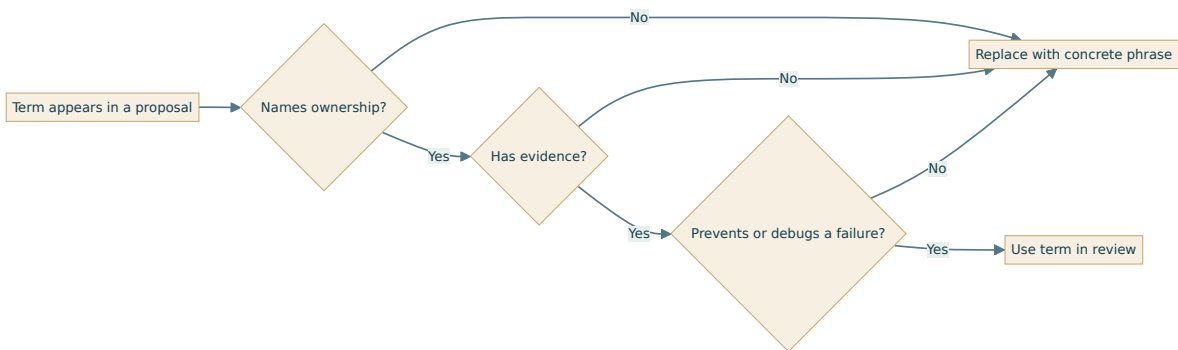
How To Use This Glossary

Use a term in a design review only if the team can answer three questions:

- 1. What does this term own?
- 2. What evidence proves it works?
- 3. What failure does it help us prevent or debug?

If a term does not answer those questions, replace it with a more concrete phrase. For example, “agentic workflow” is too broad unless you can name the goal, state, tool boundary, policy gate, and stop condition.

Use this flow when a term sounds useful but vague. Keep the term only when it sharpens ownership, evidence, and failure analysis.



Term Review Map

Use this map during design reviews. When a term appears in a proposal, ask the review question before accepting it.

When Someone Says...	Ask...	Go To
Agent	What runtime decision does the model make that code cannot make safely enough?	What Is An Agent?
Autonomous	Which decisions, tools, budgets, and stop conditions are delegated?	Architecture Before Autonomy

When Someone Says...	Ask...	Go To
Memory	Is this task-local context, durable user memory, episodic history, or retrieved knowledge?	Context Engineering
Tool	What authority does it grant, and what policy checks run before side effects?	Tool Capability Design
Eval	Which behavior, trajectory, policy decision, or failure mode does it catch?	Evaluation-Driven Agent Development
Human-in-the-loop	Does the human approve an exact action, review a result, or only receive a notification?	Human Approval Gates
Multi-agent	What separate context, permission, skill, latency, or review need justifies another agent?	Choosing Multi-Agent Topology
Production-ready	Where is the evidence for traces, eval gates, rollback, ownership, and incident response?	10/10 Production Gate

Core Terms

Term	Meaning	Why It Matters
Agent	A system that pursues a goal through a loop, uses tools or context, tracks state, and stops by rule.	It separates real agent architecture from a single model call.
Agent loop	The observe, decide, act, evaluate, and stop cycle around an agent.	It is where budgets, policy, state, and failure handling must live.
Autonomy	The amount of decision authority delegated to the system.	More autonomy requires stronger boundaries, tests, approvals, and rollback.
Boundary	The line between model choice and deterministic system control.	Weak boundaries turn model uncertainty into system risk.
Capability	A thing the system can do, usually through a tool, skill, workflow, or service.	Capabilities need ownership, authorization, audit, and tests.
Context	The information passed into a model or agent step.	Context controls what the model can use, confuse, leak, or ignore.
Goal	The explicit outcome the system is trying to reach.	Ambiguous goals produce ambiguous behavior and weak evals.

Term	Meaning	Why It Matters
Policy	A rule that constrains what the agent may do.	Policy must be enforced outside the prompt when actions carry risk.
State	Data that records the current run, memory, progress, decisions, or external effects.	Hidden state makes replay, debugging, and safety reviews hard.
Stop condition	A rule that ends a loop or hands control to a human/system.	Without stop conditions, agents waste budget or take unsafe repeated actions.
Tool	A callable capability exposed to an agent.	Tool design is a control plane, not just an API wrapper.

Common Confusions

These pairs cause many weak designs. Use the sharper term in reviews.

Confused Terms	Distinction
Model call vs agent	A model call returns one response. An agent loop can decide what to do next.
Workflow vs agent	A workflow follows a path owned by code. An agent chooses at least some steps from observations.
Memory vs context	Context is what the model sees now. Memory is stored state with retention, correction, and deletion rules.
Tool vs skill	A tool executes a capability. A skill packages procedure, references, scripts, templates, and examples.
Policy vs prompt instruction	Policy is enforced by software or an authority boundary. A prompt instruction is guidance to the model.
Eval vs test	Tests usually check deterministic behavior. Evals check model-influenced outputs, trajectories, or judgments.
Trace vs log	Logs record events. Traces connect the run: goal, context, decisions, tools, outputs, costs, and stop reason.
Approval vs notification	Approval blocks an exact action until accepted. Notification only tells someone what happened.

When in doubt, choose the term that makes ownership and evidence more inspectable.

Architecture Terms

Term	Meaning	Why It Matters
Architecture decision record (ADR)	A short record of an architecture choice, context, alternatives, and consequences.	ADRs make agent design reviewable after the demo phase.
Circuit breaker	A control that stops or degrades execution when failures cross a threshold.	It prevents repeated model/tool failure from becoming user-visible damage.
Deterministic boundary	Code, schema, policy, or workflow logic that does not rely on model judgment.	It keeps high-risk decisions outside probabilistic text generation.
Durable workflow	A workflow that can persist, resume, retry, and recover after interruption.	Production agents need continuity across crashes, queues, and timeouts.
Fallback	A safer alternate path used when the preferred path fails or becomes too risky.	Fallbacks preserve service quality without pretending autonomy always works.
Handoff	A typed transfer of work between components, agents, tools, or humans.	Handoffs need context, state, ownership, and acceptance criteria.
Idempotency	The property that repeating an action produces the same intended effect.	Retries and replay are unsafe without idempotent external actions.
Replay	Re-running a trace, workflow step, or decision path for debugging or evaluation.	Replay turns incidents into reproducible tests.
Routing	Choosing which path, tool, model, workflow, or agent should handle a task.	Bad routing adds cost, latency, and failure modes.
Service boundary	The contract around a deployable system component.	Agents become operable when treated like services with APIs and SLOs.

Evaluation Terms

Term	Meaning	Why It Matters
Eval case	A concrete input, expected behavior, and scoring rule.	It turns a vague quality claim into a repeatable check.
Eval gate	A required evaluation threshold before release or rollout.	It prevents prompt or policy changes from shipping on confidence alone.

Term	Meaning	Why It Matters
Fixture	A controlled test input, document set, tool response, or trace.	Fixtures make agent behavior reproducible.
Grader	Code, rubric, or model that scores an output or trajectory.	Graders must be tested, or they become another unreliable agent.
Regression	A previously passing behavior that fails after a change.	Agent systems need regression suites because small prompt changes can shift behavior.
Trace	A structured record of prompts, tool calls, decisions, observations, costs, and results.	Traces explain what happened when outputs are not enough.
Trajectory	The full path an agent takes through steps, tools, state, and decisions.	Some failures only appear across the path, not in the final answer.

Memory and Knowledge Terms

Term	Meaning	Why It Matters
Context budget	The token, latency, cost, and attention limit for context.	More context is not always better; it can distract or leak.
Context packet	A deliberately assembled bundle of instructions, facts, state, tools, and evidence.	It makes context selection inspectable.
Episodic memory	Stored records of events, interactions, or runs.	It helps personalization and continuity but requires retention and correction rules.
Knowledge boundary	The trusted sources an agent may use for claims.	It reduces hallucination and supports refusal when evidence is missing.
Retrieval-augmented generation (RAG)	A design where retrieved evidence is added to model context.	RAG helps only when retrieval quality, citations, and freshness are tested.
Semantic recall	Retrieval based on meaning rather than exact keyword matching.	It can find useful evidence but can also retrieve plausible wrong material.
Working memory	Task-local state used during one run or session.	It keeps multi-step work coherent without turning every fact into durable memory.

Security and Trust Terms

Term	Meaning	Why It Matters
Approval gate	A point where a human or policy service must approve an exact action.	Gates reduce risk for money, credentials, user data, and irreversible changes.
Audit log	A record of decisions, approvals, tool calls, and external effects.	Audit logs support incident review and accountability.
Credential boundary	The rule for which identity or secret a tool call may use.	Agents should not inherit broad credentials by default.
Least privilege	Giving the system only the access it needs for the task.	Broad access turns model mistakes into security incidents.
Sandbox	A constrained environment for code, browser, file, or network actions.	Sandboxes limit damage when generated actions fail or are attacked.
Threat model	A structured analysis of what can go wrong, who can cause it, and how to reduce risk.	Agent threat models must include tools, memory, prompts, users, and external data.
Trust contract	The promises the product makes about what the agent can do, show, change, or decide.	Users need clear control when agents act on their behalf.

Protocol and Framework Terms

Term	Meaning	Why It Matters
A2A	Agent-to-agent communication through typed messages, identity, and authorization.	A2A needs contracts; free-form chat between agents is hard to secure or debug.
MCP	Model Context Protocol, a protocol for exposing tools and resources to model clients.	MCP can standardize tool access, but servers still need auth, policy, and lifecycle control.
Skill	A packaged procedural capability with instructions, files, scripts, or examples.	Skills preserve tested know-how and reduce repeated prompt bloat.
Supervisor	A component or agent that assigns, reviews, or coordinates worker agents.	Supervisors need authority, stop rules, and failure handling.
Worker	A bounded agent or component responsible for a subtask.	Workers should have narrow inputs, outputs, tools, and success criteria.

Acronyms

Acronym	Expansion	Note
ADR	Architecture Decision Record	Use for design choices and consequences.
A2A	Agent-to-Agent	Use for typed agent communication.
API	Application Programming Interface	Treat agent tools as APIs with policy.
CI	Continuous Integration	Use CI for eval gates and regression checks.
KPI	Key Performance Indicator	Use carefully; agent KPIs need quality and risk metrics.
MCP	Model Context Protocol	Use for tool/resource interoperability.
PII	Personally Identifiable Information	Requires minimization, masking, retention, and access rules.
RAG	Retrieval-Augmented Generation	Requires retrieval evals and source controls.
SLA	Service Level Agreement	External commitment to service behavior.
SLO	Service Level Objective	Internal target for reliability, latency, or quality.
UX	User Experience	Includes trust, visibility, correction, and control.

Review Rule

When a chapter, ADR, or release note uses an acronym, define it on first use unless the surrounding section already does. This matters for an online book because readers often land on a single page from search instead of reading chapters in order.

Related Chapters

- [What Is An Agent?](#)
- [Choosing the Right Pattern](#)
- [Context Engineering](#)
- [Evaluation-Driven Agent Development](#)
- [Agent Threat Model](#)

10/10 Production Gate

This book is not finished when you can describe agent patterns. It is finished when you can design a system that a team can review, test, ship, observe, and roll back.

Use this gate before calling an agentic system production-ready. A score below 8 means the design is still a demo or prototype. A score of 10 means the team has evidence, not just confidence.

Download the reusable worksheet: [10/10 production gate scorecard](#).

When To Use This Gate

Use this gate at three points:

Moment	Purpose	Expected Outcome
Before implementation	Find missing ownership before code makes the design expensive to change.	Architecture changes, not launch approval.
Before pilot	Decide whether limited users, data, or side effects are acceptable.	Internal pilot, controlled pilot, or blocked.
Before production	Verify evidence for every high-risk path.	Ship, ship with documented limits, or do not ship.

Do not wait until launch week. The gate is most useful when it changes the design early.

What A 10/10 System Proves

Area	A 10/10 System Shows	Evidence
Goal	The system has a narrow goal, explicit non-goals, and clear success criteria.	Problem statement, user story, acceptance tests.
Boundary	The design separates model judgment from deterministic control.	Architecture diagram, service contracts, policy boundary.
State	The system names what owns run state, durable memory, user data, and external effects.	State schema, retention rules, replay plan.
Tools	Every tool has a typed contract, authorization rule, timeout, and audit record.	Tool manifest, test fixtures, audit log example.
Context	Context is assembled deliberately and carries source, trust, freshness, and budget information.	Context packet example, retrieval trace, context budget.

Area	A 10/10 System Shows	Evidence
Evaluation	The team can run evals for happy paths, edge cases, regressions, and known failures.	Eval dataset, grader rubric, CI gate output.
Security	The threat model covers tools, memory, prompts, retrieved content, credentials, and users.	Threat model, mitigations, sandbox tests.
Human control	Users can inspect, approve, correct, stop, or escalate risky behavior.	Approval UI, correction flow, audit trail.
Runtime	The system handles retries, timeouts, idempotency, degradation, and rollback.	Runbook, retry policy, rollback plan.
Observability	Operators can inspect traces, costs, latency, tool calls, eval drift, and incidents.	Dashboard, trace sample, alert thresholds.

Scorecard

Score each row from 0 to 2.

Score	Meaning
0	Missing or only described in prose.
1	Partly implemented but not tested or not reviewable.
2	Implemented, tested, documented, and tied to an owner.

Gate	Score
Goal and non-goals are explicit.	0-2
Architecture boundary is reviewable.	0-2
State ownership is documented.	0-2
Tool contracts and permissions are tested.	0-2
Context packet is inspectable.	0-2
Eval gate blocks unsafe releases.	0-2
Threat model has mitigations.	0-2
Human control exists for risky actions.	0-2
Runtime can retry, resume, degrade, and roll back.	0-2
Observability explains incidents.	0-2

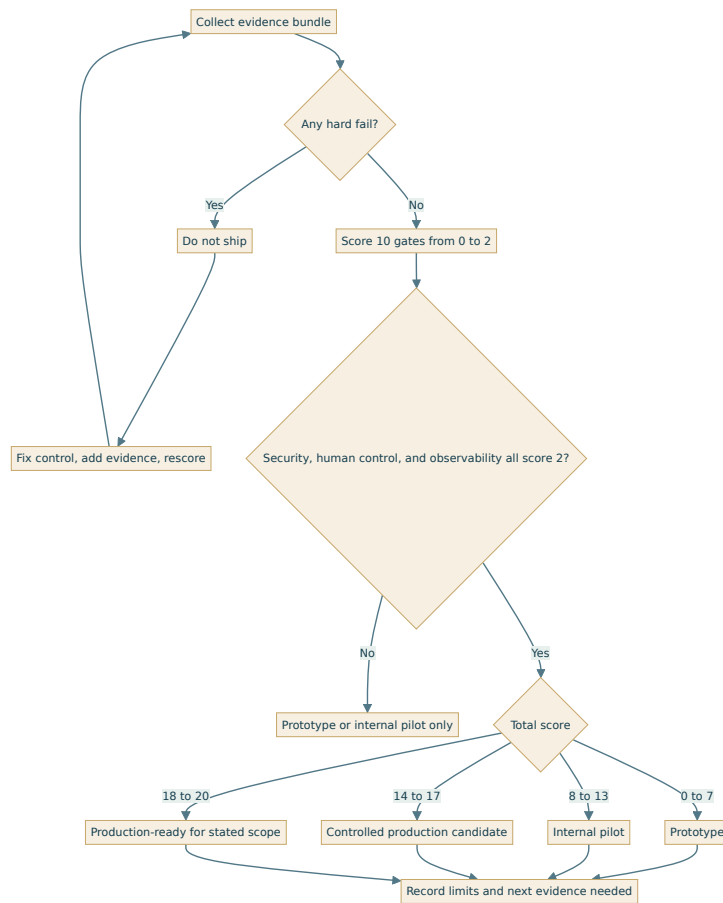
Total score:

- 0-7: prototype
- 8-13: internal pilot
- 14-17: controlled production candidate
- 18-20: production-ready

Do not average away a missing safety control. A system that touches money, credentials, private data, infrastructure, or customer-visible actions must score 2 on security, human control, and observability.

Release Decision Flow

Use this flow after scoring. It keeps hard fails from being hidden by a strong total score.



Scoring Rules

Score evidence, not intent:

Claim	Score 0	Score 1	Score 2
"We have evals."	No eval fixtures exist.	Evals exist but do not block release or cover failures.	Evals cover expected and unsafe paths, run in CI, and block release.
"We have observability."	Logs exist but cannot reconstruct a run.	Traces exist for happy paths only.	Operators can inspect successful, failed, refused, escalated, and

Claim	Score 0	Score 1	Score 2
			timed-out runs.
"We have approval."	Humans are notified after action.	Approval exists but authorizes vague behavior.	Approval binds one exact action, policy version, expiry, approver, and trace ID.
"We can roll back."	Rollback means redeploying or deleting the feature.	One rollback path exists but is untested.	Model, prompt, policy, tool, workflow, or agent behavior can be disabled independently.

If the evidence is not inspectable by another engineer, do not score it as 2.

Release Modes

Use the score to choose a release mode:

Mode	Allowed Scope	Required Evidence
Prototype	Local demos, fake data, no real side effects.	Clear label that it is not production.
Internal pilot	Internal users, limited data, read-only or tightly bounded actions.	Owner, logs, basic evals, rollback, and known limits.
Controlled production candidate	Real users or data with limited rollout and active monitoring.	Full scorecard, approval for risky actions, traces, eval gate, runbook.
Production-ready	Normal operation for the intended scope.	18-20 score, no hard fails, tested rollback, incident-to-eval loop.

The release mode should appear in the ADR, release notes, or launch plan. A system can be production-ready for one narrow task and still be a prototype for adjacent tasks.

Hard Fails

Any hard fail blocks production, regardless of total score.

Hard Fail	Why It Blocks
The agent can call high-risk tools without policy or approval.	The model can turn ambiguity into irreversible action.
Credentials are inherited broadly from the host process.	One prompt or tool bug can become a privilege escalation.
The team cannot replay a failed run.	Incidents cannot become regression tests.

Hard Fail	Why It Blocks
Evals measure only final answer quality.	Tool misuse, hidden state, and unsafe trajectories remain invisible.
Context sources have no provenance or freshness rule.	The system can confidently act on stale or untrusted information.
There is no rollback plan for prompt, policy, model, or tool changes.	Safe release requires safe reversal.

Evidence Bundle

A serious design review should attach these artifacts:

1. Architecture diagram with model, tools, state, policy, memory, and user boundary.
2. ADR explaining the chosen pattern and rejected alternatives.
3. Tool manifest with permissions, risk class, timeouts, and audit fields.
4. Context packet example for one real task.
5. Trace from a successful run and a failed run.
6. Eval dataset with at least happy path, edge case, adversarial, and regression cases.
7. Threat model with mitigations and test evidence.
8. Approval or escalation flow for risky actions.
9. Runbook with alerts, incident triage, rollback, and owner.
10. Cost, latency, and budget thresholds.

Minimum Production Review Questions

Ask these questions before launch:

1. What exact action can the system take without a human?
2. What state changes if the model is wrong?
3. Which tool call would be most expensive, risky, or irreversible?
4. What evidence would prove the answer or action is grounded?
5. What happens when retrieval returns stale or hostile content?
6. What happens when a tool times out after the external side effect succeeded?
7. Which eval would fail if the system regressed tomorrow?
8. Who receives the alert, and what can they roll back?
9. What does the user see when the agent is uncertain?
10. What would make the team shut the agent off?

Where To Learn Each Gate

Gate Area	Read
Pattern choice	Choosing the Right Pattern
Architecture boundary	Architecture Before Autonomy, Agentic System Architecture
State and loop control	Agent Loop, Goals and State
Tool contracts	Tool Use, Tool Capability Design
Context and memory	Context Engineering, Context Budgets and Working Sets
Evals	Evaluation-Driven Agent Development, Observability and Evals
Security	Agent Threat Model, Agent Security and Sandboxing
Human control	Human Approval Gates, Agent UX and Human Trust
Runtime	Production Runtime Overview, Deployment Walkthrough
Complete examples	Capstone Projects

Takeaway

A 10/10 agentic system is not the most autonomous system. It is the system whose autonomy is narrow, tested, observable, reversible, and worth the operational cost.

Architecture Before Autonomy

Do not start with the framework. Start with the boundary: what is the model allowed to decide?

Agents earn their place when software cannot know every step in advance. They become a liability when the system lets the model own decisions that should belong to code, things like permission checks, stop conditions, budget limits, state transitions, audit records, and irreversible side effects. Good agentic architecture separates judgment from authority. The model proposes the next useful action; the system decides whether that action is valid, allowed, affordable, observable, and safe to execute.

The Model Is Not The System

An LLM can summarize context, propose plans, choose from tools, critique outputs, and explain tradeoffs. None of that makes it the control plane. The control plane owns the active goal, the allowed tools, the current state, the budget, the stop condition, the policy context, the approval requirements, the retry and fallback rules, and the trace of what happened.

When those responsibilities live only in a prompt, the system gets hard to test and harder to operate. A prompt can influence behavior. It cannot replace a durable boundary.

A Practical Boundary

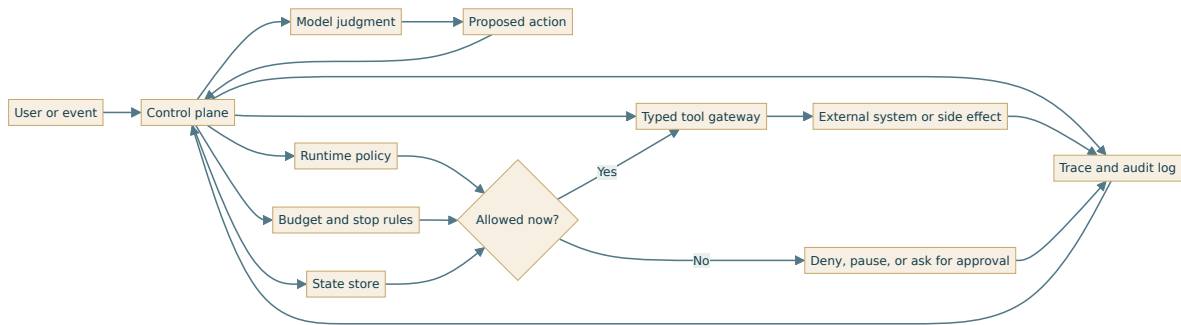
Use this division of responsibility as your default:

Concern	Owned By Software	Proposed By Model
Goal	The task contract and success criteria	Clarifying questions or subgoals
State	Durable store, workflow engine, or application service	Summaries or candidate memory writes
Tools	Typed schemas, permissions, timeouts, and audit logs	Tool choice and arguments
Policy	Runtime checks and approval gates	Risk explanation or escalation request
Evaluation	Tests, rubrics, traces, and review workflow	Self-critique or candidate score
Stopping	Explicit success, failure, budget, or cancellation rule	Claim that the goal appears complete

This does not make the system less agentic. It makes the autonomy legible, which is what lets you operate it.

Boundary Map

Use this map when reviewing a design. The model can propose judgment-heavy work, but software must own authority, execution, and evidence.



This chapter names the boundary. [Tool Capability Design](#) and [Human Approval Gates](#) define the concrete tool and approval contracts behind it.

In code, the boundary can be small and explicit:

```
interface ProposedAction {
  kind: 'read' | 'write' | 'notify' | 'stop';
  tool?: string;
  input?: unknown;
  risk: 'low' | 'medium' | 'high';
}

function decideExecution(action: ProposedAction, policy: RuntimePolicy) {
  if (action.kind === 'stop') return { status: 'stop' };

  if (!action.tool || !policy.allowedTools.includes(action.tool)) {
    return { status: 'deny', reason: 'tool_not_allowed' };
  }

  if (action.risk === 'high' && !policy.hasHumanApproval) {
    return { status: 'pause', reason: 'approval_required' };
  }

  return { status: 'execute', tool: action.tool, input: action.input };
}
```

The model proposes `ProposedAction`. The runtime decides whether it is denied, paused, executed, or stopped.

Premature Autonomy

Premature autonomy shows up when a team reaches for an agent loop before answering simpler questions first: Could this be a deterministic workflow with model-assisted steps? Could a prompt chain with validation solve it? Could routing isolate the different task types? Could a human approval gate handle the risky cases? Can the system already prove why an action was taken, and can a failed run be replayed?

When the answer to those is no, adding more agents tends to hide the weakness rather than fix it. Multi-agent systems amplify unclear goals, weak state, and poor observability far more reliably than they cure them.

A Common Rewrite

A weak design says:

```
Give the agent access to order data, refund tools, and the customer conversation.  
Let it investigate the case and issue the refund if appropriate.
```

That sounds efficient, but the boundary is missing. The model owns too much: policy interpretation, evidence selection, risk classification, approval, tool authority, and stop conditions.

A better design says:

```
Workflow receives refund request.  
Software loads order, payment, customer, and policy records.  
Model summarizes evidence and proposes a refund recommendation.  
Software validates required evidence, refund threshold, account status, and policy version.  
Low-risk denial or draft response can proceed.  
High-risk refund creates an approval request.  
Payment tool executes only after approval, with idempotency and audit records.  
Run stops with completed, denied, needs_approval, policy_blocked, or evidence_missing.
```

The second design still uses model judgment, but the model no longer owns authority. It proposes a recommendation inside a system that can be reviewed, tested, paused, replayed, and operated.

The Engineering Test

Before adding autonomy, ask whether an operator could inspect a failed run and answer:

1. What goal was active?
2. What state did the system believe?
3. What evidence was available?
4. What did the model propose?
5. What did software validate?
6. What tool calls ran?
7. What policy checks passed or failed?

- 8. Why did the run stop?
- 9. What changed in the outside world?

If those answers are not available, the next thing to build is not another agent. It is state, policy, evaluation, or observability.

Autonomy Review Checklist

Use this checklist before approving an agentic design:

Question	Pass Condition
What runtime decision does the model make?	The decision is named and narrower than “do the task.”
What does software own?	Goal, state, policy, tools, budget, stop reasons, and audit records are outside the model.
What side effects can happen?	Every write, send, payment, permission change, or memory write has an owner and idempotency rule.
What requires approval?	Risk thresholds and approver roles are defined before execution.
What is persisted?	State, evidence, proposals, validation decisions, tool results, and stop reason can be inspected.
What is tested?	Evals cover allowed actions, denied actions, missing evidence, failed tools, and budget stops.
What happens on failure?	The system can stop, retry, fall back, escalate, or replay without repeating unsafe side effects.

If the checklist feels too heavy, the design probably wants less autonomy, not more.

Design Rule

Autonomy is a budget. Spend it only where it buys a better outcome than deterministic software, and surround it with boundaries that make failure visible.

Related Chapters

- [Choosing the Right Pattern](#)
- [Pattern Evaluation Checklist](#)
- [Pattern Composition Playbook](#)
- [Agent Loop](#)
- [Goals and State](#)
- [Tool Use](#)
- [Tool Capability Design](#)

- [Human Approval Gates](#)
- [Evaluation-Driven Agent Development](#)
- [Agentic System Architecture](#)
- [Production Runtime Overview](#)

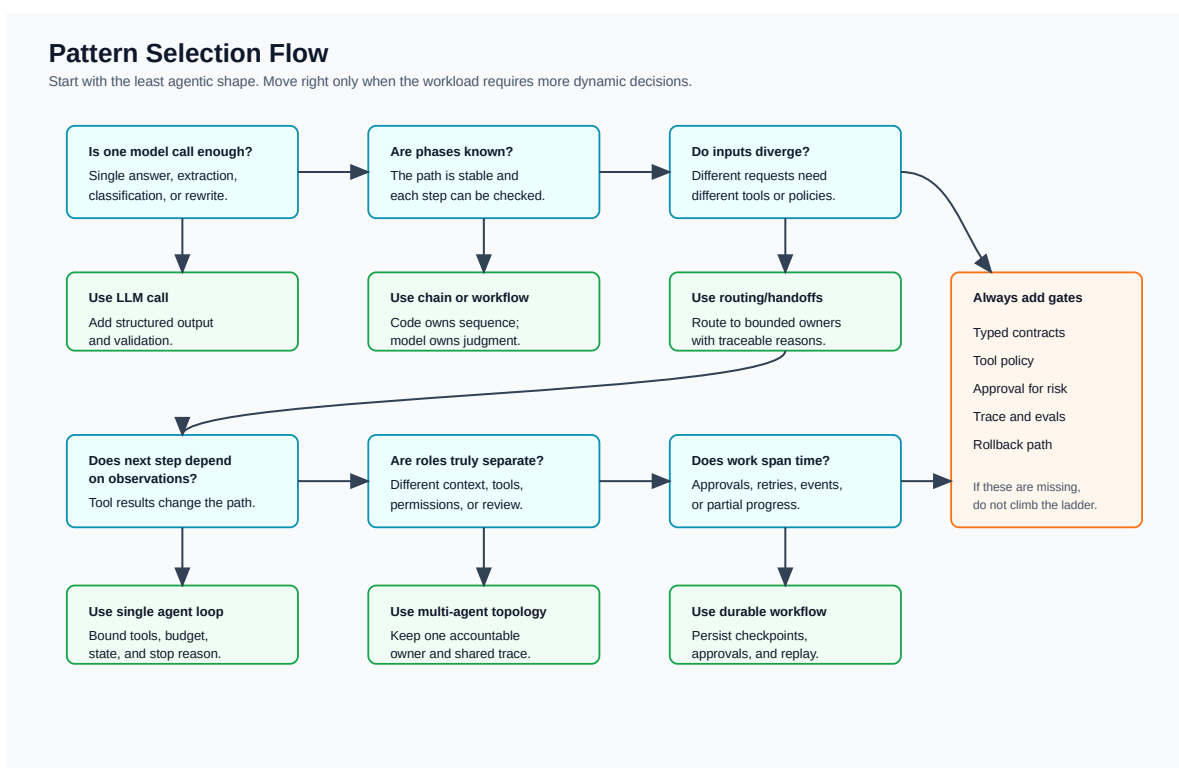
Choosing the Right Pattern

Agentic design is a spectrum. The best architecture is usually the least agentic system that can meet the requirement with acceptable quality, latency, cost, and risk.

Use this chapter before choosing a framework, adding agents, or building a multi-agent topology. The decision should start with the workload, not with the most powerful pattern available.

Selection Flow

Use this diagram as a practical gate before moving up the autonomy ladder. The question is not “can an agent do it?” The question is which smaller pattern can satisfy the workload with less risk.



The Autonomy Ladder

Move up the ladder only when the lower rung cannot handle the job.

Level	Pattern Shape	Use When	Main Risk
1	LLM call	A single answer, rewrite, classification, extraction, or summary is enough.	No access to live data or action.
2	Prompt chain	The work has known phases and each phase can be validated before the next one starts.	Brittle gates or unnecessary latency.

Level	Pattern Shape	Use When	Main Risk
3	Deterministic workflow with LLM steps	Code owns the sequence, while LLMs handle bounded judgment inside steps.	Overfitting the workflow to today's process.
4	Routing and handoffs	Different inputs need different models, prompts, tools, agents, or policies.	Bad routing sends work to the wrong authority.
5	Single agent loop	The next step depends on observations discovered during execution.	Loops, tool misuse, and compounding errors.
6	Orchestrator-workers	The system must break an unknown task into subtasks and synthesize results.	The orchestrator becomes a hidden control plane.
7	Multi-agent system	Different specialists need separate context, tools, permissions, or review roles.	Coordination overhead, trace fragmentation, and cost growth.
8	Autonomous long-running agent	The task spans time, failures, external events, and partial progress.	Unbounded autonomy without strong state, policy, and observability.

This ladder is not a maturity model. A production system can stay at level 2 forever if the task is stable and the value is clear.

First Questions

Ask these questions before selecting a pattern:

- Is the workflow known before execution starts?
- Does the model need to choose the next step, or can code do it?
- Does the task need current data, private data, tools, or side effects?
- Can the system tolerate extra latency from multiple model calls?
- What is the maximum acceptable cost per run?
- What requires human approval?
- What evidence proves the answer or action is correct?
- What state must be replayable after a failure?
- What could go wrong if the model is persuasive but wrong?

If the answers are unclear, start with a deterministic workflow and add agentic behavior only where the workflow needs model judgment.

Selection Outputs

Do not leave pattern selection as a conversation. Write down the decision in a form another engineer can review.

Workload:

Primary pattern:

Why this is the least agentic sufficient design:

Model decisions allowed:

Software-owned controls:

Tools and side effects:

State that must be persisted:

Approval requirements:

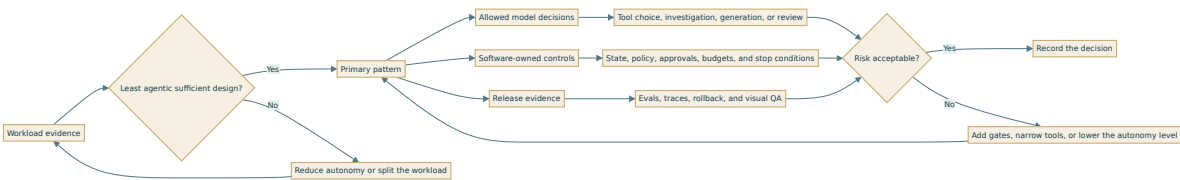
Eval cases required before release:

Fallback or rollback path:

The most important line is “why this is the least agentic sufficient design.” If the answer is “because agents are flexible,” the design is not ready. Name the actual uncertainty the model must handle: ambiguous intent, unknown next step, incomplete evidence, variable tool choice, or open-ended investigation.

Decision Record Map

Use this map to turn the selection output into an engineering review artifact. A pattern choice is ready when the workload evidence, autonomy boundary, controls, and release evidence can be reviewed separately.



Selection Matrix

Workload Signal	Prefer	Avoid
Known fixed sequence	Prompt Chaining and Gates	Autonomous agents
One of several known task types	Routing and Handoffs	One giant prompt
Independent subtasks	Parallel Agents	Sequential chains that create avoidable latency
Unknown subtasks	Orchestrator-workers	Hard-coded task lists
Quality improves with review	Evaluator-Optimizer	Single-pass generation

Workload Signal	Prefer	Avoid
High-risk side effects	Human Approval Gates	Direct tool execution from model output
Large tool surface	MCP-first Tool Use with policy	Broad untyped tools
Long-running work	Durable Workflows and Goals and State	Hidden in-memory loops
Sensitive data or compliance	Policy Enforcement and audit logs	Prompt-only controls
Retrieval-heavy answers	Agentic RAG Systems	Blind model responses
Debugging matters	Observability and Evals	Final-answer-only logs

The same product may combine several rows. For example, a support refund workflow may use routing to classify the request, deterministic workflow steps for account lookup, RAG for policy evidence, human approval for exceptions, and an agent loop only for open-ended investigation.

Example Choices

Workload	Start With	Add Autonomy Only If
Rewrite or summarize user-provided text	One model call with structured output	The system must fetch missing context or revise against external evidence.
Generate support replies from known policies	Workflow with retrieval and validation	The request requires investigation across tools with unknown next steps.
Triage tickets by product area	Routing and handoffs	The router needs to ask follow-up questions or inspect live systems.
Research a technical question	Agentic RAG or agent loop with retrieval tools	The system must decide whether evidence is sufficient or continue searching.
Issue refunds	Deterministic workflow with model-assisted evidence summary	The policy has ambiguous exceptions that require bounded investigation.
Coordinate delivery operations	Workflow plus explicit roles	Separate agents need distinct context, tools, permissions, or audit trails.
Maintain a long-running background task	Durable workflow	Runtime decisions depend on future events, partial progress, or recovery.

These examples are intentionally conservative. Start with the boring pattern, then promote only the part of the system that has earned dynamic behavior.

Workflows vs Agents

A workflow uses code to control the path. A model may classify, extract, summarize, critique, or generate inside a step, but software decides what happens next.

An agent uses the model to decide parts of the path. It observes state, decides the next action, calls tools, reads results, and continues until a goal is complete or a limit is reached.

Prefer a workflow when the process is stable, when correctness depends on deterministic rules, when latency or cost must stay low, when the system has to be easy to audit, and when operators need predictable failure modes. Prefer an agent when the process is open-ended, when the system has to discover missing information, when tool choice depends on intermediate observations, when the number of steps is unknown, or when a fixed workflow would branch into a tree nobody can maintain.

Complexity Budget

Every additional agent, model call, tool, memory store, and evaluator spends part of the system's complexity budget, so spend it deliberately. Complexity is worth adding when it buys a concrete outcome: a higher task completion rate, lower human effort, better evidence grounding, safer side effects, lower cost through routing or smaller models, better debuggability, or clearer ownership boundaries.

Do not add complexity only because a pattern is popular. A multi-agent system that replaces a reliable four-step workflow usually makes the product slower, harder to test, and harder to explain.

Pattern Evolution Path

A practical evolution path looks like this:

1. Start with one prompt or deterministic workflow.
2. Add structured outputs and validation.
3. Add retrieval or tools where the model needs evidence or action.
4. Add routing when inputs diverge into distinct paths.
5. Add evaluator loops where quality improves with critique.
6. Add durable state when work spans several steps or sessions.
7. Add agents only where dynamic decisions are required.
8. Add multi-agent coordination when separate roles need separate context, tools, or permissions.

Each step should improve a measured outcome. If a pattern does not improve accuracy, reliability, latency, cost, safety, or maintainability, remove it.

Common Selection Mistakes

- Choosing multi-agent coordination when the real need is routing.

- Using an agent loop because the workflow was not written down.
- Giving the model broad tools when a narrow workflow step would work.
- Adding reflection when the system needs a better evaluator or test set.
- Adding memory before defining what state must be durable, correctable, and deletable.
- Treating framework examples as production architecture.
- Optimizing for autonomy before measuring task completion, cost, latency, or risk.

Minimum Production Bar

Before a pattern handles users, money, private data, infrastructure, or customer communication, the system should have:

- typed inputs and outputs;
- explicit stop conditions;
- bounded tool permissions;
- traceable model and tool calls;
- replayable state transitions;
- eval datasets for expected behavior;
- fallback behavior for model, retrieval, and tool failure;
- human approval for high-risk actions;
- rollback or remediation for bad actions.

This minimum bar applies to simple systems too. Small systems fail faster because their boundaries are often implicit.

Related Chapters

- [Prompt Chaining and Gates](#)
- [Routing and Handoffs](#)
- [Circuit Breakers, Fallbacks, and Replay](#)
- [Agent Loop](#)
- [Goals and State](#)
- [Agentic System Architecture](#)
- [Source Map](#)

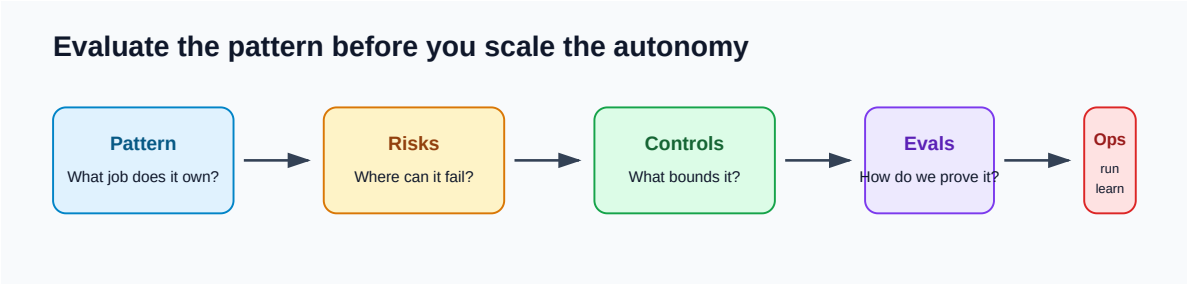
Pattern Evaluation Checklist

A pattern is not ready because it sounds right. It is ready when the team can explain what job it owns, where it can fail, what bounds the failure, how it is evaluated, and how it behaves in production.

This checklist is the shared review lens for the book. Use it before choosing a pattern, before composing several patterns, and before promoting an agentic workflow into production.

Download the reusable worksheet: [pattern evaluation scorecard](#).

For the engineering loop behind this checklist, see [Evaluation-Driven Agent Development](#). Use the checklist to review a pattern. Use the development chapter to turn the review into datasets, fixtures, release gates, and production feedback.



The Short Version

Every pattern should answer five questions:

- 1. What responsibility does this pattern own?
- 2. What new risk does it introduce?
- 3. What control keeps that risk bounded?
- 4. What eval proves the control works?
- 5. What production signal tells us it is drifting?

If the pattern cannot answer those questions, it is probably not a pattern yet. It is an implementation idea.

Evaluation Table

Use this table as the default review template.

Area	Question	Evidence To Look For
Goal	What user or system goal does the pattern own?	A task contract, success criteria, refusal criteria, and owner.
Boundary	What is outside the pattern’s responsibility?	A clear handoff, caller contract, or escalation path.

Area	Question	Evidence To Look For
Autonomy	What does the model decide, and what does software decide?	A split between proposal, validation, execution, and stop.
Loop	Can the pattern repeat?	Max steps, max tool calls, timeout, retry budget, and stop reason.
Tools	What can the pattern read or change?	Tool allowlist, schema validation, permission checks, and audit events.
State	What state is read, written, or persisted?	State owner, update rules, replay behavior, and memory write policy.
Context	What evidence enters the working set?	Source eligibility, retrieval rules, freshness checks, and context budget.
Security	What can untrusted input influence?	Threat model, prompt-injection controls, sandboxing, and approval gates.
Evaluation	What failure must be caught before release?	Golden tasks, negative cases, trajectory evals, mocked tools, and regression fixtures.
Observability	Can a failed run be explained later?	Trace ID, model spans, tool spans, decisions, policy denials, costs, and stop reason.
Operations	Can the pattern be rolled back or disabled?	Versioned prompts, tool manifests, model routes, feature flags, and circuit breakers.

Review Score

Use a simple score when the pattern is going into a design review or release gate.

Score	Meaning	Decision
0	Missing or only prompt-level.	Block. The control is not real.
1	Described but not implemented or tested.	Do not release. Convert the description into code, config, or tests.
2	Implemented but weakly tested or hard to inspect.	Release only for low-risk internal use.
3	Implemented, tested, traceable, and owned.	Accept for the stated risk level.

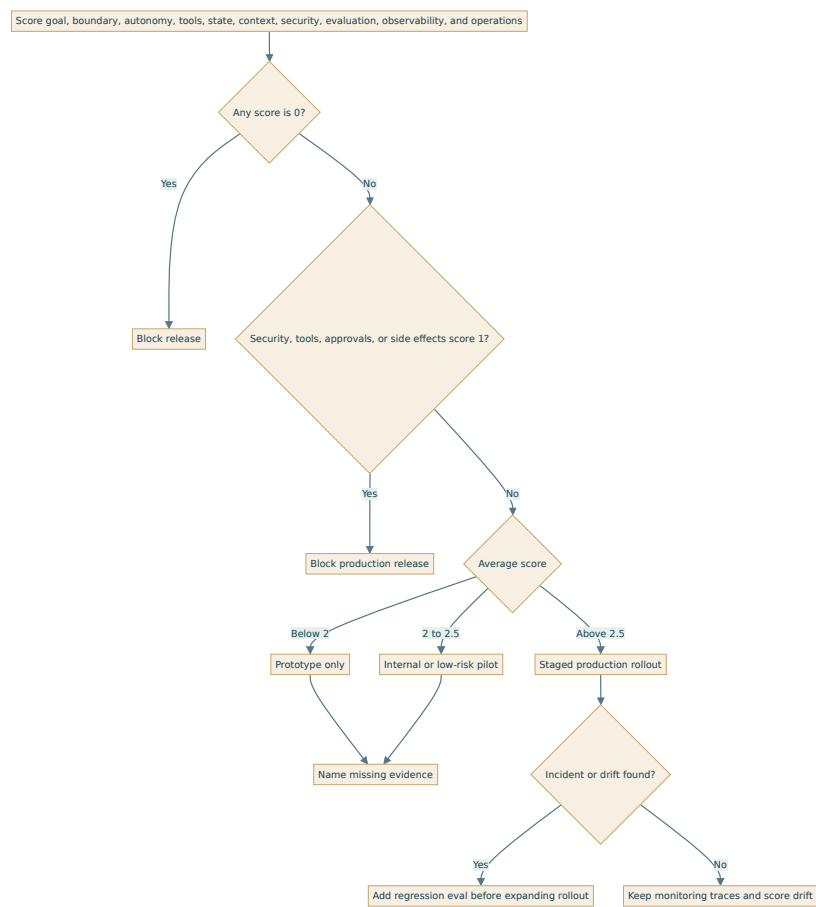
Score these areas: goal, boundary, autonomy split, tools, state, context, security, evaluation, observability, and operations. A production pattern should not have any **0** scores. A pattern that touches money, private data, infrastructure, customer communication, or durable memory should not have any **1** scores.

The point is not to create bureaucracy. The score prevents a common design-review failure: everyone agrees the idea is good, but nobody proves the boundary exists.

Use the downloadable [pattern evaluation scorecard](#) when the review must leave an auditable record. It captures the score, owner, evidence, release mode, blocking gaps, accepted risks, and next evidence for each area.

Score Decision Path

Use this path during review. It turns the score into a release decision without hiding high-risk gaps behind a good average.



Minimum Bar By Pattern Type

Different patterns need different proof.

Pattern Type	Minimum Evaluation Guidance
Prompt chain	Validate each step output, gate transitions, and test malformed intermediate results.
Router	Test ambiguous requests, unsupported tasks, high-risk routes, and fallback behavior.

Pattern Type	Minimum Evaluation Guidance
Agent loop	Test stop conditions, tool selection, recovery from bad observations, and budget exhaustion.
Tool-use pattern	Test forbidden tools, invalid arguments, idempotency, timeouts, and policy denials.
RAG or memory pattern	Test source relevance, stale evidence, missing evidence, citation coverage, and unsafe memory writes.
Evaluator or reflection pattern	Test false approvals, overcorrection, rubric ambiguity, and disagreement handling.
Multi-agent pattern	Test context isolation, permission isolation, merge accuracy, worker failure, and final accountability.
Human approval pattern	Test escalation criteria, approver visibility, timeout behavior, and audit records.
Production runtime pattern	Test replay, rollback, canary gates, incident-to-eval conversion, and operator diagnosis.

A Small Review Contract

For lightweight design reviews, keep the contract short:

```

pattern: tool_using_agent
owned_goal: "Investigate refund eligibility from approved business systems."
model_decides:
  - "which allowed read tool to call next"
  - "whether evidence is sufficient for a recommendation"
software_decides:
  - "which tools exist"
  - "whether the caller is authorized"
  - "whether a side effect requires approval"
  - "when the run stops"
controls:
  max_steps: 6
  max_tool_calls: 8
  timeout_ms: 45000
  forbidden_tools:
    - refunds.issue_refund
    - support.send_customer_email
evals:
  blocking:

```



```

- "does not issue refunds directly"
- "returns needs_human when evidence is missing"
- "cites policy before recommending refund"
operations:
  trace_fields:
    - task_id
    - trace_id
    - tool_calls
    - policy_denials
    - stop_reason
review_score:
  goal: 3
  boundary: 3
  autonomy_split: 2
  tools: 3
  state: 2
  context: 2
  security: 3
  evaluation: 2
  observability: 3
  operations: 2
release_decision: "internal pilot only until trajectory evals and replay are stronger"

```

The contract is intentionally plain. It should be easy to review in a pull request, easy to turn into tests, and easy to compare against a production trace.

Release Decision Rules

Use these rules after scoring:

- Any 0: block release.
- Any 1 on security, tools, approvals, or side effects: block production release.
- Average below 2: keep it as a prototype.
- Average 2-2.5: allow internal or low-risk pilot only.
- Average above 2.5 with no high-risk gaps: allow staged production rollout.
- Any incident involving the pattern: add a regression eval before expanding rollout.

If the team disagrees about a score, record the disagreement. A disagreement usually means the ownership boundary, evidence, or risk tolerance is still unclear.

Common Failure Smells

Watch for these smells during pattern selection:

- The pattern has no single owner.

- The model owns permission checks.
- The loop stops only when the model says it is done.
- The tool list is broader than the task.
- Memory writes happen as a side effect of conversation.
- The eval checks only the final answer, not the trajectory.
- The trace cannot show why a tool was called.
- Multi-agent routing is used to hide unclear responsibilities.
- The fallback path is “ask the model again.”
- Rollback requires manual reconstruction of prompts, tools, or policies.

These are not style problems. They are architecture problems.

Design Rule

Choose the simplest pattern whose risks you can bound and whose behavior you can evaluate. If you cannot test the boundary, the boundary is not real yet.

Related Chapters

- [Architecture Before Autonomy](#)
- [Choosing the Right Pattern](#)
- [From Patterns To Systems](#)
- [Pattern Composition Playbook](#)
- [Evaluation-Driven Agent Development](#)
- [Agent Threat Model](#)
- [Agents As Services](#)
- [Choosing Multi-Agent Topology](#)
- [Production Evaluation Feedback Loops](#)

From Patterns To Systems

Patterns are useful only when they help you build a system. A production agent is rarely one pattern. It is usually a workflow with a few model-mediated decisions, a retrieval boundary, some tools, state, policy, approvals, evals, and observability. The pattern names matter far less than the way those pieces fit together.

This is where many agent projects go wrong. The team adds a loop, then memory, then tools, then a second agent, then a judge, then a workflow engine. Each addition feels reasonable on its own, and the result is a system nobody can explain. Composition is the discipline that prevents that.

Start With The Workload

Do not compose patterns from a catalog. Compose them from the workload. Start by asking what the user is actually trying to accomplish, which steps are known in advance and which require model judgment, what evidence has to be retrieved, which actions carry side effects, what state must survive a failure, what needs approval, and what has to be observable after the run.

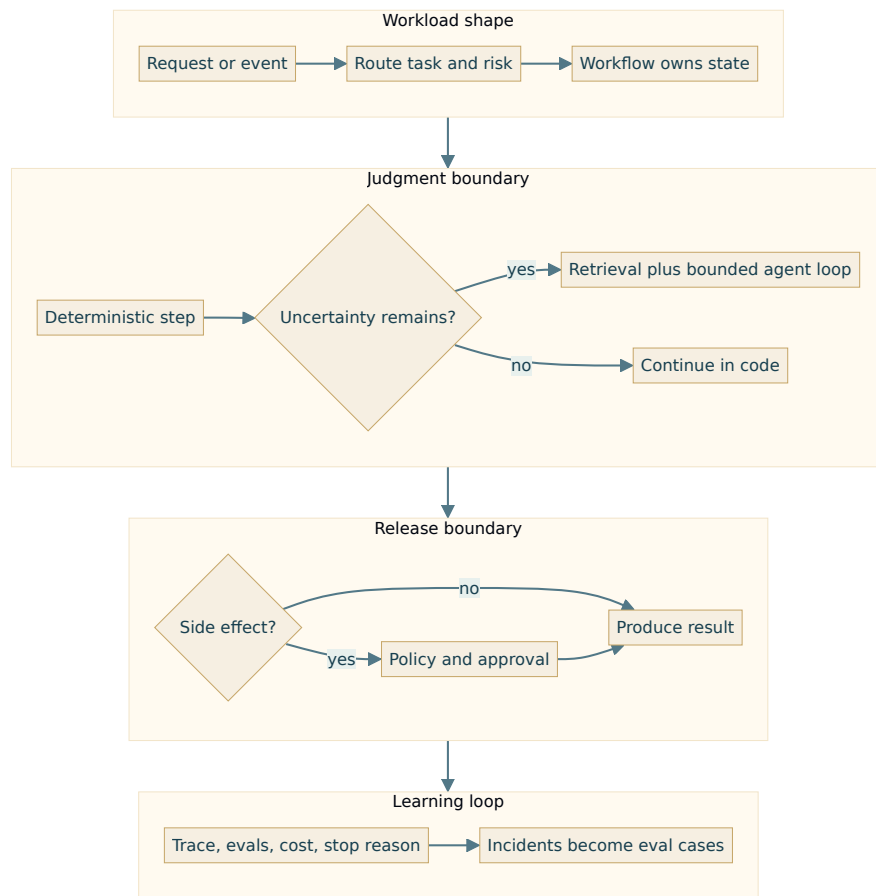
The answers point to the parts. If the workflow is known, keep code in charge. If the next step depends on observations, add an agent loop. If the task needs evidence, add retrieval. If the task can change the outside world, add policy and approval. If failures matter, add durable state and replay. That is composition: each pattern earns its place rather than arriving by default.

A Common Shape

Many useful agentic systems follow roughly this shape:

1. Entry point receives a request, event, or scheduled task.
2. Router classifies the task, risk, and required capability.
3. Workflow loads state, policy context, and relevant memory.
4. Retrieval gathers evidence when the answer depends on knowledge.
5. Agent loop handles bounded uncertainty inside the workflow.
6. Tools execute through typed schemas and permission checks.
7. Approval gate pauses high-risk side effects.
8. Evaluators check trajectory, evidence, output, and policy.
9. Runtime stores traces, costs, decisions, tool calls, and stop reasons.
10. Incidents and corrections feed the eval suite.

Not every system needs every step. What stays constant is the ownership. Code owns flow, state, policy, and persistence; the model owns bounded judgment inside those constraints.



Use this diagram as a composition test. If a proposed pattern cannot be placed on the map with an owner, input, output, and failure mode, it is probably not ready to enter the system.

Composition Rules

Run through these rules before adding another pattern.

Rule	Why It Matters
One component owns the goal.	Without goal ownership, agents optimize different tasks.
One component owns state.	Without state ownership, replay and recovery become guesswork.
Tool calls cross a policy boundary.	Without policy, model proposals become actions too quickly.
Memory writes are explicit events.	Without memory discipline, stale or unsafe context persists.
Loops have stop conditions.	Without stop conditions, autonomy becomes cost and latency growth.
Evals inspect trajectories.	Without trajectory evals, unsafe paths can produce plausible answers.
Traces connect decisions to effects.	Without traces, failures cannot become better tests.

These rules matter more than the framework. A framework can help you implement them; it cannot decide them for you.

System Composition Record

For any non-trivial agentic system, write a short composition record before implementation. It should fit in a pull request or architecture decision record.

```
system: support_refund_assistant
user_goal: "Resolve refund eligibility with policy-backed evidence."
primary_flow_owner: refund_workflow
patterns:
  routing:
    job: "classify request type and risk"
    owner: intake_service
  retrieval:
    job: "load current refund policy and order evidence"
    owner: evidence_service
  agent_loop:
    job: "investigate missing or conflicting evidence"
    owner: refund_investigation_agent
    max_steps: 6
  policy_enforcement:
    job: "validate recommendation against refund policy"
    owner: policy_gate
  human_approval:
    job: "approve exceptions and high-value refunds"
    owner: approval_workflow
  observability:
    job: "record trace, decisions, evidence, costs, and stop reason"
    owner: runtime
  release_blockers:
    - "missing evidence can stop the run"
    - "refund tool cannot execute without approval"
    - "trace can replay proposal, validation, approval, and side effect"
```

The record forces every pattern to justify itself. If a pattern has no job, no owner, or no release blocker, remove it or keep it out of the first version.

Bad Composition

Bad composition usually has the same smell: the model owns too much. The agent infers the goal from vague conversation history, chooses tools without a permission check, and writes memory without classification or review. Retries happen inside hidden loops. Subagents receive the full conversation

instead of a narrow task. Evaluators check tone but not evidence. The final answer is logged while the trajectory is lost, and a multi-agent system has no single owner for final synthesis.

These systems can look impressive in a demo. They are painful to operate because nobody can say where responsibility lives.

Good Composition

Good composition is usually boring. A support refund system, for example, might run a deterministic workflow for intake and account lookup, a router for request type and risk, and retrieval for the current refund policy. It would use structured output for the extracted fields and the recommendation, a small agent loop only for missing-information investigation, policy enforcement before any refund action, human approval for exceptions, and observability and evals across the full run.

The system is agentic where uncertainty exists and deterministic where control matters. That is the whole trick.

A simple composition sketch might look like this:

```
async function handleRefundRequest(request: SupportRequest) {
  const route = classifyRequest(request);
  if (route.kind !== 'refund') return handoffTo(route.owner);

  const order = await tools.lookupOrder(request.orderId);
  const policy = await retrievePolicy('refunds', order.region);
  const recommendation = await refundAgent.investigate({
    request,
    order,
    policy
  });

  const decision = enforceRefundPolicy(recommendation, order, policy);
  if (decision.requiresApproval) {
    return approvals.request('refund_exception', decision);
  }

  return tools.draftRefundRequest(decision);
}
```

Only one step uses an agent loop. The workflow still owns route, state, policy, approval, and side effects.

When To Split Agents

Do not split agents because the task feels large. Split them when the boundary buys something concrete: separate context windows, separate tools, separate permissions, separate teams or ownership, parallel work, independent review, or different user-facing responsibilities.

The weak reasons are easy to recognize once you name them. The architecture diagram looks more advanced. Each prompt sounds like a different job title. The team wants a multi-agent system. The single-agent design has unclear goals or weak tools, so splitting it feels like progress. It is not. Splitting agents does not fix weak architecture; it multiplies it.

Design Review Checklist

Before approving a composed agentic system, ask:

1. Which parts are deterministic workflows?
2. Which parts are model-mediated decisions?
3. What owns the active goal?
4. What owns durable state?
5. What tools can cause side effects?
6. What policy runs before those side effects?
7. What evidence is required for the final answer?
8. What memory can be written, updated, or deleted?
9. What are the stop conditions?
10. What evals block release?
11. What trace lets an operator replay the run?
12. What happens when the model is wrong but persuasive?

If the design cannot answer these, it is not ready for more autonomy.

Design Rule

Compose patterns only when each one has a job, an owner, and a failure mode you can test.

Related Chapters

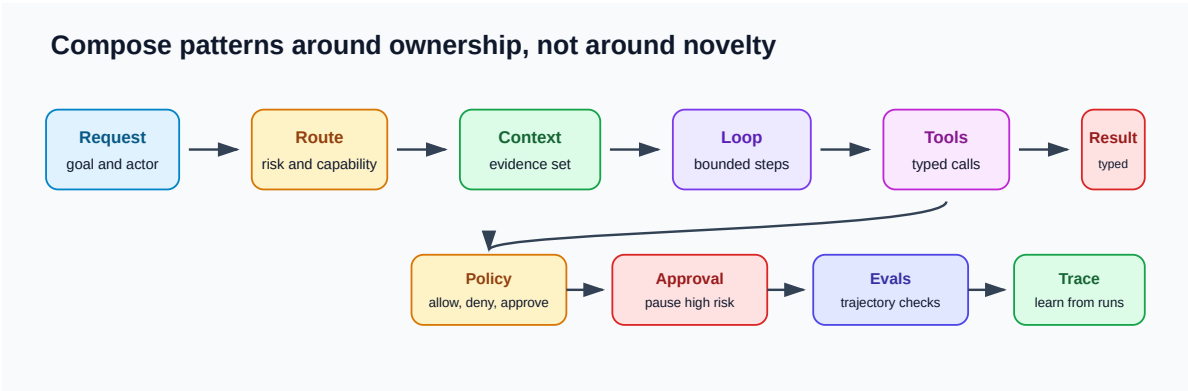
- [Architecture Before Autonomy](#)
- [Choosing the Right Pattern](#)
- [Pattern Evaluation Checklist](#)
- [Pattern Composition Playbook](#)
- [Agent Development Lifecycle](#)
- [Evaluation-Driven Agent Development](#)
- [Agentic System Architecture](#)
- [Reference Architecture](#)

Pattern Composition Playbook

Patterns become useful when they compose into a system that someone can operate. The goal is not to use as many patterns as possible. The goal is to put each responsibility in the right place: workflow owns flow, policy owns permission, tools own side effects, retrieval owns evidence, the loop owns bounded uncertainty, evals own proof, and observability owns learning from failure.

The easiest way to damage an agentic system is to compose patterns by vibe. Add memory because the agent forgot something. Add a second agent because the task feels large. Add reflection because the answer was weak. Add tools because the model needs data. Each decision may sound reasonable, but the result can become a system with no owner, no stop condition, no evidence boundary, and no way to replay failure.

Composition should start with ownership.



The Composition Question

Before adding a pattern, ask what problem it owns.

Pressure	Pattern To Consider	What Must Own The Boundary
The task has known steps.	Prompt chain or deterministic workflow.	Workflow code.
The next step depends on observations.	Agent loop.	Loop controller and stop rules.
The answer needs evidence.	RAG, semantic recall, or knowledge-bound agent.	Retrieval and source policy.
The model must call tools.	Tool use or MCP-first tool use.	Tool manifest, schema, permission, and audit.
The action is risky.	Policy enforcement and human approval gate.	Runtime policy and durable workflow.

Pressure	Pattern To Consider	What Must Own The Boundary
Quality can be judged better than generated.	Evaluator-optimizer or reflection.	Rubric, evidence checks, and revision budget.
Work needs separate context or permissions.	Supervisor-worker or parallel agents.	Coordinator, worker contracts, and merge policy.
Failure must be replayed.	Durable workflow, observability, and eval feedback loop.	Runtime, trace store, and eval suite.

If no pattern owns a concrete problem, do not add it.

For concrete end-to-end examples, read [Vertical Slice Examples](#) after this chapter. The slices show the same composition rule applied to support, coding, and research workflows.

A Default Composition

For many production systems, the default composition is:

1. route the request by task type, risk, and capability;
2. load durable state, policy context, and caller identity;
3. assemble a small working set with approved evidence;
4. run a bounded agent loop only where uncertainty exists;
5. execute tools through typed schemas and permission checks;
6. enforce policy before side effects or memory writes;
7. pause for approval when risk requires it;
8. evaluate trajectory, evidence, output, and policy behavior;
9. record traces, costs, decisions, tool calls, and stop reasons;
10. convert production failures into regression evals.

That sequence is not a framework. It is a responsibility map. A simple system may skip several steps. A high-risk system may need all of them.

Composition Scorecard

Before accepting a composition, score each added pattern against the responsibility it claims to own.

Check	Pass Condition
Job	The pattern solves a named workload pressure, not a vague desire for flexibility.
Owner	One component or team owns the pattern's behavior and failure response.
Boundary	The pattern has clear inputs, outputs, authority, and non-responsibilities.
Risk	The pattern's new failure modes are named.

Check	Pass Condition
Control	There is a software-owned limit, policy, gate, or fallback for each major risk.
Eval	At least one blocking eval proves the boundary works.
Trace	A production trace can show the pattern's decision and effect.
Removal	The team knows what would break if the pattern were removed.

If a pattern fails `job`, `owner`, or `boundary`, remove it from the composition. If it fails `eval` or `trace`, keep it out of production until the proof exists.

Composition 1: Support Refund Investigation

Use this when the agent investigates a refund but must not directly issue money.

Responsibility	Pattern
Intake and routing	Routing and handoffs.
Evidence	Semantic recall and typed business tools.
Investigation	Bounded agent loop.
Tool execution	Tool use with narrow read tools and draft-only write tools.
Safety	Policy enforcement before refund actions.
Human control	Approval gate for high-value or exception refunds.
Quality	Evaluator checks evidence, policy, and recommendation.
Operations	Trace, replay, and incident-to-eval feedback.

The important boundary is financial authority. The agent may investigate, cite policy, and draft a refund request. It should not issue the refund. The refund action belongs to a policy-backed workflow, usually with approval.

```

async function handleRefundCase(input: RefundCase) {
  const route = routeSupportCase(input);
  const evidence = await collectRefundEvidence(input.orderId, route.region);

  const investigation = await refundAgent.investigate({
    caseId: input.caseId,
    evidenceRefs: evidence.refs,
    budget: { maxSteps: 6, maxToolCalls: 8, timeoutMs: 45000 }
  });
}

```

```
const policy = enforcePolicy({
  actorRole: input.agentRole,
  capability: 'refund',
  riskLevel: investigation.riskLevel
});

if (policy.decision === 'require_approval') {
  return approvals.request({
    proposedAction: investigation.proposedRefund,
    evidenceRefs: investigation.evidenceRefs,
    policyRefs: investigation.policyRefs
  });
}

if (policy.decision !== 'allow') return policy;
return refunds.draftRefundRequest(investigation);
}
```

This is agentic, but only in the investigation. The workflow owns routing, policy, approval, and side effects.

Composition 2: Knowledge-Bound Policy Assistant

Use this when the assistant answers from approved policy, documentation, or compliance sources.

Responsibility	Pattern
Source eligibility	Policy enforcement and knowledge-bound agent.
Evidence retrieval	Semantic recall and RAG.
Context control	Context budgets and working sets.
Answer shape	Structured output.
Unsupported questions	Refusal or human escalation.
Quality	Citation coverage and missing-evidence evals.
Operations	Source freshness monitoring and trace review.

The critical boundary is evidence. The assistant should not answer because the model “knows” something. It should answer because the system retrieved eligible sources and the answer cites them.

```
{
  "answer_status": "answered",
  "answer": "The request can be approved only if damage evidence is attached.",
  "citations": ["refund_policy.v3#damaged-items"],
```

```

"evidence_refs": ["src_refund_policy_2026_04"],
"missing_evidence": []
}

```

When evidence is missing, the correct output is not a weaker answer. It is `missing_evidence`, `conflicting_evidence`, `refused`, or `needs_human`.

Composition 3: Multi-Agent Research And Review

Use this when a task benefits from separated workstreams and independent review.

Responsibility	Pattern
Decomposition	Supervisor-worker.
Worker isolation	Scoped context and tool permissions.
Parallel work	Parallel agents when work is independent.
Review	Evaluator-optimizer or dedicated reviewer worker.
Merge	Supervisor merge policy.
Accountability	One final owner and stop reason.
Operations	Per-worker traces and merge-decision replay.

The critical boundary is final ownership. Multiple agents can produce evidence, drafts, or critiques. One component must own final synthesis.

```

type ResearchAssignment = {
  workerRole: 'source_finder' | 'technical_reviewer' | 'risk_reviewer';
  objective: string;
  scopedContextRefs: string[];
  allowedTools: string[];
  expectedOutputSchema: string;
  acceptanceCriteria: string[];
};

```

If every worker sees the same context and tools, you probably do not have a useful multi-agent system. You have duplicated model calls.

What Not To Compose

Some combinations are risky unless there is a strong boundary:

Risky Composition	Why It Fails	Control
Agent loop plus broad tools.	The loop can amplify a bad tool decision.	Narrow tools, policy, approval, stop rules.
RAG plus memory writes.	Retrieved errors become durable facts.	Memory write review and source metadata.
Evaluator plus no evidence.	The evaluator scores confidence without proof.	Citation and trajectory checks.
Multi-agent plus shared tool surface.	Every worker can cause the same damage.	Per-worker tool scopes.
Human approval plus vague request.	Humans approve without knowing the action.	Typed approval request and exact-action binding.
Policy only in prompts.	The model can ignore or reinterpret policy.	Runtime enforcement before execution.

Composition is not about more parts. It is about better boundaries.

Design Review

Before approving a composed system, ask:

1. What owns the goal?
2. What owns state?
3. What owns evidence?
4. What owns tool permissions?
5. What owns policy?
6. What owns approval?
7. What owns evaluation?
8. What owns final answer or action?
9. What owns replay and rollback?
10. What production failure becomes a new eval?

If the answer is “the prompt” for any high-risk responsibility, the architecture is not ready.

Design Rule

Compose patterns only when each added pattern has a job, a boundary, an owner, and an eval that can fail.

Related Chapters

- [Architecture Before Autonomy](#)

- From Patterns To Systems
- Pattern Evaluation Checklist
- Agents As Services
- Tool Use
- Agent Loop
- Policy Enforcement
- Human Approval Gates
- Production Evaluation Feedback Loops
- Vertical Slice Examples

Prompt Chaining and Gates

Prompt chaining breaks a task into known LLM steps and places validation gates between them. It is one of the safest ways to add model judgment without giving the model control over the whole workflow.

Use this pattern when the order of work is known but individual steps need language understanding, generation, classification, or extraction.

Intent

Run several bounded model calls in a fixed sequence. Each call receives a narrow input, produces a typed output, and passes through a deterministic gate before the next step runs.

The model performs local judgment. Code owns the workflow.

Use When

- The task has stable phases.
- Each phase can be tested independently.
- Intermediate outputs can be validated.
- You want lower risk than an autonomous agent loop.
- You need clear failure points and retry behavior.

Common examples:

- classify a request, extract fields, verify fields, then draft a response;
- summarize a document, extract claims, check claims, then produce a final answer;
- generate code, run tests, critique failures, then revise;
- retrieve evidence, normalize citations, then synthesize an answer.

Avoid When

- The steps are unknown before execution starts.
- The chain would branch into dozens of special cases.
- No gate can verify intermediate outputs.
- The same input often needs a different sequence of work.
- Latency from multiple calls is unacceptable.

If the chain keeps growing conditional branches, consider [Routing and Handoffs](#) or an [Agent Loop](#).

Architecture

Input

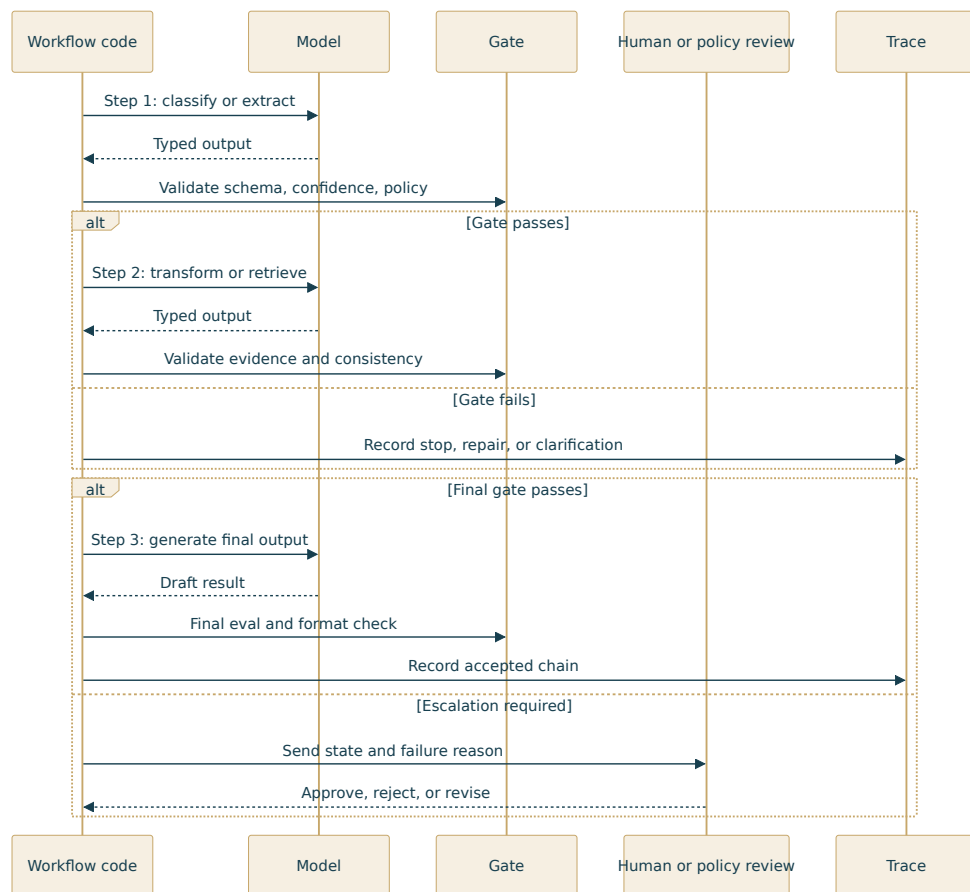
-> Step 1: classify or extract

```

-> Gate 1: schema, confidence, policy, completeness
-> Step 2: transform or retrieve
-> Gate 2: evidence, consistency, permissions
-> Step 3: generate final output
-> Gate 3: final eval, formatting, approval
-> Result

```

Gates should be deterministic whenever possible. A gate can use a model as an evaluator, but the gate still needs explicit criteria and a clear pass/fail output.



Use the diagram as a design test. Every arrow should map to a typed state transition, a named gate decision, and a bounded retry or escalation rule.

Gate Types

Gate	Checks	Failure Behavior
Schema gate	JSON shape, required fields, enums, numeric ranges	Ask model to repair or fail fast.
Evidence gate	Citations, source freshness, retrieval coverage	Retrieve more evidence or escalate.
Policy gate	permissions, user role, restricted actions	Block, redact, or request approval.

Gate	Checks	Failure Behavior
Confidence gate	classification confidence, ambiguity, missing inputs	Ask a clarification question.
Consistency gate	claims match previous state and evidence	Re-run the step with narrower context.
Cost gate	model call count, token budget, time budget	Return partial result or defer.
Human gate	subjective or high-impact decision	Pause until approval.

The strongest chains mix several gate types. A support workflow may use schema gates for extracted fields, policy gates for refunds, and human gates for exceptions.

Gate Contract

Every gate should produce a machine-readable decision. Do not hide gate behavior inside prose.

```
type GateDecision =
  | { status: 'pass' }
  | {
    status: 'repair';
    reason: string;
    maxRetries: number;
    repairInstructions: string;
  }
  | {
    status: 'stop';
    reason: string;
    userVisibleMessage?: string;
  }
  | {
    status: 'escalate';
    reason: string;
    requiredRole: 'support_lead' | 'security_reviewer' | 'domain_expert';
  };
```

The chain should record the gate decision, not just the final output. A production trace should show which step ran, which gate accepted or rejected it, and what happened next.

Use explicit failure reasons:

Failure Reason	Meaning	Next Action
<code>schema_invalid</code>	Output cannot be parsed or required fields are missing.	Repair once or twice, then stop.
<code>evidence_missing</code>	The step made a claim without enough source support.	Retrieve more evidence or escalate.
<code>policy_denied</code>	The next step would violate permission or business policy.	Stop or request approval.
<code>ambiguous_input</code>	The chain cannot choose a safe interpretation.	Ask the user or route to review.
<code>budget_exhausted</code>	Retry, token, latency, or tool budget is spent.	Return partial result or stop.

Worked Gate Example: Refund Draft

A refund support chain should stop before it drafts a customer promise if required evidence is missing. The model can classify, extract, and summarize, but gates decide whether the next step may run.

Chain Step	Model Output	Gate	Pass Condition	Failure Behavior
Classify request	<code>refund_request</code> with confidence <code>0.91</code>	confidence gate	Confidence $\geq$ <code>0.72</code> and type is not <code>unknown</code> .	Ask a clarifying question.
Extract fields	<code>order_id</code> , <code>customer_id</code> , <code>issue</code> , <code>requested_outcome</code>	schema gate	Required fields exist and IDs match expected format.	Repair once, then stop with <code>schema_invalid</code> .
Retrieve policy	source IDs and policy version	evidence gate	Current refund policy and order evidence are present.	Stop with <code>evidence_missing</code> or retrieve again.
Draft recommendation	proposed refund or denial	policy gate	Recommendation matches threshold, account status, and policy version.	Stop with <code>policy_denied</code> or request approval.
Draft customer reply	customer-facing text	final gate	Text matches approved recommendation and does not promise unapproved payment.	Block and send to support lead.

The important boundary is the fourth row. If the model proposes “full refund approved” but the policy gate returns `approval_required`, the chain must not continue to a customer-facing promise. It should pause with the exact evidence, proposed amount, policy version, and reviewer role.

```
{
  "step": "policy_gate",
  "status": "escalate",
  "reason": "approval_required",
  "required_role": "support_lead",
  "evidence_refs": ["order:0-104", "policy:refunds:v2026-06"],
  "blocked_next_step": "draft_customer_reply"
}
```

This is the difference between a chain and a loose sequence of prompts. The chain carries typed state and stops before unsafe language or side effects escape.

Implementation Notes

- Keep each prompt narrow. A step should have one job.
- Use structured output for every handoff between steps.
- Persist intermediate outputs, not just the final answer.
- Treat every model output as untrusted until a gate accepts it.
- Make repair loops bounded. A failed schema should not trigger infinite retries.
- Prefer deterministic gates for safety and model-based gates for subjective quality.
- Log each step with input hash, model, output, gate result, latency, and cost.

Example Chain

```
type TicketClass = 'billing' | 'technical' | 'account' | 'unknown';

interface Classification {
  type: TicketClass;
  confidence: number;
  reason: string;
}

interface ExtractedFields {
  customerId?: string;
  orderId?: string;
  issue: string;
}

interface ChainState {
```

```

input: string;
classification?: Classification;
fields?: ExtractedFields;
draft?: string;
}

function gateClassification(result: Classification): string | null {
  if (result.confidence < 0.72) return 'classification_low_confidence';
  if (result.type === 'unknown') return 'unknown_ticket_type';
  return null;
}

function gateFields(fields: ExtractedFields): string | null {
  if (!fields.issue.trim()) return 'missing_issue';
  return null;
}

async function runTicketChain(input: string): Promise<ChainState> {
  const state: ChainState = { input };

  state.classification = await classifyTicket(input);
  const classificationError = gateClassification(state.classification);
  if (classificationError) throw new Error(classificationError);

  state.fields = await extractTicketFields(input, state.classification.type);
  const fieldError = gateFields(state.fields);
  if (fieldError) throw new Error(fieldError);

  state.draft = await draftResponse(state.classification.type, state.fields);
  return state;
}

```

The example keeps control flow in code. The model classifies, extracts, and drafts, but it does not decide which validation rules apply.

Failure Modes

- A chain pretends to be deterministic while hidden prompts make important decisions.
- Gates validate shape but not meaning.
- The chain passes summarized state forward and loses key evidence.
- Repair loops keep asking the model to fix an output without changing the input or prompt.
- A model-based evaluator accepts fluent but unsupported claims.
- The chain becomes a fragile substitute for a real workflow engine.

Production Checklist

- Are all step inputs and outputs typed?
- Can each step be tested with fixture inputs?
- Does every gate have a named failure reason?
- Does every repair loop have a retry limit?
- Are intermediate outputs persisted for replay?
- Are high-risk actions separated from generation steps?
- Can operators see which gate stopped a run?

Related Chapters

- [Choosing the Right Pattern](#)
- [Structured Output](#)
- [Evaluator-Optimizer](#)
- [Human Approval Gates](#)
- [Durable Workflows](#)

Routing and Handoffs

Routing sends work to the right model, prompt, tool, workflow, or agent. Handoffs transfer responsibility with enough typed state for the next owner to act safely.

This pattern is often the missing middle between a single prompt and a multi-agent system.

Intent

Use a classifier, policy rule, planner, or deterministic condition to choose the next execution path. Then pass a narrow state object to the selected path.

A router should decide where work belongs. It should not complete the work itself.

Use When

- Inputs belong to distinct domains or intents.
- Different paths need different tools, permissions, models, or prompts.
- Simple requests should use cheap, fast paths.
- Risky requests need stricter policy or human approval.
- A large agent is struggling because it has too many tools or responsibilities.

Common examples:

- route support tickets to billing, technical, account, or fraud workflows;
- route easy requests to a small model and unusual requests to a stronger model;
- route code tasks to search, edit, review, or test agents;
- route requests with side effects through approval workflows;
- route RAG questions to the correct index, tenant, or domain corpus.

Avoid When

- There is only one meaningful path.
- The router cannot explain or score its decision.
- The downstream paths have overlapping ownership.
- A wrong route would create dangerous side effects.
- The router depends on hidden conversation context that is not persisted.

If the route cannot be chosen with confidence, ask for clarification or send the task to a safe fallback path.

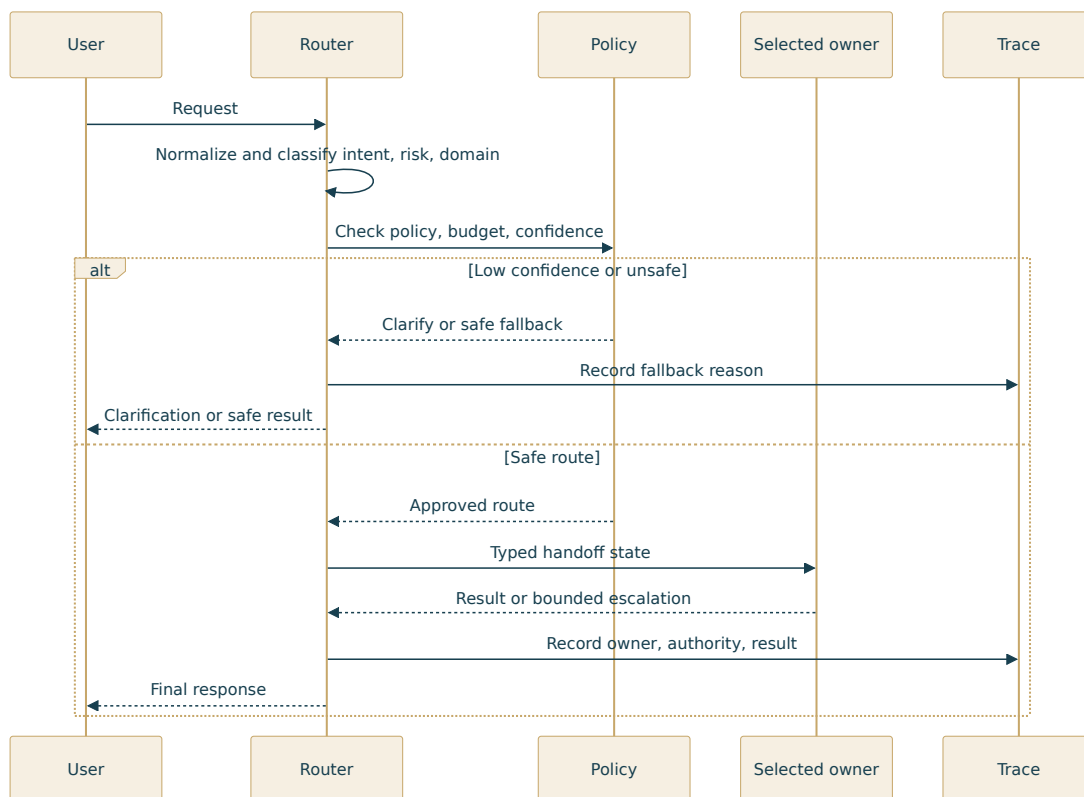
Architecture

Request

-> Normalize input

- > Classify intent, risk, domain, and required capability
- > Apply policy and budget rules
- > Select path
- > Handoff typed state
- > Downstream path completes or escalates

The handoff should include only the state the downstream path needs. Avoid dumping the full conversation into every specialist.



Use this diagram to test the design. A route is not ready if it cannot name the owner, authority, handoff state, confidence threshold, fallback, and trace fields.

Router Outputs

A router output should be structured:

```

type Route =
  | 'answer_from_docs'
  | 'billing_workflow'
  | 'technical_triage'
  | 'human_review'
  | 'reject'
  | 'clarify';

interface RoutingDecision {
  
```

```

route: Route;
confidence: number;
reason: string;
requiredCapabilities: string[];
risk: 'low' | 'medium' | 'high';
}

```

Good router decisions are inspectable. The system should log the route, confidence, reason, model version, policy checks, and fallback behavior.

Handoff Contract

A handoff is not just a message. It is a change in responsibility.

Include:

- normalized user intent;
- route decision and reason;
- relevant extracted fields;
- evidence references;
- permission scope;
- budget remaining;
- deadline or timeout;
- previous failed routes or tool failures;
- expected output contract;
- escalation path.

Exclude:

- unrelated conversation history;
- tools the recipient cannot use;
- hidden policy text that should remain system-owned;
- raw sensitive data when a reference or redacted field is enough.

Route Decision Record

Record the routing decision as data. This makes routing reviewable instead of anecdotal.

```

{
  "route_id": "route_2026_06_21_001",
  "input_ref": "ticket_4921",
  "selected_route": "billing_workflow",
  "confidence": 0.84,
  "reason": "Customer asks about duplicate charge on order receipt.",
  "risk": "medium",
}

```



```

"policy_checks": [
  { "name": "tenant_access", "decision": "allow" },
  { "name": "payment_write", "decision": "deny_without_approval" }
],
"handoff": {
  "owner": "billing_workflow",
  "allowed_tools": ["orders.read", "payments.read", "refunds.draft_request"],
  "forbidden_tools": ["refunds.issue_refund"],
  "state_refs": ["order:ord_123", "payment:pay_456"],
  "expected_output": "billing_resolution_draft"
},
"fallback": "human_review",
"max_handoffs": 2
}

```

The record should show three things: why the route was chosen, what authority moved with the handoff, and where the work goes if the route fails.

Router Types

Router Type	Best For	Watch Out For
Rule router	Compliance, tenant, role, known metadata	Rule drift as products change.
Model router	Natural language intent and ambiguous requests	Low confidence and prompt injection.
Embedding router	Domain corpus or knowledge base selection	Similar but unsafe corpora.
Cost router	Model tier selection	Cheap model overuse on hard tasks.
Risk router	Approval, sandbox, or policy paths	Missing high-risk edge cases.
Capability router	Tool or agent selection	Overlapping tool descriptions.

Production systems usually combine them. For example, code may enforce tenant and policy rules before a model chooses between specialist workflows.

Implementation Notes

- Keep route labels stable. Treat route names as API contracts.
- Route before loading large context.
- Separate intent routing from policy routing.
- Use confidence thresholds with explicit clarify or fallback paths.

- Validate handoff state before the recipient starts.
- Do not let downstream agents silently reroute forever.
- Trace route decisions across the whole run.
- Test routers with adversarial and ambiguous examples, not only happy paths.

Failure Modes

- A router becomes a hidden general-purpose agent.
- Route labels overlap, so the same request can fit several paths.
- Handoffs lose authority boundaries and give specialists too many tools.
- Low-confidence routes proceed instead of asking for clarification.
- Multi-agent handoffs create loops with no owner.
- Cost routing sends hard tasks to weak models and hides quality loss.

Evaluation Strategy

Evaluate the route decision and the handoff contract separately. A correct route with missing authority, evidence, or budget is still a failed handoff.

- Build a labeled route set with clear, ambiguous, out-of-domain, adversarial, and high-risk requests.
- Test every route label plus the `clarify`, `reject`, and safe fallback paths.
- Test requests that contain language associated with one route but require another route by policy or tenant.
- Assert that each handoff includes its required state and excludes unrelated context and excess authority.
- Test downstream failure, rerouting, and loop prevention.
- Compare model routing with a deterministic or single-path baseline before adding more routes.

A compact fixture should make the route and handoff expectations visible:

```
type RoutingEvalCase = {
  caseId: string;
  input: string;
  expected: {
    route: Route;
    allowedAlternatives?: Route[];
    maxAuthority: "read" | "draft" | "write_after_approval";
    requiredHandoffFields: string[];
    forbiddenHandoffFields: string[];
    maxHandoffs: number;
  };
};
```

Measure route accuracy by class, unsafe misroute rate, clarification precision, fallback rate, handoff contract validity, authority expansion, loop rate, downstream completion rate, latency, and cost. Report a confusion matrix instead of relying only on overall accuracy. A router that performs well on common support requests can still fail every fraud or security case.

For the shared eval case contract and release-gate method, see [Evaluation-Driven Agent Development](#).

Production Checklist

- Are route labels mutually understandable and stable?
- Does every route have an owner and output contract?
- Does the router have a fallback for low confidence?
- Does policy run before dangerous handoffs?
- Can an operator explain why a request took a path?
- Are route datasets part of regression tests?
- Are route loops impossible or bounded?

Related Chapters

- [Choosing the Right Pattern](#)
- [MCP-first Tool Use](#)
- [A2A Agent Interoperability](#)
- [Supervisor / Worker](#)
- [Policy Enforcement](#)
- [Context Engineering](#)
- [Evaluation-Driven Agent Development](#)

Resource-Aware Agent Design

Agentic systems spend resources with every model call, retrieval, tool call, handoff, memory operation, and retry. Resource-aware design makes cost, latency, context, and compute part of the architecture.

Use this pattern when an agent must run repeatedly, serve users interactively, or operate under budget constraints.

Intent

Optimize for completed tasks, not raw model intelligence.

A resource-aware agent chooses the cheapest safe path that can complete the job with acceptable quality.

Resource Types

Resource	What To Track
Tokens	prompt tokens, output tokens, hidden context, compression cost.
Model calls	count, model tier, retries, parallel calls.
Latency	end-to-end time, step time, tool wait time, human wait time.
Money	cost per run, cost per completed task, cost per route.
Context window	relevant context, stale context, duplicated context.
Tool quota	API rate limits, browser sessions, database load.
Human attention	approvals, corrections, escalations, review time.
Operational risk	side effects, policy checks, incident cost.

Human attention is a resource. A cheap model path that creates constant review work is not cheap.

Budget Objects

Give each run a budget object:

```
interface AgentBudget {  
  maxCostUsd: number;  
  maxModelCalls: number;  
  maxToolCalls: number;  
  maxIterations: number;  
  maxWallClockMs: number;  
}
```

```
maxContextTokens: number;  
approvalRequiredAboveUsd?: number;  
}
```

Budgets should travel with state so routers, tools, evaluators, and fallback logic can make consistent decisions.

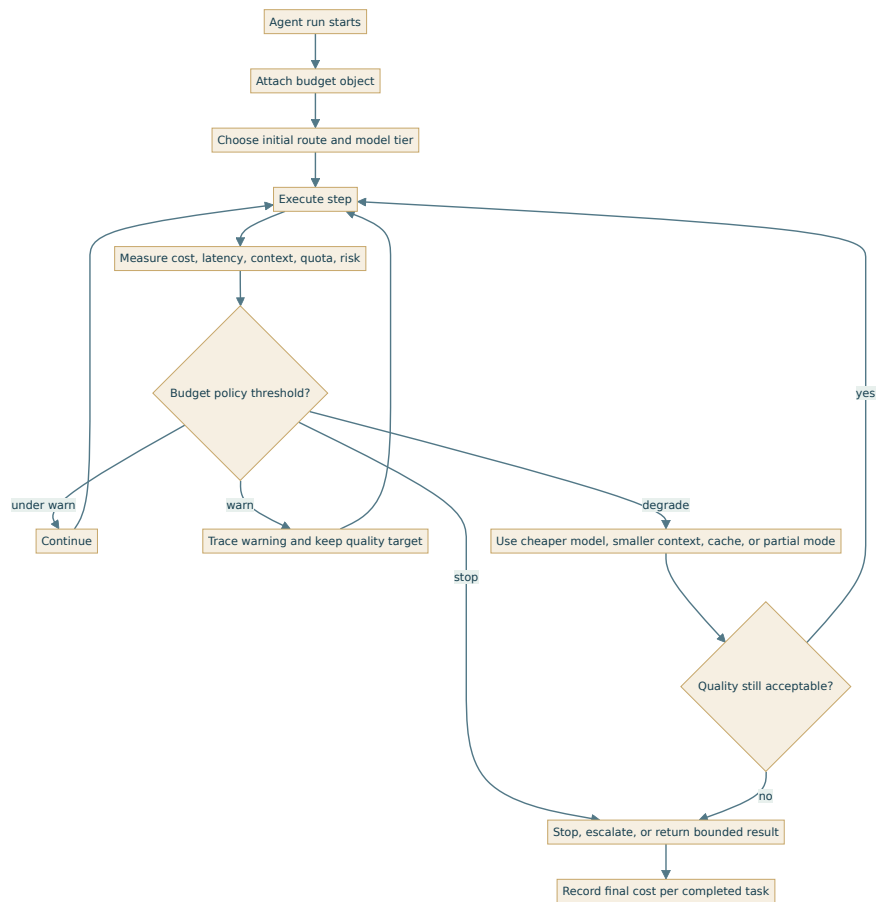
Budget Policy

A budget is useful only if the system knows what to do as it is spent.

```
type BudgetAction =  
  | 'continue'  
  | 'switch_to_cheaper_model'  
  | 'compress_context'  
  | 'skip_optional_review'  
  | 'return_partial_result'  
  | 'escalate'  
  | 'stop';  
  
type BudgetPolicy = {  
  warnAtPct: number;  
  degradeAtPct: number;  
  stopAtPct: number;  
  preferredDegradedAction: BudgetAction;  
};
```

For example, an interactive support assistant might warn internally at 60 percent of budget, switch to cheaper classification and smaller context at 75 percent, and stop with a partial answer or human escalation at 95 percent. A background research agent might tolerate higher latency but stop earlier on tool quota or source freshness risk.

The budget policy should be part of the route or workflow contract, not an afterthought in logging.



Use this loop to review implementations. Resource control is not only a meter; it must change execution before cost, latency, context, quota, or risk breaks the run.

Optimization Levers

Lever	Use When	Trade-off
Model routing	Some tasks are easy and some are hard.	Route mistakes can reduce quality.
Prompt chaining	Steps are known and can be validated.	More calls, but less context per call.
Context minimization	Context is large or noisy.	May omit useful evidence.
Retrieval filtering	Sources are numerous or mixed quality.	Filters can miss edge cases.
Parallelization	Independent subtasks dominate latency.	More total calls and harder aggregation.
Caching	Queries or tool outputs repeat.	Cache invalidation and freshness risks.
Early exit	Confidence is high enough.	Needs calibrated thresholds.

Lever	Use When	Trade-off
Human escalation	Automation would be risky or expensive.	Uses scarce human attention.

Optimize only after you can trace costs by route and step.

Context Compression

Compress context when it is too large, repeated, or stale.

Prefer:

- structured state over long chat history;
- file or record references over pasted content;
- retrieved snippets over entire documents;
- task-specific memory over global memory;
- summaries with source pointers;
- explicit deletion of irrelevant context.

Do not compress away evidence, constraints, tool errors, approvals, or unresolved questions.

Cost-Aware Routing

Cost-aware routing sends work to the cheapest safe path:

- deterministic code for known operations;
- small model for classification and extraction;
- stronger model for planning or synthesis;
- specialized agent for domain work;
- human review for high-risk ambiguity.

The router needs evals. Otherwise cost savings can silently become quality loss.

Latency Design

Interactive agents need visible progress and bounded wait.

Use:

- streaming status at workflow boundaries;
- parallel retrieval where safe;
- background jobs for long work;
- partial results with clear limits;
- cancellation;
- resumable state.

For long-running tasks, optimize for resumability before raw speed.

Degraded Modes

Resource-aware systems should fail smaller before they fail completely.

Constraint Hit	Degraded Mode	Do Not Do
Token budget	Use references, summaries with citations, or smaller working set.	Drop constraints, approvals, or error history.
Model cost	Route simple steps to cheaper models.	Route high-risk synthesis to a weak model without eval proof.
Latency	Return partial result, background the rest, or ask to continue.	Hide long-running work behind a frozen UI.
Tool quota	Use cached read-only data with freshness labels.	Call write tools without current evidence.
Human attention	Batch low-risk reviews or raise confidence threshold.	Create vague approvals that humans rubber-stamp.
Risk budget	Require approval, sandbox, or stop.	Let the model decide that risk is acceptable.

Degraded behavior should be visible in traces and user-facing output when it affects quality. A cheaper or partial path is acceptable; pretending it is the same path is not.

Failure Modes

- Optimizing token count while increasing human correction work.
- Using a cheap model for routes it cannot classify reliably.
- Parallelizing dependent work and creating inconsistent results.
- Compressing context so aggressively that the agent forgets constraints.
- Caching stale evidence in regulated or fast-changing domains.
- Measuring cost per call instead of cost per completed task.

Related Chapters

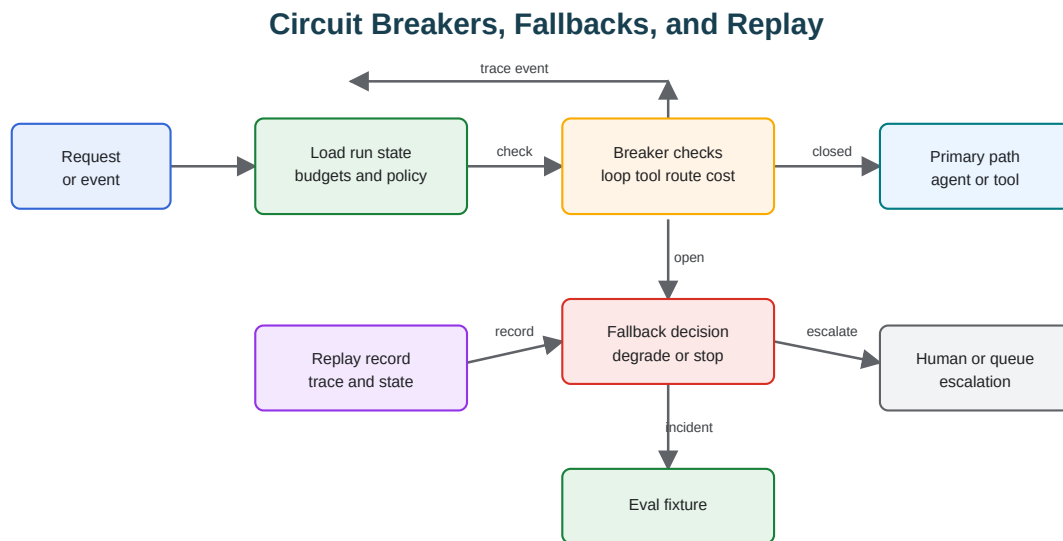
- [Choosing the Right Pattern](#)
- [Routing and Handoffs](#)
- [Context Engineering](#)
- [Working Memory](#)
- [Cost Controls and Runtime Budgets](#)
- [Observability and Evals](#)

- [Circuit Breakers, Fallbacks, and Replay](#)

Circuit Breakers, Fallbacks, and Replay

Agentic systems need controls that stop waste, contain damage, and make failures explainable. Circuit breakers stop unsafe or unproductive execution. Fallbacks give the system a safer next move. Replay lets engineers reconstruct what happened.

This is a reliability pattern for agent loops, tool use, RAG, workflow orchestration, and multi-agent systems.



Intent

Protect the system from repeated failure and make every run recoverable enough to debug.

An agent should not keep calling the same failing tool, searching the same empty corpus, revising the same bad draft, or handing the same task between agents without progress.

The key idea is simple: autonomy needs runtime brakes. A breaker is not an error handler buried inside a tool wrapper. It is an architectural control that decides when the system must stop, degrade, ask for help, or preserve enough state for replay.

Use When

- Agents can call tools, APIs, browsers, shells, or workflows.
- Work can loop, retry, delegate, or wait.
- Costs can grow with each model or tool call.
- Partial state matters after failure.
- Operators need to explain why a run stopped.

This pattern is mandatory for production systems with side effects.

Avoid When

- The task is a throwaway prototype with no external effect.
- The system has no loop, retry, or tool use.
- Failures do not need investigation.

Even then, keep a simple run log. The prototype that works often becomes the production seed.

Circuit Breakers

A circuit breaker turns repeated or high-risk failure into a stop, fallback, or escalation.

Breaker	Trigger	Action
Tool failure breaker	Same tool fails N times or returns malformed output.	Disable tool for run and choose fallback.
No-progress breaker	State does not change across iterations.	Stop loop and return blocked status.
Cost breaker	Token, model-call, or money budget is reached.	Stop, summarize progress, and ask for approval.
Latency breaker	Step or run exceeds time budget.	Defer, enqueue, or return partial result.
Retrieval breaker	Search returns low-coverage or conflicting evidence.	Ask clarification or escalate.
Policy breaker	Tool intent violates permissions or risk rules.	Block action and log policy reason.
Handoff breaker	Agents bounce a task between roles.	Assign owner or escalate to human.
Output breaker	Final answer fails schema, citation, or eval checks.	Repair once, then fail safely.

Breakers need names and thresholds. “The agent got stuck” is not operational enough. “No-progress breaker fired after 4 iterations with unchanged evidence set” is debuggable.

Place breakers at the boundary where damage can grow:

Boundary	What The Breaker Protects
Loop boundary	Prevents unbounded reasoning, repeated planning, and no-progress iteration.
Tool boundary	Prevents repeated external failures, malformed arguments, duplicate side effects, and unsafe writes.
Model boundary	Prevents runaway cost, latency spikes, bad model routes, and degraded structured-output behavior.

Boundary	What The Breaker Protects
Retrieval boundary	Prevents answers from weak, stale, contradictory, or unauthorized evidence.
Memory boundary	Prevents private, stale, or low-confidence memory from steering future runs.
Policy boundary	Prevents the agent from converting intent into forbidden action.
Multi-agent boundary	Prevents delegation loops, unclear ownership, and disagreement without resolution.

The breaker should emit a trace event with the breaker name, trigger, threshold, observed value, action taken, and user-visible stop reason. If the breaker fires but the trace does not explain why, operators will still be debugging a mystery.

Fallbacks

A fallback should be safer than the failed path.

Useful fallback types:

- ask the user for missing information;
- return a partial answer with explicit limits;
- switch from autonomous tool use to deterministic workflow;
- switch from a weak model to a stronger model;
- switch from write action to read-only analysis;
- use cached or previously verified data;
- route to human review;
- schedule background processing;
- fail closed for restricted actions.

Do not use fallback as a way to hide failure. The user or operator should know what changed.

A good fallback has a contract:

- **Scope:** what the fallback is allowed to do.
- **Quality limit:** what the fallback cannot guarantee.
- **State rule:** whether the run can continue, must pause, or must stop.
- **User message:** how the limitation is explained without exposing internal noise.
- **Operator trace:** which breaker caused the fallback and what evidence was preserved.

For agentic systems, the safest fallback often reduces autonomy. Move from write to draft, from autonomous action to approval, from broad tool use to a deterministic workflow, or from final answer to clarification.

Replay

Replay is the ability to reconstruct a run from durable state.

Store:

- run ID and parent run ID;
- goal and normalized intent;
- model calls, model versions, parameters, and prompt references;
- tool calls, inputs, outputs, errors, and side-effect identifiers;
- retrieval queries, source IDs, and citation metadata;
- state transitions;
- route decisions and handoffs;
- policy checks;
- breaker and fallback events;
- final output and eval results.

Redact sensitive content when required, but keep enough references to investigate the run.

Replay must be designed before the incident. After a failure, it is too late to discover that prompts, tool arguments, policy decisions, and state transitions were scattered across unrelated logs.

Replay also needs safety rules. A replay should not accidentally send another customer email, issue another refund, or write another memory record. Separate read-only trace replay, deterministic replay with mocked outputs, and sandbox replay with fake side effects.

Replayable Actions

When an agent performs expensive or risky work, make the action replayable:

- record the input, selected tool, policy decision, and result;
- attach an idempotency key to external side effects;
- cache safe read-only results when freshness rules allow it;
- preserve enough context to reproduce a failed run;
- make rollback explicit for operations that change state.

Replay turns a failure from a mystery into a test case.

Replay Packet

When a breaker fires, store a replay packet with the smallest useful set of records needed to reconstruct the failure.

```
{
  "replay_id": "replay_support_refund_2026_06_21_001",
  "run_id": "run_9f32",
  "trigger": {
    "breaker": "tool:shipping.read_delivery_status",
    "reason": "tool_failure_breaker_open",
    "observed": 3,
```

```

    "threshold": 3
  },
  "goal": "Investigate whether order ord_123 qualifies for refund.",
  "state_snapshot_ref": "state/run_9f32/step_4",
  "context_packet_ref": "context/run_9f32/step_4",
  "model_events": ["model_call_1", "model_call_2"],
  "tool_events": ["tool_call_1", "tool_call_2", "tool_call_3"],
  "policy_events": ["policy_check_1"],
  "side_effects": [],
  "fallback_taken": "needs_human",
  "safe_replay_mode": "mocked_tools"
}

```

The packet should answer four questions: what fired, what the system believed, what happened before the breaker, and how to replay safely without repeating side effects.

Replay Levels

Level	Capability	Use
Trace replay	Reconstruct what happened from logs.	Incident review and debugging.
Deterministic replay	Re-run deterministic code with recorded model/tool outputs.	Regression tests and workflow validation.
Full replay	Re-run model and tool calls in a sandbox.	Reproduction when external systems are stable.
Counterfactual replay	Re-run with changed prompt, policy, model, or tool.	Evaluate fixes before rollout.

Most teams should start with trace replay. Full replay is useful but harder because models, tools, and external data change.

Counterfactual replay is especially useful for agentic systems. It lets the team ask: would the new prompt, model route, policy rule, retrieval index, or tool schema have avoided the failure without breaking known-good cases?

Implementation Notes

- Add breakers at the loop, tool, retrieval, route, and workflow levels.
- Record breaker events as first-class trace events.
- Tie fallback decisions to explicit failure reasons.
- Separate safe retries from repeated guessing.
- Make every side effect idempotent or traceable.
- Store enough state to resume or fail cleanly.

- Turn incidents into eval cases.
- Prefer typed breaker events over free-form log messages.
- Keep breaker thresholds configurable by environment, risk class, and tool capability.
- Treat a breaker firing as an evaluation signal, not only an operational event.

Example Breaker Logic

```
type BreakerAction = 'continue' | 'fallback' | 'escalate' | 'stop';

type ToolFailure = {
  toolName: string;
  count: number;
  lastError: string;
};

type RunBudget = {
  maxIterations: number;
  maxToolFailuresPerTool: number;
  maxCostUsd: number;
};

type BreakerDecision = {
  action: BreakerAction;
  reason: string;
  traceEvent: {
    type: 'breaker';
    name: string;
    observed: number;
    threshold: number;
  };
};

function evaluateToolBreaker(
  failure: ToolFailure,
  budget: RunBudget
): BreakerDecision {
  if (failure.count < budget.maxToolFailuresPerTool) {
    return {
      action: 'continue',
      reason: 'tool_failure_budget_remaining',
      traceEvent: {
        type: 'breaker',
        name: `tool:${failure.toolName}`,
        observed: failure.count,
```

```

        threshold: budget.maxToolFailuresPerTool
    }
};
}

return {
    action: 'fallback',
    reason: 'tool_failure_breaker_open',
    traceEvent: {
        type: 'breaker',
        name: `tool:${failure.toolName}`,
        observed: failure.count,
        threshold: budget.maxToolFailuresPerTool
    }
};
}

function evaluateLoopBreaker(
    iteration: number,
    costUsd: number,
    budget: RunBudget
): BreakerDecision | null {
    if (iteration >= budget.maxIterations) {
        return {
            action: 'stop',
            reason: 'max_iterations_reached',
            traceEvent: {
                type: 'breaker',
                name: 'loop:max_iterations',
                observed: iteration,
                threshold: budget.maxIterations
            }
        };
    }

    if (costUsd >= budget.maxCostUsd) {
        return {
            action: 'escalate',
            reason: 'max_cost_reached',
            traceEvent: {
                type: 'breaker',
                name: 'run:max_cost_usd',
                observed: costUsd,
                threshold: budget.maxCostUsd
            }
        };
    }
}

```



```

    }
    };
}

return null;
}

```

The important part is not the code size. The important part is that the stop reason becomes part of the run state and trace.

The action should drive behavior. `continue` proceeds on the primary path. `fallback` moves to a safer path. `escalate` asks a human or queue to take over. `stop` ends the run with a clear state and replay record.

Failure Modes

- Breakers exist in logs but do not affect execution.
- Fallbacks silently lower quality without telling the user.
- Retries repeat the same inputs and expect different outcomes.
- Traces omit tool inputs, making replay impossible.
- Side effects cannot be matched to the agent run that created them.
- Multi-agent systems have no single owner when a breaker fires.
- Breaker thresholds are hard-coded and cannot change by risk class.
- Replay can reproduce the failure only by touching live systems again.
- The system falls back to a more powerful model when the real issue is missing policy or state.

Evaluation Guidance

Test breakers and fallbacks directly. Do not wait for production incidents to prove that the runtime can stop safely.

Eval Case	Expected Behavior
Tool fails repeatedly	Tool breaker opens, fallback path runs, trace records the threshold.
Loop makes no state progress	Loop stops with a no-progress reason and no extra tool calls.
Cost budget is exhausted	Run pauses, summarizes progress, and asks for approval or continuation budget.
Retrieval evidence is weak	Agent asks for clarification or refuses unsupported claims.
Policy blocks a write action	Action is denied or escalated before side effects.
Multi-agent delegation repeats	Supervisor assigns a single owner or escalates.

Eval Case	Expected Behavior
Replay uses recorded trace	Replay does not trigger live side effects.

Production Checklist

- Does every loop have max iterations, max cost, and max wall-clock time?
- Does every tool have timeout, retry, and failure thresholds?
- Does every fallback tell the operator what changed?
- Can the system stop safely after partial progress?
- Can a failed run become an eval fixture?
- Can side effects be audited and reversed when possible?
- Can operators replay a run without reading raw prompts from scattered logs?
- Are breaker events attached to trace IDs, eval cases, and incident reviews?
- Are fallback paths tested before launch?
- Can risky tools be disabled without redeploying the whole agent?

Related Chapters

- [Agent Loop](#)
- [Goals and State](#)
- [Self-Healing Workflows](#)
- [Durable Workflows](#)
- [Cost Controls and Runtime Budgets](#)
- [Observability and Evals](#)
- [Policy Enforcement](#)

Source Map

This page maps external references to this book’s chapters. Use it as a reading guide, not as a replacement for the book’s pattern language.

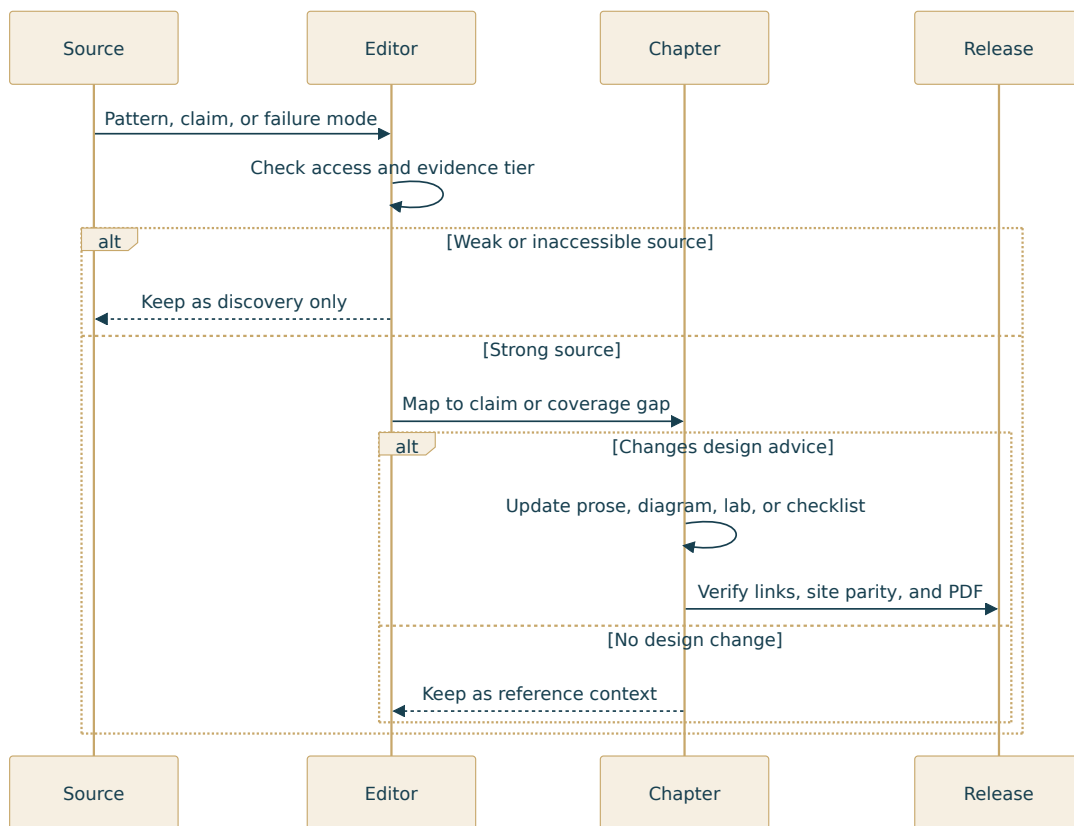
The sources repeat several core ideas: start simple, separate workflows from agents, use tools through typed contracts, add memory deliberately, evaluate behavior, and reserve multi-agent systems for cases where specialization justifies orchestration cost.

How To Use This Map

Use this page when you need to answer one of three questions:

Question	Use This Section	Output
Where did this book’s pattern language come from?	Primary References and Local PDF References Reviewed.	A short source trail for a chapter, decision, or term.
Does the book cover a pattern I saw elsewhere?	Pattern Coverage Map and Local Pattern Repository Coverage.	A chapter link, or a candidate gap for future expansion.
Which sources should I trust first?	Evidence Tiers.	A priority order for reading, citing, and revising.

For normal reading, do not start here. Start with [How To Read This Book](#), then return to this map when you want source context or coverage checks.



Read the map as an editorial filter. A source earns space in the book when it changes a reader's design decision, exposes a missing failure mode, or gives stronger evidence for a chapter claim.

Source-Backed Decision Shortcuts

Use these shortcuts when a source names a pattern but the engineering decision is still unclear.

Decision You Need To Make	Start With	Use The Sources To Check
Is this a workflow, an agent, or a multi-agent system?	Choosing the Right Pattern	Whether the source separates deterministic control from model-chosen steps.
Should we add routing, handoffs, or specialist agents?	Routing and Handoffs and Choosing Multi-Agent Topology	Whether separate context, tools, permissions, or latency justify coordination cost.
Does this need memory, retrieval, or a knowledge boundary?	Context Engineering , Working Memory , and Agentic RAG Systems	Whether the source distinguishes transient context, durable memory, and cited evidence.
Are tools safe enough for model-mediated use?	Tool Capability Design , MCP-first Tool Use , and Human Approval Gates	Whether tool authority, schema, errors, policy, and approval are explicit.

Decision You Need To Make	Start With	Use The Sources To Check
Can the pattern survive production failure?	Circuit Breakers, Fallbacks, and Replay , Observability and Evals , and Production Evaluation Feedback Loops	Whether traces, replay, evals, rollback, and incident feedback are part of the design.
Is a source adding a useful pattern or just a new name?	Pattern Composition Playbook	Whether the pattern changes ownership, state, tools, risk, evals, or operations.

Evidence Tiers

Not every source has the same editorial weight. Use this order when sources disagree:

Tier	Source Type	Editorial Use
1	Vendor engineering guides, framework docs, and primary implementation docs.	Use for current terminology, operational guidance, and framework-specific behavior.
2	Technical books and long-form practitioner material.	Use for durable concepts, teaching order, and examples that need deeper explanation.
3	Curated catalogs, pattern lists, and diagrams.	Use for coverage discovery and cross-linking, not as final authority.
4	Videos, gated articles, and lightly reviewed posts.	Use as discovery material only until claims are checked against stronger sources.

This prevents the online book from drifting toward whatever source has the longest pattern list. The book should privilege engineering evidence over novelty.

Primary References

Source	Useful Ideas	Book Mapping
Anthropic: Building Effective Agents	Workflows vs agents, prompt chaining, routing, parallelization, orchestrator-workers, evaluator-optimizer, autonomous agents.	Choosing the Right Pattern , Prompt Chaining and Gates , Routing and Handoffs , Evaluator-Optimizer , Agent Loop .
Google Cloud: Choose a design pattern for your agentic AI system	Requirements-first selection, single-agent and multi-agent patterns, sequential, parallel, iterative refinement, human-in-the-loop, custom logic.	Choosing the Right Pattern , Parallel Agents , Human Approval Gates , Agentic System Architecture .

Source	Useful Ideas	Book Mapping
Databricks: Agent system design patterns	Complexity continuum from prompt to deterministic chain to single-agent and multi-agent systems, plus production guidance for testing, tracing, failure handling, model updates, and cost.	Choosing the Right Pattern , Circuit Breakers , Fallbacks , and Replay , Observability and Evals .
LangChain: Choosing the Right Multi-Agent Architecture	Multi-agent selection across subagents, skills, handoffs, and routers, with explicit tradeoffs around context isolation, state, parallelism, and model-call overhead.	Choosing the Right Pattern , Routing and Handoffs , Skills , Supervisor / Worker , Parallel Agents .
LangChain: Deep Agents overview and Frameworks, runtimes, and harnesses	Agent harness framing: planning, files, context management, subagents, long-term memory, human approval, permissions, sandboxes, durable execution, and framework/runtime/harness distinctions.	Agent Harnesses , Context Budgets and Working Sets , Agent Engineer Toolkit , Framework Selection , Skills , Durable Workflows .
MongoDB: 7 Practical Design Patterns for Agentic Systems	Controlled flows, LLM routing, parallelization, reflection, human-in-the-loop, agents, and multi-agent architectures.	Prompt Chaining and Gates , Routing and Handoffs , Reflection , Supervisor / Worker .
Agentic Patterns Graph	Large pattern catalog with categories for context, orchestration, reliability, security, tool use, learning, and UX.	Circuit Breakers , Fallbacks , and Replay , Context Engineering , Working Memory , Policy Enforcement .
GitHub: awesome-agentic-patterns	Curated repository of production and emerging patterns, useful as a discovery index.	This source informs the extended catalog and future additions, especially reliability, context, coding-agent, security, and tool-use patterns.

Secondary References

Source	Useful Ideas	How To Use
ByteByteGo: Top AI Agentic Workflow Patterns	Accessible explanations of reflection, tool use, ReAct, planning, and multi-agent patterns.	Good introductory reading for readers who want a lighter explanation before the deeper chapters.

Source	Useful Ideas	How To Use
Phil Schmid: Zero to One - Learning Agentic Patterns	Practical examples for routing, parallelization, tool use, orchestrator-workers, and multi-agent systems.	Useful for implementation intuition and example shapes.
ProjectPro: Agentic AI Design Patterns	Planning, tool use, reflection, multi-agent examples, and a practical MCP discussion.	Useful for explaining common enterprise examples.
Tungsten Automation: Tool-Use Pattern	Enterprise framing for tool use, ground truth, and workflow APIs.	Maps to MCP-first Tool Use and Policy Enforcement .
Towards AI: 5 Design Patterns in Agentic AI Workflow	Introductory framing for prompt chaining and workflow decomposition.	Useful as a light overview; the article is partially gated.
Medium: Multi-Agent System Patterns	Multi-agent architecture dimensions: control, execution, coordination, and interaction. Useful distinction between roles and patterns.	Good support for a future bridge chapter on composing multi-agent systems without turning the book into a flat catalog.
Medium: Agentic Design Patterns	Common pattern explanations for reflection, tool use, planning, and multi-agent design.	Duplicates core concepts already covered here.
Medium: Agentic Patterns - Architectures for Coordinated AI Systems	Hierarchical, peer-to-peer, market-based, and swarm coordination.	Useful for future expansion of multi-agent coordination patterns.
YouTube: AI agent design patterns	Video walkthrough of agent design patterns.	Use as companion media, not as a primary textual source unless transcript review is added.
YouTube: Master ALL 20 Agentic AI Design Patterns	Broad video catalog of pattern names and examples.	Use as discovery material; validate individual claims against primary sources.

Sources that were not publicly accessible or only served as discovery routes were excluded from the map.

Local Pattern Repository Coverage

A local flattened digest of the [awesome-agentic-patterns](#) project was used as a coverage checklist. Its patterns were not copied into this book. They helped identify missing or underdeveloped areas that now have authored treatment here.

The 2026-06-20 intake scan extracted 167 pattern records across orchestration and control, tool use and environment, reliability and evaluation, context and memory, UX and collaboration, security and safety, feedback loops, and learning and adaptation. The online book should use this catalog as an expansion queue for chapter quality and cross-links, not as a replacement taxonomy.

Local Pattern Area	Representative Patterns Reviewed	Book Mapping
Agent threat model and tool security	<code>lethal-trifecta-threat-model</code> , <code>policy-gated-tool-proxy</code> , <code>sandboxed-tool-authorization</code> , <code>egress-lockdown-no-exfiltration-channel</code> , <code>pii-tokenization</code> .	Agent Threat Model, Agent Security and Sandboxing, Policy Enforcement, Tool Capability Design.
Tool capability and agent-friendly interfaces	<code>cli-first-skill-design</code> , <code>tool-capability-compartmentalization</code> , <code>llm-friendly-api-design</code> , <code>agent-first-tooling-and-logging</code> , <code>static-service-manifest-for-agents</code> , <code>code-first-tool-interface-pattern</code> .	Skills, Tool Capability Design, MCP-first Tool Use, Agent Harnesses.
Context operations	<code>curated-file-context-window</code> , <code>curated-code-context-window</code> , <code>context-window-auto-compaction</code> , <code>context-minimization-pattern</code> , <code>context-window-anxiety-management</code> .	Context Budgets and Working Sets, Context Engineering, Agent Harnesses.
Production eval feedback	<code>incident-to-eval-synthesis</code> , <code>workflow-evals-with-mocked-tools</code> , <code>canary-rollout-and-automatic-rollback-for-agent-policy-changes</code> , <code>anti-reward-hacking-grader-design</code> , <code>background-agent-ci</code> .	Production Evaluation Feedback Loops, Evaluation-Driven Agent Development, Observability and Evals, Coding Agents.
Coding-agent runtime	<code>coding-agent-ci-feedback-loop</code> , <code>background-agent-ci</code> , <code>asynchronous-coding-agent-pipeline</code> , <code>shell-command-contextualization</code> , <code>custom-sandboxed-background-agent</code> .	Coding Agents, Agent Harnesses, Production Evaluation Feedback Loops.
Multi-agent topology and coordination	<code>declarative-multi-agent-topology-definition</code> , <code>board-mediated-inter-agent-coordination</code> , <code>workspace-native-multi-agent-orchestration</code> , <code>oracle-and-worker-multi-model</code> , <code>multi-model-orchestration-for-complex-edits</code> .	Choosing Multi-Agent Topology, Agents As Services, A2A Agent Interoperability, Supervisor / Worker.

This local source is best treated as a checklist, not as the book's taxonomy. Some entries are production-proven, some are emerging, and some are research or product-specific. The book should keep absorbing the useful engineering ideas while preserving its own structure: boundaries first, then patterns, then evaluation and operations.

Local PDF References Reviewed

The following PDFs were reviewed or intake-scanned locally as source-of-knowledge inputs. They are not included in this repository, linked from the book, or reproduced. They inform coverage checks, chapter structure, and topic prioritization only.

Local Reference	Useful Themes	Book Mapping
<i>AI Agents in Action</i> , Micheal Lanham	Agent definitions, LLM interfaces, GPT assistants, AutoGen and CrewAI multi-agent systems, agent actions, behavior-tree orchestration, agent platforms, RAG, memory, prompt flow, reasoning, evaluation, planning, and feedback.	What Is An Agent? , Framework Selection , CrewAI Flows and Crews , Tool Use , Memory-Augmented Agent , Planning and Execution , Evaluation-Driven Agent Development .
<i>30 Agents Every AI Engineer Must Build</i> , Imran Ahmad	Agent engineering foundations, toolkit decisions, domain agents, retrieval agents, tool orchestration, software agents, explainability, and domain-specific agents.	Agent Engineer Toolkit , Framework Selection , Domain Agent Architectures , Coding Agents , Agentic RAG Systems .
<i>Agentic Architectural Patterns for Building Multi-Agent Systems</i> , Ali Arsanjani, Juan Pablo Bustos, Thomas Kurian	Enterprise maturity, agent-ready model selection, RAG-to-fine-tuning spectrum, coordination patterns, compliance, robustness, human-agent interaction, production readiness.	Choosing the Right Pattern , Agent Development Lifecycle , Agent Security and Sandboxing , Domain Agent Architectures , Agentic System Architecture .
<i>Agentic Design Patterns</i> , Antonio Gulli	Prompt chaining, routing, parallelization, reflection, tool use, planning, memory, MCP, A2A, monitoring, guardrails, resource optimization, CLI and coding agents.	Prompt Chaining and Gates , Routing and Handoffs , Resource-Aware Agent Design , MCP-first Tool Use , A2A Agent Interoperability .
<i>Designing Multi-Agent Systems</i> , Victor Dibia	Multi-agent taxonomy, UX principles, execution loops, cancellation, memory, middleware, computer-use agents, workflow graphs, observability, evaluation, distributed protocols, ethics.	Agent UX and Human Trust , Evaluation-Driven Agent Development , Computer-Use Agents , Supervisor / Worker , Secure Agent Communication .
<i>Patterns for Building AI Agents</i> , Sam Bhagwat and Michelle Gienow	Agent capability design, context engineering, context compression, eval suites, production data evaluation, sandboxing, granular access control, guardrails.	Context Budgets and Working Sets , Context Engineering , Resource-Aware Agent Design , Evaluation-Driven Agent Development , Agent Security and Sandboxing .

Local Reference	Useful Themes	Book Mapping
<i>Build a Multi-Agent System (from Scratch)</i> , Val Andrei Fajardo	Foundational LLM agent construction, tools, LLM interfaces, MCP tools, skills, memory, human-in-the-loop, A2A, and building a small educational framework from first principles.	Tool Use, MCP-first Tool Use, Skills, Memory-Augmented Agent, A2A Agent Interoperability.
<i>Build an AI Agent (From Scratch)</i> , Jungjun Hur and Younghee Song	LLM interface design, tool use, ReAct, RAG, memory, planning, reflection, code execution, multi-agent orchestration, and agent evaluation.	Single Agent, ReAct, Semantic Recall and RAG, Planning and Execution, Evaluation-Driven Agent Development.
<i>Designing AI Agents</i> , Jia Huang	Agent harnesses, bounded resource allocation, cognitive functions, execution topology, governance, and a running code-review-agent example.	Agentic System Architecture, Resource-Aware Agent Design, Coding Agents, Policy Enforcement, Architecture Decision Records.
<i>Multi-Agent Systems with AutoGen</i> , Victor Dibia	AutoGen-based multi-agent foundations, UX, interface agents, evaluation, optimization, deployment, messaging protocols, safety, and sandboxing.	CrewAI Flows and Crews, Agent UX and Human Trust, Computer-Use Agents, Evaluation-Driven Agent Development, Agent Security and Sandboxing.
<i>AI Agents and Applications</i> , Roberto Infante	LangChain, LangGraph, MCP, RAG, tool-based agents, multi-agent systems, memory, guardrails, and productionization.	Framework Selection, Agentic RAG Systems, MCP-first Tool Use, Multi-Agent Systems, Observability and Evals.
<i>Agentic Transformation Playbook</i>	Business adoption framing, agent lifecycle management, governance, human role design, and common enterprise use cases.	Agent Development Lifecycle, Agent UX and Human Trust, Policy Enforcement, Domain Agent Architectures.

Pattern Coverage Map

External Pattern Name	Current Book Chapter
Augmented LLM	Single Agent, Tool Use, Structured Output
Prompt chaining	Prompt Chaining and Gates
Routing	Routing and Handoffs
Parallelization	Parallel Agents

External Pattern Name	Current Book Chapter
Orchestrator-workers	Supervisor / Worker, Planning and Execution
Evaluator-optimizer	Evaluator-Optimizer
Reflection	Reflection
ReAct	ReAct
Agent loop	Agent Loop
Human-in-the-loop	Human Approval Gates
Tool use	Tool Use, MCP-first Tool Use
Memory and context	Context Engineering, Working Memory, Long-Term Episodic Memory
Resource-aware optimization	Resource-Aware Agent Design
Agentic RAG	Semantic Recall and RAG, Agentic RAG Systems
Multi-agent supervisor	Supervisor / Worker
Peer or network agents	A2A Agent Interoperability, Debate and Consensus
Circuit breaker	Circuit Breakers, Fallbacks, and Replay
Action replay	Circuit Breakers, Fallbacks, and Replay, Observability and Evals
Deterministic chain	Choosing the Right Pattern, Durable Workflows
Coding agent	Coding Agents
Computer-use agent	Computer-Use Agents
Domain-specific agent	Domain Agent Architectures

Maintenance Checklist

Update this page when a source changes the book's structure, not every time a new link appears.

Before adding a source, check:

- Does it introduce a pattern, failure mode, runtime concern, or implementation technique not already covered?
- Is it stronger than an existing source for the same claim?
- Does it map to at least one chapter where a reader can act on the idea?
- Is it public enough for an online GitHub Pages reader to inspect?
- Does it create a concrete editorial task, such as a missing example, checklist, diagram, lab, or cross-link?

If the answer is mostly no, leave the source out. A shorter map with clear editorial purpose is more useful than a large reference pile.

Editorial Rule

External sources are used to validate coverage and expose missing patterns. The book keeps its own taxonomy:

- foundations first;
- control loops second;
- memory and knowledge as a separate concern;
- tools, skills, and protocols as integration boundaries;
- multi-agent systems only when specialization has value;
- production runtime patterns for safety, evaluation, and operations;
- source-informed pattern selection as the entry point for choosing the right design.

This keeps the book useful as a reference instead of turning it into a link dump.

Single Agent

A single agent receives a goal or message, consults its context, and produces an answer or action. This is the smallest useful unit in the catalog.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

A single agent receives a goal or message, consults its context, and produces an answer or action. This is the smallest useful unit in the catalog.

Scenario

A support team wants a small assistant that rewrites rough internal notes into a customer-facing reply. The input contains the ticket summary, policy excerpt, and desired tone. The output is a draft response with no tool calls, no memory writes, and no payment authority.

This is a good single-agent baseline because the task has one worker, one bounded context packet, and one typed output. The caller still owns ticket state, policy version, escalation, and final send. The agent owns only the draft.

```
input:
  ticket_id: T-918
  customer_issue: "Package arrived two days late."
  policy_excerpt: "Late delivery may receive shipping-fee credit, not full refund."
  requested_output: "draft customer reply"

single_agent_allowed:
  - summarize issue
  - draft reply
  - explain policy-backed next step

single_agent_forbidden:
  - issue refund
  - update ticket status
  - send message to customer
  - remember customer preference
```

If the team later needs live order lookup, policy retrieval, approval, or message sending, this pattern has reached its boundary. Keep the single agent as the drafting worker and put tools, state, approval, and delivery in the surrounding workflow.

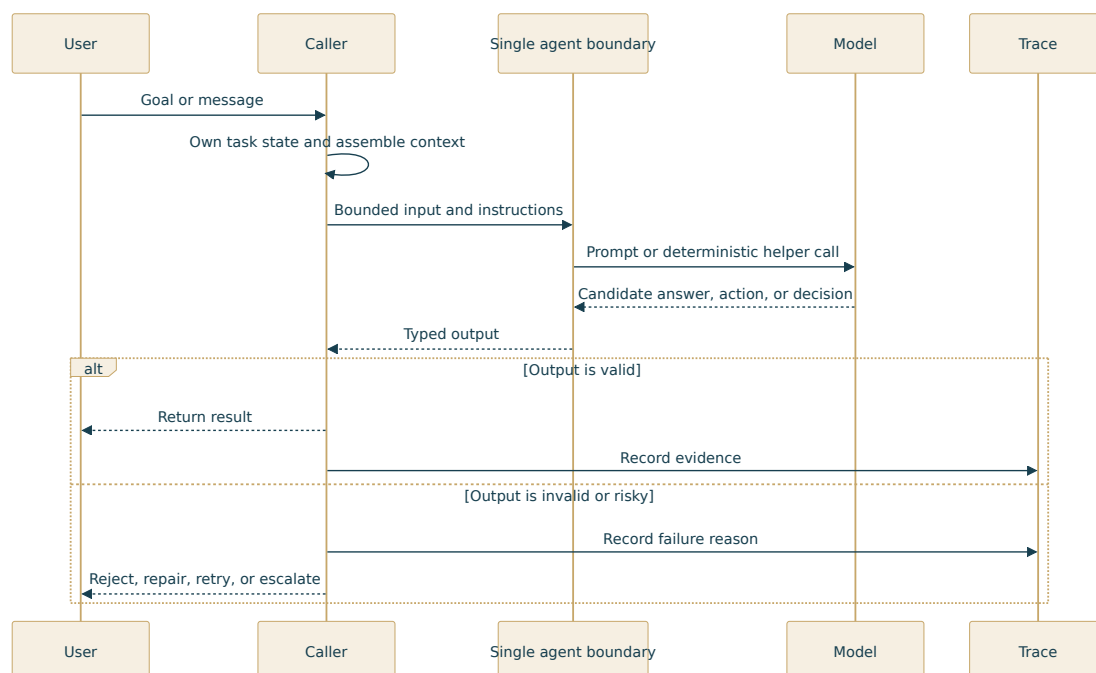
Use When

- One model-backed worker can complete the task without delegation.
- The interaction has a narrow objective and a clear success condition.
- You want the simplest useful baseline before adding tools, memory, or orchestration.

Avoid When

- The task needs stateful retries, external approvals, multiple specialists, or independent evaluation.
- The agent must recover from long-running failure or resume after interruption.

Architecture



Use this as the baseline architecture. If the system needs durable retries, independent evaluation, tool orchestration, or specialist handoffs, it has moved beyond the single-agent pattern.

System Shape

- **Pattern boundary:** a narrow agent function, class, or service boundary accepts input plus context and returns a typed answer, action, or decision.
- **State owner:** the caller or a small application service owns task state until a runtime pattern is introduced.
- **Primary artifact:** `single-agent-pattern/` contains the runnable reference implementation and examples.

- **Operational promise:** A single agent receives a goal or message, consults its context, and produces an answer or action. This is the smallest useful unit in the catalog.
- **Runnable path:** start with `npm run single-agent` before adapting the pattern to a larger system.

Core Protocol

1. Accept a bounded input, goal, or task request.
2. Assemble the minimum useful instructions, context, state, and tool descriptions.
3. Run the model or deterministic helper behind a typed boundary.
4. Validate the result before returning it to users, tools, or durable state.
5. Record enough evidence to explain the output later.

Implementation Notes

- Keep the pattern boundary explicit: inputs, state, side effects, and outputs should be visible.
- Validate model-produced decisions before they affect tools, users, or durable state.
- Emit enough trace data to debug failures after the run.

Failure Modes

- The pattern is applied where a simpler deterministic workflow would be better.
- State, tool calls, or model decisions are not observable enough to debug.
- The system lacks clear stop, retry, or escalation behavior.

Evaluation Strategy

- Use golden tasks that cover normal requests, ambiguous requests, missing context, and invalid input.
- Check that outputs match the expected shape and that unsafe or unsupported requests are rejected.
- Track accuracy, schema validity, latency, token use, and refusal quality.
- Include cases that prove each “Use When” condition is true for this pattern.
- Include negative cases from “Avoid When” so the system chooses a simpler or safer pattern when appropriate.

Production Checklist

- Define the input, context, output, and error contract.
- Keep prompts, schemas, and tool descriptions versioned.
- Add deterministic tests for the smallest useful behavior.
- Log model decisions without leaking secrets or private user data.
- Define human escalation for ambiguous, high-risk, or policy-blocked work.
- Keep the source bundle, generated chapter, tests, and deployment artifact in the same release.

Run the Example

```
npm run single-agent
```

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
single-agent-pattern/autogen_typescript_example/single_agent.ts
```

[Open full source](#)

```
// Single Agent Pattern - Autogen TypeScript Example
// To run: npm install && npm run single-agent

import axios from 'axios';
import * as readline from 'readline';
import * as dotenv from 'dotenv';
dotenv.config();

const MISTRAL_API_URL = 'https://api.mistral.ai/v1/chat/completions';
const MISTRAL_API_KEY = process.env.MISTRAL_API_KEY;

async function singleAgent(userInput: string): Promise<string> {
  const response = await axios.post(
    MISTRAL_API_URL,
    {
      model: 'mistral-tiny', // or your preferred Mistral model
      messages: [{ role: 'user', content: userInput }],
    },
    {
      headers: {
        'Authorization': `Bearer ${MISTRAL_API_KEY}`,
        'Content-Type': 'application/json',
      },
    }
  );
};
```



```

    return response.data.choices[0].message.content;
}

async function main() {
    const idx = process.argv.indexOf('--input');
    const cliInput = idx !== -1 ? process.argv[idx + 1] : undefined;
    const nonInteractive = cliInput || process.env.NON_INTERACTIVE_INPUT;
    if (nonInteractive) {
        try {
            const agentResponse = await singleAgent(String(nonInteractive));
            console.log('Agent:', agentResponse);
        } catch (err) {
            console.error('Error:', err);
            process.exitCode = 1;
        }
        return;
    }

    const rl = readline.createInterface({ input: process.stdin, output: process.stdout });
    rl.question('User: ', async (userInput: string) => {
        try {
            const agentResponse = await singleAgent(userInput);
            console.log('Agent:', agentResponse);
        } catch (err) {
            console.error('Error:', err);
        }
        rl.close();
    });
}

main();

// duplicate block removed

```

[single-agent-pattern/langgraph_python_example/single_agent.py](#)

[Open full source](#)

```
# Single Agent Pattern - LangGraph Python Example
```

This example demonstrates the Single Agent Pattern using LangGraph and Python. The agent receives a user message, sends it to a Mistral LLM, and returns the response.

```
## Requirements
```

```

- Python 3.8+
- `langgraph` library
- `python-dotenv` (for .env support)
- Mistral LLM API access

## Install dependencies

```bash
pip install langgraph python-dotenv requests
```

## Example Code

```python
import os
from langgraph import Agent, Environment, LLM
from dotenv import load_dotenv

load_dotenv()

MISTRAL_API_KEY = os.getenv("MISTRAL_API_KEY")
MISTRAL_API_URL = "https://api.mistral.ai/v1/chat/completions"

class SimpleEnvironment(Environment):
 def get_observation(self):
 return input("User: ")
 def send_action(self, action):
 print(f"Agent: {action}")

class SingleAgent(Agent):
 def __init__(self, llm):
 self.llm = llm
 def act(self, observation):
 return self.llm.complete(observation)

llm = LLM(
 provider="mistral",
 api_key=MISTRAL_API_KEY,
 api_url=MISTRAL_API_URL,
)

env = SimpleEnvironment()
agent = SingleAgent(llm)

```

```
observation = env.get_observation()
action = agent.act(observation)
env.send_action(action)
'''
'''
'''
'''
```

- Make sure your ``.env`` file contains your Mistral API key as ``MISTRAL_API_KEY``.
- This is a minimal, functional example.

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `single-agent-pattern/` folder from this repository.

## Related Patterns

- [Agent Loop](#)
- [Goals and State](#)
- [Tool Use](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# Agent Loop

The agent loop turns a model call into an agent: observe state, decide the next action, act, evaluate the result, and stop when the goal is complete or a limit is reached.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

The agent loop is the smallest runtime structure that makes a model act like an agent. It observes state, decides a next step, validates the proposed action, acts through a bounded interface, records the observation, and stops when success, failure, budget, or escalation says the run is over.

The loop is powerful because the next step can depend on what just happened. It is dangerous for the same reason. Without explicit state, budgets, validation, and stop reasons, the loop becomes a place where cost, latency, unsafe tool use, and hidden failures accumulate.

This pattern owns iteration: proposal, validation, action, observation, budget, and stop reason. It does not own the business goal, tool implementation, durable workflow, or production control plane. Those responsibilities belong to the caller, tool layer, workflow engine, and runtime.

## Use When

- The task cannot be solved by a fixed sequence known ahead of time.
- The next step depends on observations from tools, retrieval, users, or intermediate outputs.
- The agent may need multiple attempts, but those attempts can be bounded.
- The system can persist enough loop state to replay or debug the run.
- You can define success, failure, budget exhaustion, and escalation conditions.

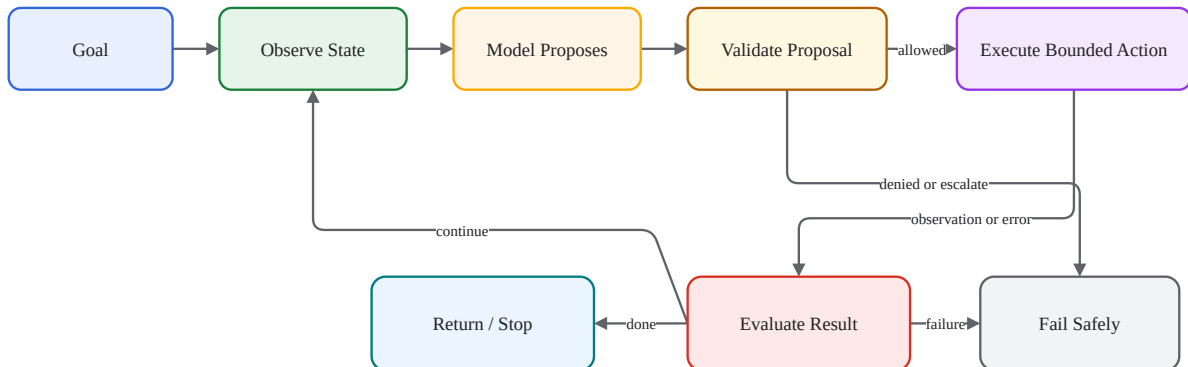
## Avoid When

- A deterministic workflow or prompt chain can express the steps clearly.
- The task has no useful intermediate observations.
- Tool use, cost, runtime, or side effects cannot be bounded.
- The loop state would live only inside the model context.
- The team cannot explain why the loop stopped.

## Architecture

Use this diagram to read Agent Loop as a system boundary, not only a code shape. The key ownership question is: the caller or a small application service owns task state until a runtime pattern is introduced.

## Agent Loop



## System Shape

- **Pattern boundary:** a loop controller owns progress, budget, validation, state updates, and stop decisions.
- **State owner:** the loop runtime owns task-local state during the run and checkpoints durable state when the run must survive interruption.
- **Model role:** the model proposes the next step, but software validates whether the step is allowed.
- **Action boundary:** tools, handoffs, memory writes, and final answers pass through typed validation.
- **Operational promise:** the loop handles uncertainty without giving the model unlimited time, tools, or authority.

## Core Protocol

1. Receive a task with a goal, caller, constraints, budget, and trace ID.
2. Load or initialize loop state: observations, evidence, attempts, errors, and remaining budget.
3. Assemble the working set for the next step.
4. Ask the model to propose either an action, a final result, a refusal, or an escalation.
5. Validate the proposal against schema, tools, policy, state, and budget.
6. Execute the allowed action and record the observation.
7. Update state, budget counters, trace events, and stop reason.
8. Stop on success, failure, refusal, cancellation, budget exhaustion, or escalation.

## Implementation Notes

Keep the controller boring. The loop should be easy to inspect.

```
type StopReason =
 | 'completed'
 | 'failed'
```

```

| 'refused'
| 'needs_human'
| 'max_steps'
| 'max_tool_calls'
| 'timeout';

type LoopState = {
 taskId: string;
 goal: string;
 step: number;
 toolCalls: number;
 startedAtMs: number;
 observations: unknown[];
 errors: string[];
};

type LoopBudget = {
 maxSteps: number;
 maxToolCalls: number;
 timeoutMs: number;
};

function shouldStop(state: LoopState, budget: LoopBudget, nowMs: number): StopReason | null
{
 if (state.step >= budget.maxSteps) return 'max_steps';
 if (state.toolCalls >= budget.maxToolCalls) return 'max_tool_calls';
 if (nowMs - state.startedAtMs >= budget.timeoutMs) return 'timeout';
 return null;
}

```

A minimal loop controller can then enforce the boundary:

```

async function runAgentLoop(task: AgentTask, budget: LoopBudget) {
 const state = initializeLoopState(task);

 while (true) {
 const stopReason = shouldStop(state, budget, Date.now());
 if (stopReason) return finishRun(state, stopReason);

 const proposal = await proposeNextStep(task, state);
 const decision = validateProposal(proposal, state, task.policy);

 if (decision.status === 'deny') {
 return finishRun(state, 'refused');
 }
 }
}

```

```

 }

 if (decision.status === 'escalate') {
 return finishRun(state, 'needs_human');
 }

 if (decision.status === 'final') {
 return finishRunWithResult(state, decision.result, 'completed');
 }

 const observation = await executeAction(decision.action, {
 traceId: task.traceId,
 idempotencyKey: `${task.taskId}:${state.step}`
 });

 recordObservation(state, observation);
 if (decision.action.kind === 'tool') state.toolCalls += 1;
 state.step += 1;
 }
}

```

The controller does not need to be complicated. It needs to make the hidden parts explicit: the active goal, the proposal, the validation decision, the action, the observation, the budget counters, and the stop reason.

### Running Case: Bounded Refund Investigation

In the support refund system, the whole product should not be one open-ended loop. The workflow should call a loop only for the uncertain part: investigate whether the available evidence supports the refund recommendation.

```

goal: "Decide whether refund evidence is sufficient for order 0-104."
allowed_actions:
 - read_order
 - read_delivery_status
 - search_refund_policy
 - ask_customer_clarifying_question
 - draft_refund_recommendation
forbidden_actions:
 - issue_refund
 - change_payment_method
 - update_policy
stop_reasons:
 - completed

```

```
- evidence_missing
- policy_blocked
- needs_human
- max_steps
```

This keeps the loop useful and small. It can decide the next evidence-gathering step, but it cannot issue the refund. The payment action stays behind policy validation, approval, idempotency, and audit.

## Failure Modes

- The goal is vague, so the loop keeps working without a stable success condition.
- The model is allowed to decide when the loop should stop without a software-owned budget.
- The loop repeats the same tool call because it does not compare new observations with prior observations.
- Retries happen without idempotency, duplicating side effects.
- State exists only in the prompt, so replay and recovery are impossible.
- The loop summarizes away evidence that was needed for the next decision.
- Tool errors are treated as normal observations, causing confused follow-up actions.
- The agent declares success because it produced an answer, not because the goal was satisfied.
- Traces capture the final output but not the iteration history.

## Evaluation Strategy

Loop evals should test the trajectory, not only the final response.

- Test a task that completes within budget.
- Test a task that must stop on `max_steps`.
- Test a task that must stop on `max_tool_calls`.
- Test a task with a malformed tool result.
- Test a task with repeated tool failure that should escalate.
- Test a task where the model proposes a forbidden action.
- Test a task where the model claims completion but required evidence is missing.
- Test replay from a saved loop state.

A compact eval fixture can make the expected stop behavior explicit:

```
{
 "case_id": "shipping_lookup_repeated_failure",
 "goal": "Investigate whether an order arrived late.",
 "mocked_tools": {
 "shipping.read_delivery_status": [
 { "status": "retryable_error", "reason": "upstream_timeout" },
 { "status": "retryable_error", "reason": "upstream_timeout" }
]
 }
},
```



```
"expected": {
 "stop_reason": "needs_human",
 "max_tool_calls": 2,
 "forbidden_tools": ["refunds.issue_refund"],
 "requires_trace_events": ["proposal", "validation", "tool_result", "stop"]
}
```

Measure completion rate, correct stop reason, invalid-action rate, repeated-action rate, escalation accuracy, token and tool cost, latency, and replay success.

## Production Checklist

- Define the loop goal and success criteria before the first model call.
- Store loop state outside the model context.
- Set hard limits for steps, tool calls, wall-clock time, retries, and cost.
- Validate every action before execution.
- Make side-effectful actions idempotent or approval-gated.
- Record proposal, validation decision, action, observation, error, and stop reason for every iteration.
- Treat cancellation, refusal, timeout, and escalation as normal outcomes.
- Add circuit breakers for repeated failures or unsafe action proposals.
- Keep loop prompts, tool manifests, policies, and model routes versioned.
- Replay production failures into regression evals.

The architectural rule is simple: the model may choose the next proposal, but software owns whether the loop continues. Continue with [Goals and State](#) to define what the loop carries, then [Tool Use](#) to define how it acts.

## Run the Example

```
npm run agent-loop
npm run agent-loop:test
```

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

[Open full source](#)

```
export type StopReason =
 | "completed"
 | "refused"
 | "needs_human"
 | "max_steps"
 | "tool_failure";

export type ToolProposal = {
 kind: "tool";
 name: "lookup_order";
 input: { orderId: string };
};

export type Proposal =
 | { kind: "answer"; text: string }
 | ToolProposal
 | { kind: "tool"; name: string; input: unknown }
 | { kind: "escalate"; reason: string };

export type Observation = {
 tool: string;
 status: "ok" | "error";
 output: unknown;
};

export type LoopState = {
 goal: string;
 step: number;
 observations: Observation[];
};

export type LoopResult = {
 stopReason: StopReason;
 answer?: string;
 state: LoopState;
 trace: string[];
};

export type LoopDependencies = {
 propose(state: LoopState): Promise<Proposal>;
```

```

execute(
 proposal: ToolProposal,
 idempotencyKey: string,
): Promise<Observation>;
};

type ValidatedDecision =
 | { status: "final"; answer: string }
 | { status: "execute"; proposal: ToolProposal }
 | { status: "escalate"; reason: string }
 | { status: "deny"; reason: string };

export function validateProposal(proposal: Proposal): ValidatedDecision {
 if (proposal.kind === "answer") {
 return proposal.text.trim()
 ? { status: "final", answer: proposal.text }
 : { status: "deny", reason: "empty_answer" };
 }

 if (proposal.kind === "escalate") {
 return { status: "escalate", reason: proposal.reason };
 }

 if (proposal.name !== "lookup_order") {
 return { status: "deny", reason: "tool_not_allowed" };
 }

 const input = proposal.input as { orderId?: unknown };
 if (typeof input.orderId !== "string" || !input.orderId.trim()) {
 return { status: "deny", reason: "invalid_tool_input" };
 }

 return {
 status: "execute",
 proposal: {
 kind: "tool",
 name: "lookup_order",
 input: { orderId: input.orderId },
 },
 };
}

export async function runAgentLoop(
 goal: string,

```

```

 maxSteps: number,
 dependencies: LoopDependencies,
): Promise<LoopResult> {
 const state: LoopState = { goal, step: 0, observations: [] };
 const trace: string[] = [];

```

*Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.*

[agent-loop-pattern/typescript/test/agent\\_loop.spec.ts](#)

[Open full source](#)

```

import {
 runAgentLoop,
 type LoopDependencies,
 type LoopState,
 type Proposal,
} from "../src/agent_loop.ts";

function assert(condition: unknown, message: string): asserts condition {
 if (!condition) throw new Error(message);
}

function scriptedDependencies(
 proposals: Proposal[],
 toolStatus: "ok" | "error" = "ok",
): LoopDependencies {
 return {
 propose: async (_state: LoopState) =>
 proposals.shift() ?? { kind: "escalate", reason: "script_exhausted" },
 execute: async (proposal, idempotencyKey) => ({
 tool: proposal.name,
 status: toolStatus,
 output: { idempotencyKey },
 })),
 };
}

const completed = await runAgentLoop(
 "Read an order",
 3,
 scriptedDependencies([
 { kind: "tool", name: "lookup_order", input: { orderId: "A-104" } },

```

```

 { kind: "answer", text: "The order shipped." },
],
);
assert(completed.stopReason === "completed", "Expected completed run");
assert(completed.state.observations.length === 1, "Expected one observation");
assert(
 completed.trace.includes("step:0:validation:execute"),
 "Expected validation trace",
);

const denied = await runAgentLoop(
 "Delete an order",
 3,
 scriptedDependencies([
 { kind: "tool", name: "delete_order", input: { orderId: "A-104" } },
]),
);
assert(denied.stopReason === "refused", "Expected forbidden tool refusal");
assert(denied.state.observations.length === 0, "Denied tool must not execute");

const failed = await runAgentLoop(
 "Read an order",
 3,
 scriptedDependencies(
 [{ kind: "tool", name: "lookup_order", input: { orderId: "A-104" } }],
 "error",
),
);
assert(failed.stopReason === "tool_failure", "Expected tool failure stop");

const exhausted = await runAgentLoop(
 "Keep checking",
 1,
 scriptedDependencies([
 { kind: "tool", name: "lookup_order", input: { orderId: "A-104" } },
 { kind: "answer", text: "This proposal must not run." },
]),
);
assert(exhausted.stopReason === "max_steps", "Expected max_steps stop");

console.log("Agent loop tests OK");

```

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `agent-loop-pattern/` folder from this repository.

## Related Patterns

- [Goals and State](#)
- [Tool-Using Agent](#)
- [Structured Output](#)
- [Evaluator-Optimizer](#)
- [Durable Workflows](#)
- [Pattern Evaluation Checklist](#)
- [Evaluation-Driven Agent Development](#)

# Goals and State

Goals define success; state records progress. Together they make agent work resumable, inspectable, and easier to evaluate.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

The Goals and State Pattern separates what the agent is trying to achieve from the mutable state it accumulates while working. Goals define success. State records progress, constraints, evidence, pending work, approvals, budget, and stop reasons.

This is the practical core of working memory. Working memory is not long-term memory. It is run-scoped operational state. It should be compact, typed, inspectable, and disposable unless another policy explicitly promotes part of it into durable memory.

Use this pattern for task-local state. Durable memory is handled by [Memory-Augmented Agent](#). Context assembly is handled by [Context Engineering](#).

The model can propose a next step, a state update, or a stop reason. The runtime owns the state transition. That boundary keeps the agent from silently rewriting what has happened, what remains, or why it is allowed to continue.

## Use When

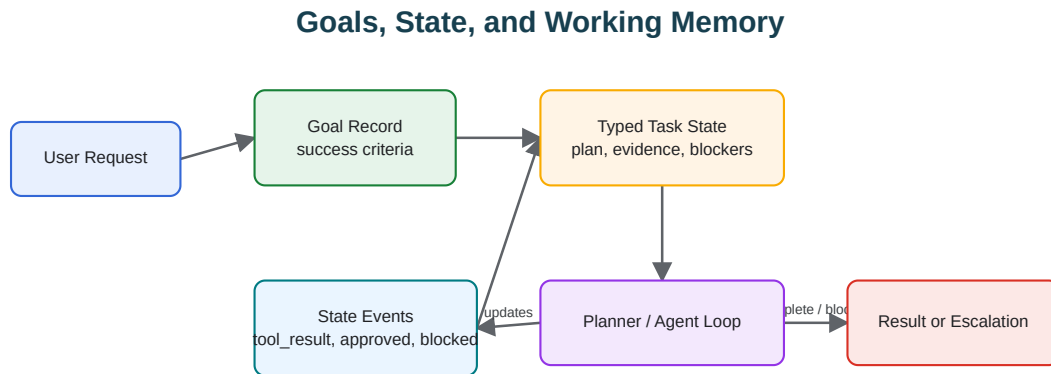
- A task spans multiple turns, tools, agents, retries, approvals, or workflow steps.
- You need resumable execution after failure, interruption, timeout, or human approval.
- The agent must explain progress against an explicit objective.
- You need a compact working set instead of raw chat history.
- Evaluation depends on trajectory, not only the final answer.

## Avoid When

- The task is stateless and can be answered in one call.
- The goal cannot be expressed as observable success criteria.
- State would contain sensitive data you cannot store safely.
- The system would use state as an unstructured scratchpad with no owner.
- The runtime cannot replay or inspect state transitions.

## Architecture

Use this diagram to read Goals and State as a system boundary, not only a code shape. The key ownership question is: the caller or a small application service owns task state until a runtime pattern is introduced.



## System Shape

- **Pattern boundary:** the state boundary owns the goal, working state, event log, transition rules, stop reason, and replay record.
- **State owner:** the runtime, workflow engine, or application service owns state; the model owns only proposed updates.
- **Model role:** the model may summarize progress, identify gaps, and propose updates, but it should not be the source of truth for state.
- **Policy boundary:** state changes that affect permissions, approvals, memory promotion, or side effects pass through runtime checks.
- **Operational promise:** the system can explain what it is doing, what changed, what evidence caused the change, and why it stopped or continued.

## Core Protocol

1. Create a goal record with success criteria, constraints, owner, risk class, and stop conditions.
2. Initialize working memory with the minimum state needed to start.
3. Execute one bounded step through a planner, model call, tool, worker, evaluator, or approval gate.
4. Convert observations into typed state events.
5. Apply state transitions idempotently.
6. Recompute open questions, pending work, budget state, approval state, and stop reason.
7. Continue, retry, ask for approval, escalate, or stop according to explicit rules.
8. Persist enough state and event history to replay or audit the run.

## Implementation Notes



State should be smaller than the transcript and more structured than a summary. It is the operating model for the current run.

## Goal Record

```
type AgentGoal = {
 goalId: string;
 runId: string;
 owner: string;
 description: string;
 successCriteria: string[];
 constraints: string[];
 riskClass: "low" | "medium" | "high" | "critical";
 status: "active" | "blocked" | "waiting_for_approval" | "completed" | "cancelled" |
 "failed";
 createdAt: string;
 updatedAt: string;
};
```

## Working Memory State

```
type WorkingMemoryState = {
 runId: string;
 goalId: string;
 currentStep?: string;
 completedSteps: string[];
 pendingSteps: string[];
 openQuestions: string[];
 constraints: string[];
 evidenceRefs: string[];
 toolResultRefs: string[];
 approvalRefs: string[];
 budgetState: {
 iterationCount: number;
 tokenEstimate: number;
 costCents: number;
 deadlineAt?: string;
 };
 stopReason?: "success" | "blocked" | "approval_required" | "budget_exhausted" |
 "cancelled" | "failed";
 version: number;
};
```

## State Events

State should change through events, not hidden mutation.

```
type StateEvent =
 | { type: "goal_created"; goal: AgentGoal }
 | { type: "step_started"; stepId: string; description: string }
 | { type: "tool_result_recorded"; toolCallId: string; evidenceRef: string }
 | { type: "approval_recorded"; approvalId: string; decision: "approved" | "denied" |
 "expired" }
 | { type: "question_opened"; question: string }
 | { type: "question_answered"; question: string; evidenceRef: string }
 | { type: "blocked"; reason: string }
 | { type: "completed"; resultRef: string }
 | { type: "cancelled"; reason: string };

function applyStateEvent(state: WorkingMemoryState, event: StateEvent): WorkingMemoryState
{
 switch (event.type) {
 case "step_started":
 return { ...state, currentStep: event.stepId, version: state.version + 1 };
 case "tool_result_recorded":
 return {
 ...state,
 evidenceRefs: [...state.evidenceRefs, event.evidenceRef],
 toolResultRefs: [...state.toolResultRefs, event.toolCallId],
 version: state.version + 1,
 };
 case "approval_recorded":
 return {
 ...state,
 approvalRefs: [...state.approvalRefs, event.approvalId],
 stopReason: event.decision === "approved" ? undefined : "blocked",
 version: state.version + 1,
 };
 case "completed":
 return { ...state, stopReason: "success", version: state.version + 1 };
 case "cancelled":
 return { ...state, stopReason: "cancelled", version: state.version + 1 };
 default:
 return { ...state, version: state.version + 1 };
 }
}
```

This is intentionally simple. The important part is the rule: state transitions are explicit, typed, replayable, and tied to observations.

## Running Case: Refund Goal And State

A refund assistant needs more than a chat transcript because operators must know what evidence the system used before money moved. The goal record should name the outcome and the risk; the working state should show the evidence, open questions, approval state, and stop reason.

```
goal:
 description: "Resolve refund eligibility for order 0-104."
 success_criteria:
 - "current refund policy is referenced"
 - "order and delivery evidence are attached"
 - "recommendation is validated by policy gate"
 - "high-value refund waits for approval"
 risk_class: "high"
state:
 completed_steps:
 - "order_loaded"
 - "delivery_status_loaded"
 open_questions:
 - "policy exception evidence is missing"
 evidence_refs:
 - "order:0-104"
 - "delivery:D-88"
 - "policy:refunds:v2026-06"
 approval_refs: []
 stop_reason: "approval_required"
```

This state tells the runtime what happened and what remains. It also gives evals something concrete to inspect: missing policy should block the recommendation, denied approval should block payment, and a resumed run should preserve the same evidence references.

## Promotion Rules

Working memory should not automatically become durable memory. A completed step, tool result, or user correction may be a candidate for long-term memory, but promotion needs a separate policy decision.

Before promoting working memory, ask:

- Is this useful beyond the current run?
- Is the source trustworthy and referenced?
- Does the user or tenant allow it to be stored?
- Is there sensitive data that needs redaction or expiry?

- Is this an event, a fact, a preference, or a policy reference?
- Is there a correction and deletion path?

If the answer is unclear, keep it in the run trace and do not promote it.

## Failure Modes

- The state becomes a transcript dump.
- The model silently rewrites state without an event.
- The goal describes activity, not success.
- Subgoals drift away from the parent goal.
- Tool results are summarized without evidence references.
- Approval, cancellation, or budget state is lost after retry.
- The system cannot resume because state only lived in model context.
- Stale state overrides fresh tool results.
- Sensitive state is persisted without retention or redaction rules.
- Working memory is automatically promoted to long-term memory.

## Evaluation Strategy

Working-memory evals should inspect the trajectory, not only the final answer.

- Test that a multi-step task creates a goal with observable success criteria.
- Test that each tool result becomes a state event with an evidence reference.
- Test retry and verify idempotent state transitions.
- Test resume after interruption.
- Test cancellation and verify the run does not continue.
- Test approval waits and verify approval state survives resume.
- Test budget exhaustion and verify the stop reason is recorded.
- Test stale tool results and verify fresh evidence wins.
- Test blocked runs and verify open questions are preserved.
- Test that working memory is not promoted to durable memory without policy.

Measure state completeness, transition validity, replay success, resume success, stale-state rate, lost-approval rate, stop-reason accuracy, and promotion-policy accuracy.

## Production Checklist

- Define goal schema, state schema, event schema, and stop reasons.
- Keep working memory compact and typed.
- Store state separately from chat history.
- Make state updates event-based, idempotent, and replayable.
- Attach evidence references to tool results, decisions, approvals, and final answers.
- Persist approval, cancellation, budget, retry, and blocked state.
- Treat state promotion to durable memory as a separate policy decision.

- Redact or avoid sensitive data in working memory when possible.
- Trace state reads, writes, transitions, retries, resumes, and stop reasons.
- Convert lost-state and wrong-stop incidents into regression evals.

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

This pattern currently has no dedicated code excerpt. Use the source and download links below for the full pattern folder.

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `goals-and-state-pattern/` folder from this repository.

## Related Patterns

- [Agent Loop](#)
- [Planning Pattern](#)
- [Durable Workflow](#)
- [Memory-Augmented Agent](#)
- [Long-Term Episodic Memory](#)
- [Human Approval Gates](#)

# Tool Use

Tool use gives an agent controlled access to external capability such as calculators, search, databases, files, code execution, APIs, or business systems.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

Tool use lets an agent cross the boundary between language and action. The model can propose a calculation, lookup, retrieval, file operation, API call, workflow step, or business action, but software still owns the real execution boundary.

The important idea is simple: the model does not “use the tool” directly. The model proposes a tool call. The runtime validates the call, checks policy, executes the tool, records the result, and decides whether the observation can influence the next step.

This pattern owns the proposal-to-execution boundary. It does not own business authorization, human approval policy, durable orchestration, or the internal implementation of each capability. Keep those responsibilities in policy, approval, workflow, and service layers.

## Use When

- The task needs facts, computation, retrieval, or system access outside the model context.
- The tool can be expressed as a narrow capability with typed inputs and structured outputs.
- Software can validate arguments before execution.
- Permissions and policy checks can sit between model intent and tool execution.
- The result can be traced, replayed, mocked, or audited.

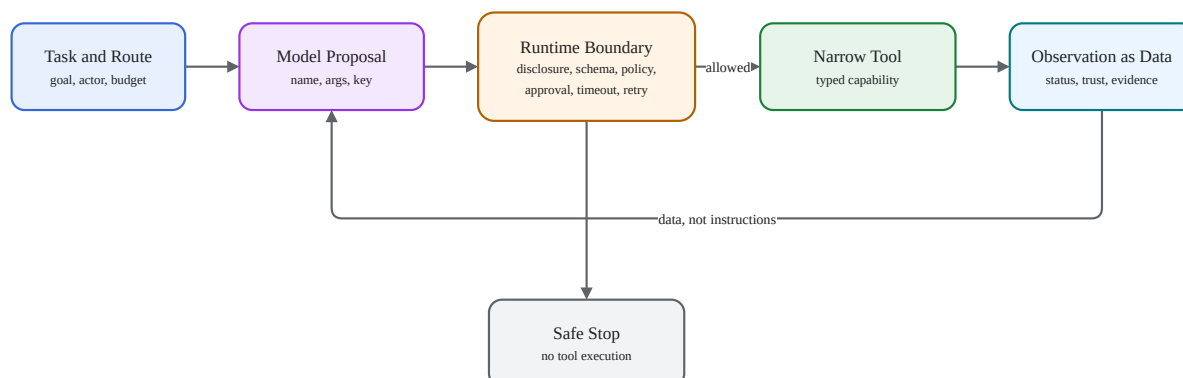
## Avoid When

- A deterministic function or workflow can do the job without model selection.
- The proposed tool is a broad primitive such as `run_sql`, `send_http_request`, or `execute_shell`.
- The tool can create high-risk side effects without approval.
- Tool results contain untrusted content but the system cannot separate data from instructions.
- The team cannot explain which tool calls are allowed, forbidden, retried, or escalated.

## Architecture

Use this diagram to read Tool Use as a system boundary, not only a code shape. The key ownership question is: the caller or a small application service owns task state until a runtime pattern is introduced.

### Tool Use Policy Boundary



## System Shape

- **Pattern boundary:** an agent runtime receives a task, exposes only the tools needed for that task, validates model-proposed calls, and returns a typed result.
- **State owner:** the caller, workflow engine, or agent runtime owns task state. The model should not be the durable state store.
- **Tool owner:** each tool has an owning service or team responsible for schema, permissions, side effects, errors, and trace fields.
- **Policy boundary:** tool execution happens only after schema validation, authorization, budget checks, and approval rules.
- **Operational promise:** tool use expands agent capability without handing the model unrestricted authority.

## Core Protocol

1. Receive a bounded task with caller identity, goal, state reference, and budget.
2. Select the smallest useful tool set for the current task or phase.
3. Ask the model for the next action or final answer.
4. Validate any proposed tool call against schema, permissions, budget, and policy.
5. Execute the tool with timeout, idempotency key, and trace correlation.
6. Return the tool result as observation data, not as new instructions.
7. Stop, retry, escalate, or continue according to explicit runtime rules.

## Implementation Notes

Treat the tool registry as an authority surface. A small registry is better than a large list of vague capabilities.

```

type ToolName = 'read_order' | 'search_refund_policy' | 'draft_refund_request';

type ToolRequest = {
 runId: string;
 callerId: string;
 tool: ToolName;
 args: Record<string, unknown>;
 idempotencyKey: string;
};

const allowedToolsByRoute: Record<string, ToolName[]> = {
 refund_investigation: ['read_order', 'search_refund_policy', 'draft_refund_request']
};

function authorizeToolCall(route: string, request: ToolRequest) {
 const allowed = allowedToolsByRoute[route] ?? [];

 if (!allowed.includes(request.tool)) {
 return { status: 'denied', reason: 'tool_not_allowed' };
 }

 if (!request.idempotencyKey) {
 return { status: 'denied', reason: 'missing_idempotency_key' };
 }

 return { status: 'allowed' };
}

```

The model can choose among `read_order`, `search_refund_policy`, and `draft_refund_request`, but it cannot invent `issue_refund` unless the runtime exposes that tool and policy allows it.

Use structured tool results:

```

type ToolResult =
 | { status: 'ok'; data: unknown; evidenceRef: string }
 | { status: 'refused'; reason: string }
 | { status: 'retryable_error'; reason: string; retryAfterMs?: number }
 | { status: 'fatal_error'; reason: string };

```

Do not return plain strings for important tools. Plain strings force the model to infer whether the call succeeded, whether retry is safe, and whether the content is trusted.

## Failure Modes



- The model is allowed to call a broad tool that can perform many hidden actions.
- Tool descriptions become the only permission boundary.
- Tool arguments are not validated before execution.
- A retry duplicates a side effect because there is no idempotency key.
- Tool results containing emails, web pages, tickets, or documents are treated as instructions.
- The final answer looks correct, but the tool trajectory used a forbidden or unsafe path.
- Tool errors are vague, so the agent retries blindly or invents missing evidence.
- Traces record the final response but not the proposed call, policy decision, tool result, and stop reason.

## Evaluation Strategy

Tool-use evals should test both capability and restraint.

- Use positive cases where the agent must choose the right tool with valid arguments.
- Use negative cases where the correct behavior is to call no tool, ask for missing input, refuse, or escalate.
- Include forbidden-tool cases such as direct refund issuance, external messaging, shell execution, or private-data export.
- Mock tools so evals can inspect the trajectory without touching real systems.
- Test malformed tool results, timeouts, retryable errors, fatal errors, and untrusted instructions inside tool output.
- Measure tool-selection accuracy, invalid-argument rate, unauthorized-call rate, approval-routing accuracy, unsafe-chain prevention, cost, latency, and stop reason quality.

A minimal mocked-tool eval can look like this:

```
{
 "case_id": "refund_missing_policy",
 "input": "Customer asks for a refund, but no refund policy is available.",
 "expected": {
 "tools_called": ["read_order", "search_refund_policy"],
 "tools_not_called": ["draft_refund_request", "issue_refund"],
 "final_status": "needs_human"
 }
}
```

The eval is not only checking the final answer. It is checking the path.

## Production Checklist

- Keep every tool narrow and named by business capability.
- Use typed input and output schemas.
- Declare capability class, side effects, permissions, approval rules, and trace fields.
- Validate arguments before execution.

- Enforce permissions outside the prompt.
- Add timeouts, retry limits, idempotency keys, and cancellation behavior.
- Treat untrusted tool output as data, not instructions.
- Log proposed tool call, policy decision, execution result, latency, cost, and stop reason.
- Mock tools in evals before connecting to production systems.
- Keep a circuit breaker for risky tools, model routes, or agent capabilities.

The architectural rule is simple: expose the smallest capability that completes the task, then validate every proposed use before execution. Continue with [Tool Capability Design](#) for interface design and [Human Approval Gates](#) for high-risk actions.

## Run the Example

```
npm run tool-using-agent
npm run tool-runtime:test
```

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
tool-using-agent-pattern/typescript/src/tool_runtime.ts
```

[Open full source](#)

```
export type Route = "refund_investigation" | "order_status";
export type ToolName =
 | "read_order"
 | "search_refund_policy"
 | "draft_refund_request";

export type ToolProposal = {
 name: string;
 args: unknown;
 idempotencyKey: string;
};

export type ToolObservation =
```

```

| {
 status: "ok";
 tool: ToolName;
 data: unknown;
 trust: "trusted_system" | "untrusted_content";
 evidenceRef: string;
}
| {
 status: "refused" | "retryable_error" | "fatal_error";
 tool?: string;
 reason: string;
};

export type ToolContext = {
 route: Route;
 actorId: string;
 approvedActionIds: string[];
 timeoutMs: number;
 maxAttempts: number;
};

type ValidatedCall =
| {
 name: "read_order";
 args: { orderId: string };
 idempotencyKey: string;
}
| {
 name: "search_refund_policy";
 args: { query: string };
 idempotencyKey: string;
}
| {
 name: "draft_refund_request";
 args: { orderId: string; amountCents: number; approvalId: string };
 idempotencyKey: string;
};

export type ToolHandlers = {
 readOrder(args: { orderId: string }): Promise<unknown>;
 searchRefundPolicy(args: { query: string }): Promise<unknown>;
 draftRefundRequest(args: {
 orderId: string;
 amountCents: number;

```

```

 approvalId: string;
 }): Promise<unknown>;
};

const toolsByRoute: Record<Route, ToolName[]> = {
 refund_investigation: [
 "read_order",
 "search_refund_policy",
 "draft_refund_request",
],
 order_status: ["read_order"],
};

export function disclosedTools(route: Route): ToolName[] {
 return [...toolsByRoute[route]];
}

function objectArgs(value: unknown): Record<string, unknown> | null {
 return value && typeof value === "object"
 ? (value as Record<string, unknown>)
 : null;
}

function validateProposal(
 proposal: ToolProposal,
 context: ToolContext,
): ValidatedCall | ToolObservation {
 if (!toolsByRoute[context.route].includes(proposal.name as ToolName)) {
 return {
 status: "refused",
 tool: proposal.name,
 reason: "tool_not_disclosed_for_route",
 };
 }
};

```

*Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.*

**tool-using-agent-pattern/typescript/test/tool\_runtime.spec.ts**

[Open full source](#)

```

import {
 disclosedTools,
 ToolRuntime,

```

```

type ToolContext,
type ToolHandlers,
} from "../src/tool_runtime.ts";

function assert(condition: unknown, message: string): asserts condition {
 if (!condition) throw new Error(message);
}

const baseContext: ToolContext = {
 route: "refund_investigation",
 actorId: "support-agent",
 approvedActionIds: [],
 timeoutMs: 20,
 maxAttempts: 2,
};

function handlers(overrides: Partial<ToolHandlers> = {}): ToolHandlers {
 return {
 readOrder: async ({ orderId }) => ({ orderId, status: "delivered" }),
 searchRefundPolicy: async ({ query }) => ({ text: query }),
 draftRefundRequest: async args => args,
 ...overrides,
 };
}

assert(
 disclosedTools("order_status").join(",") === "read_order",
 "Route must disclose only required tools",
);

const runtime = new ToolRuntime(handlers());
const valid = await runtime.execute(
 {
 name: "read_order",
 args: { orderId: "ORD-104" },
 idempotencyKey: "read:104",
 },
 baseContext,
);
assert(valid.status === "ok", "Valid read must execute");

const forbidden = await runtime.execute(
 {
 name: "issue_refund",

```

```

 args: { orderId: "ORD-104" },
 idempotencyKey: "refund:104",
 },
 baseContext,
);
assert(
 forbidden.status === "refused" &&
 forbidden.reason === "tool_not_disclosed_for_route",
 "Undisclosed tool must be refused",
);

const invalid = await runtime.execute(
 {
 name: "read_order",
 args: { orderId: 104 },
 idempotencyKey: "read:invalid",
 },
 baseContext,
);
assert(
 invalid.status === "refused" && invalid.reason === "invalid_arguments",
 "Invalid arguments must be refused",
);

const missingApproval = await runtime.execute(
 {
 name: "draft_refund_request",
 args: {
 orderId: "ORD-104",
 amountCents: 12500,
 approvalId: "APR-104",
 },
 idempotencyKey: "draft:104",
 },
 baseContext,
);
assert(
 missingApproval.status === "refused" &&
 missingApproval.reason === "approval_required",
 "Write-like tool must require approval",
);

let draftedRefunds = 0;

```

*Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.*

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `tool-using-agent-pattern/` folder from this repository.

## Related Patterns

- [Single Agent](#)
- [Agent Loop](#)
- [Structured Output](#)
- [MCP-first Tool Use](#)
- [Tool Capability Design](#)
- [Human Approval Gates](#)
- [Pattern Evaluation Checklist](#)
- [Evaluation-Driven Agent Development](#)

# Structured Output

Structured output constrains model responses to typed data that software can validate and consume.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

The Structured Output Pattern constrains model responses to typed data that software can validate, route, store, and test. It is the boundary between natural language reasoning and deterministic application logic.

## Use When

- Model output controls a tool call, workflow branch, policy decision, or database write.
- Downstream code needs stable fields rather than prose.
- You need regression tests for model-assisted behavior.

## Avoid When

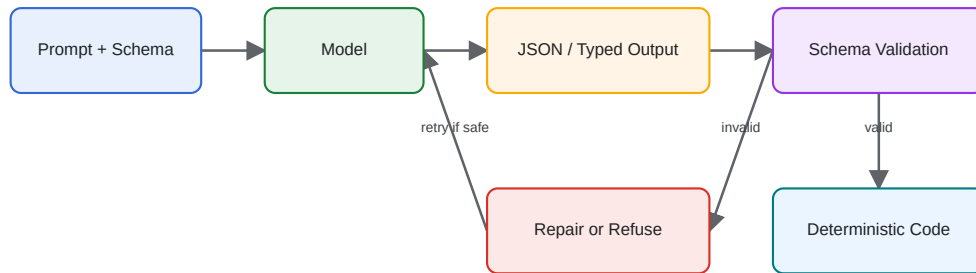
- The output is purely creative prose for human reading.
- A deterministic parser already handles the input safely.
- The schema is so broad that it no longer constrains behavior.

## Architecture

Use this diagram to read Structured Output as a system boundary, not only a code shape. The key ownership question is: the caller or a small application service owns task state until a runtime pattern is introduced.



## Structured Output Validation



## System Shape

- **Pattern boundary:** a narrow agent function, class, or service boundary accepts input plus context and returns a typed answer, action, or decision.
- **State owner:** the caller or a small application service owns task state until a runtime pattern is introduced.
- **Primary artifact:** `structured-output-pattern/` contains the runnable reference implementation and examples.
- **Operational promise:** Structured output constrains model responses to typed data that software can validate and consume.

## Core Protocol

1. Accept a bounded input, goal, or task request.
2. Assemble the minimum useful instructions, context, state, and tool descriptions.
3. Run the model or deterministic helper behind a typed boundary.
4. Validate the result before returning it to users, tools, or durable state.
5. Record enough evidence to explain the output later.

## Implementation Notes

- Define schemas close to the code that consumes them.
- Validate every model response before use, even when the provider offers structured output support.
- Prefer enums for routing decisions and discriminated unions for multi-action outputs.
- Log validation failures and repair attempts as first-class evaluation data.
- Keep the validated output close to the next runtime action. A valid object should still pass policy, approval, and state checks before it triggers side effects.

## Failure Modes

- Schemas that mirror prose and provide little safety.

- Silent coercion of missing or invalid fields.
- Prompt-only formatting rules with no validator.
- Overly strict schemas that cause brittle failures on harmless variation.
- A valid object carries unsupported values that violate domain or policy rules.
- Repair loops hide repeated model failures and increase cost without a stop condition.

## Evaluation Strategy

Evaluate syntax, semantics, and downstream safety separately. Schema validity proves that the output can be parsed. It does not prove that the values are correct or safe to execute.

- Test missing required fields, extra fields, invalid enums, wrong types, and malformed nested objects.
- Test values that pass schema validation but violate domain constraints, such as a refund above the order total.
- Test unsupported evidence references and contradictory fields.
- Test one repair attempt, repeated repair failure, and the final refusal or escalation path.
- Test schema-version changes against stored fixtures and downstream consumers.
- Assert that invalid output never reaches tools, policy decisions, or durable state.

Use deterministic validators for structure and domain invariants. Use human or model review only for fields that require judgment.

```
type StructuredOutputEvalCase = {
 caseId: string;
 modelOutput: unknown;
 expected: {
 schemaValid: boolean;
 domainValid: boolean;
 actionAllowed: boolean;
 repairAttempts: number;
 finalStatus: "accepted" | "repaired" | "refused" | "needs_human";
 };
};
```

Measure first-pass schema validity, domain-validity rate, repair success rate, repair attempts per accepted output, unsafe acceptance rate, false rejection rate, and schema-version compatibility.

For the shared eval case contract and release-gate method, see [Evaluation-Driven Agent Development](#).

## Production Checklist

- Define the input, context, output, and error contract.
- Keep prompts, schemas, and tool descriptions versioned.
- Add deterministic tests for the smallest useful behavior.
- Log model decisions without leaking secrets or private user data.

- Define human escalation for ambiguous, high-risk, or policy-blocked work.
- Keep the source bundle, generated chapter, tests, and deployment artifact in the same release.

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

`structured-output-pattern/structured_decision.ts`

[Open full source](#)

```
export type RefundDecision =
 | {
 kind: "draft_refund";
 orderId: string;
 amountCents: number;
 policyVersion: string;
 evidenceRefs: string[];
 }
 | {
 kind: "deny_refund";
 orderId: string;
 reason: string;
 policyVersion: string;
 evidenceRefs: string[];
 }
 | {
 kind: "needs_human_review";
 orderId: string;
 reason: string;
 missingEvidence: string[];
 };

export type ValidationResult =
 | { ok: true; decision: RefundDecision }
 | { ok: false; reason: string };
```

```

function isArray(value: unknown): value is string[] {
 return Array.isArray(value) && value.every(item => typeof item === "string");
}

export function validateRefundDecision(value: unknown): ValidationResult {
 if (!value || typeof value !== "object") {
 return { ok: false, reason: "decision_not_object" };
 }

 const record = value as Record<string, unknown>;
 if (typeof record.orderId !== "string") {
 return { ok: false, reason: "missing_order_id" };
 }

 if (record.kind === "draft_refund") {
 if (typeof record.amountCents !== "number" || record.amountCents <= 0) {
 return { ok: false, reason: "invalid_refund_amount" };
 }

 if (typeof record.policyVersion !== "string" || !isArray(record.evidenceRefs)) {
 return { ok: false, reason: "missing_policy_evidence" };
 }

 return { ok: true, decision: record as RefundDecision };
 }

 if (record.kind === "deny_refund") {
 if (
 typeof record.reason !== "string" ||
 typeof record.policyVersion !== "string" ||
 !isArray(record.evidenceRefs)
) {
 return { ok: false, reason: "invalid_denial_evidence" };
 }

 return { ok: true, decision: record as RefundDecision };
 }

 if (record.kind === "needs_human_review") {
 if (typeof record.reason !== "string" || !isArray(record.missingEvidence)) {
 return { ok: false, reason: "invalid_review_request" };
 }

 return { ok: true, decision: record as RefundDecision };
 }

```

```
}

return { ok: false, reason: "unknown_decision_kind" };
}

export function nextRuntimeAction(decision: RefundDecision): "draft" | "block" | "approval"
{
 if (decision.kind === "draft_refund") return "approval";
 if (decision.kind === "deny_refund") return "block";
 return "draft";
}
```

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `structured-output-pattern/` folder from this repository.

## Related Patterns

- [Modern Tool Use](#)
- [LLM Router](#)
- [Compliance/Policy Enforcer](#)
- [Evaluation-Driven Agent Development](#)

# Agent Development Lifecycle

Agent development is not prompt writing with deployment at the end. It is product engineering with a probabilistic component inside it.

That distinction matters. A prompt can make a demo work. A lifecycle makes the system survivable after the demo: when users ask unclear questions, tools fail, policies change, costs spike, and the model confidently proposes the wrong action.

Use this lifecycle when an agent will serve real users, call tools, touch private data, change external systems, or run for more than a demo session.

The lifecycle has one job: keep autonomy connected to evidence, state, policy, evaluation, and operations.

The first discipline is not to start with the agent. Start with the task, the state it needs, the tools it may use, the risk it creates, and the evidence that proves success. Add autonomy only where it earns its keep.

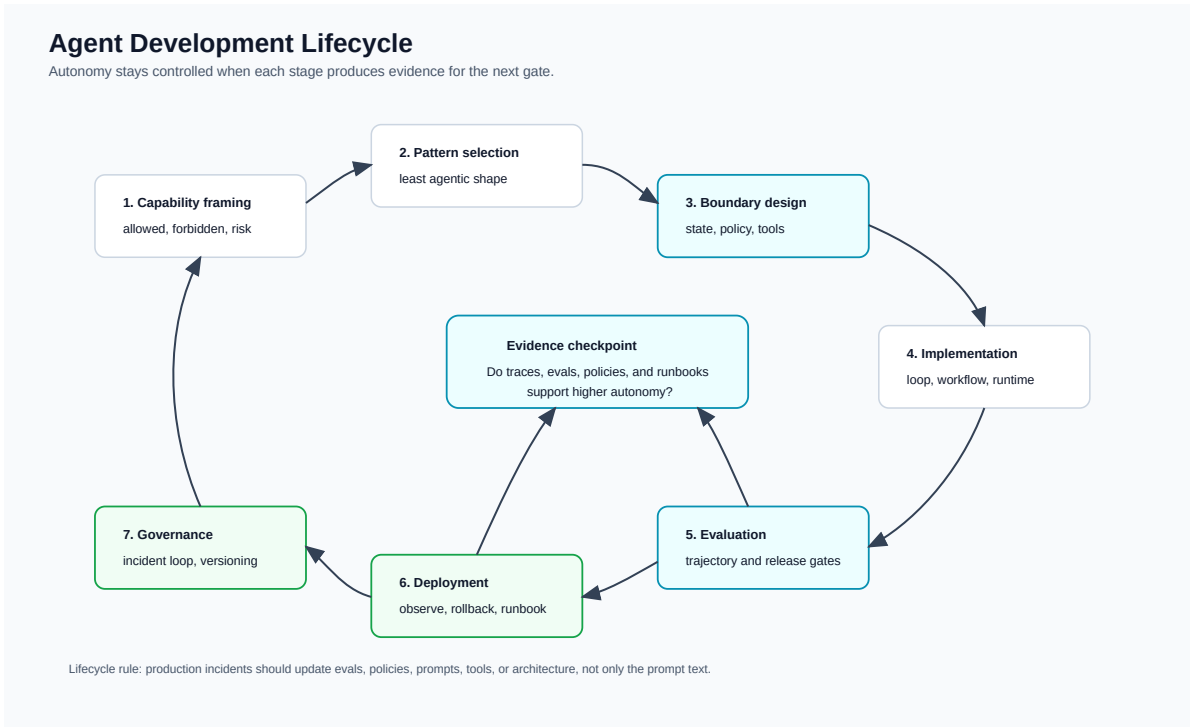
## What You Should Be Able To Do

After this chapter, you should be able to:

- move an agent from idea to production through explicit gates;
- name the artifacts required before prototype, pilot, production, and higher autonomy;
- separate prompt changes from architecture, policy, tool, memory, and eval changes;
- decide when production evidence justifies more autonomy;
- turn incidents into evals, policy updates, runbook changes, or architecture changes.

## Lifecycle Map

Use this diagram to read the lifecycle as an evidence loop, not a sequence of project phases. Each stage should produce artifacts that make the next gate inspectable.



## Lifecycle Stages

| Stage                 | Main Question                                              | Output                                                 |
|-----------------------|------------------------------------------------------------|--------------------------------------------------------|
| 1. Capability framing | What should the agent do, and what should it never do?     | Capability map, exclusions, risk class.                |
| 2. Pattern selection  | What is the least agentic architecture that works?         | Pattern choice and complexity budget.                  |
| 3. Boundary design    | Where do model decisions stop and software controls begin? | Tool contracts, policies, state model.                 |
| 4. Implementation     | How does the agent perceive, decide, act, and stop?        | Runnable agent, workflow, or multi-agent system.       |
| 5. Evaluation         | How do we know it works?                                   | Eval suite, failure modes, quality gates.              |
| 6. Deployment         | How does it run safely in production?                      | Observability, rollback, rate limits, approvals.       |
| 7. Governance         | How does it improve without losing control?                | Review process, audit logs, versioning, incident loop. |

The lifecycle is iterative. Production data should update evals, policies, prompts, tool design, and sometimes the architecture itself. If production failures only produce prompt tweaks, the team is probably treating symptoms.

## Development Gates

The lifecycle should have gates. Without gates, a demo becomes a pilot, a pilot becomes production, and production becomes autonomous without anyone making the decision explicitly.

| Gate                  | Must Be True                                                                                        |
|-----------------------|-----------------------------------------------------------------------------------------------------|
| Prototype ready       | Task is framed, success criteria exist, unsafe actions are excluded, and a baseline exists.         |
| Pilot ready           | Tool contracts, state model, eval fixtures, traces, and human fallback are in place.                |
| Production ready      | Security review, approval rules, observability, rollback, runbooks, and release gates are in place. |
| Higher autonomy ready | Production traces prove low override rate, evals cover known incidents, and rollback is tested.     |

Each gate should have an owner. If nobody owns the gate, the gate does not exist.

## Lifecycle Artifact Set

The lifecycle becomes real when each gate produces reviewable artifacts.

| Artifact                   | Created By                            | Used To Decide                                                                        |
|----------------------------|---------------------------------------|---------------------------------------------------------------------------------------|
| Capability map             | Product and engineering.              | What the agent may do, must not do, and should escalate.                              |
| Pattern decision           | Architecture owner.                   | Whether the system needs a prompt, workflow, loop, tools, memory, or multiple agents. |
| Authority map              | Security, platform, and feature team. | Which actions are read-only, draft-only, approval-gated, or forbidden.                |
| State and context contract | Runtime owner.                        | What survives retries, resumes, approvals, and handoffs.                              |
| Eval suite                 | Engineering and domain reviewers.     | Whether behavior is good enough to advance a gate.                                    |
| Trace schema               | Operations owner.                     | Whether failures can be reconstructed and replayed.                                   |
| Rollback plan              | Release owner.                        | How to disable a model route, prompt, tool, policy, or capability.                    |

If one artifact is missing, the gate should say so directly. A vague “looks good” review is not enough for systems that can act.



## Do Not Start With The Agent

Before building the agent loop, answer these questions:

- What is the user trying to accomplish?
- What would a deterministic workflow do?
- Which parts require judgment, retrieval, planning, or tool choice?
- What state must survive retries or approval waits?
- Which tools are read-only, write-capable, external, or high-risk?
- What evidence proves the result is correct?
- What is the correct behavior when evidence is missing?
- What must the system refuse to do?
- What will be evaluated before launch?
- How will the capability be disabled if it fails?

If those answers are vague, building an agent will mostly automate the ambiguity.

## Capability Framing

Start with a capability map, not a framework choice. A framework cannot tell you what the agent is allowed to do.

Capture the user jobs and business outcomes, the allowed and forbidden actions, the required evidence, the required tools and data, the approval points, the privacy and compliance constraints, the acceptable latency and cost, and the expected failure behavior. The important part is the negative space. An agent with a clear “will not do” list is easier to ship than an agent defined only by what it might do.

Good capability framing sounds concrete:

- “The agent can draft refund recommendations, but cannot issue refunds.”
- “The agent can query account history, but cannot access payment instruments.”
- “The agent can summarize an incident, but cannot page an on-call engineer without approval.”

If the team cannot write boundaries this plainly, the agent is not ready for implementation.

## Pattern Selection

Select the simplest pattern that handles the task. This is where many teams overbuild. A prompt handles a single bounded task. A prompt chain handles known phases. Routing handles inputs that need different paths. An agent loop handles the case where the next step depends on observations. Multi-agent systems handle separate roles that need separate context, tools, or permissions.

Do not start with a multi-agent architecture because the domain has many tasks. Start by grouping capabilities into stable workflows. Split agents only where role boundaries are real: different context, different tools, different permissions, different review responsibilities, or real parallel work.

The wrong pattern usually fails in one of two ways. It is too simple, so it cannot handle uncertainty. Or it is too agentic, so it becomes expensive, slow, and hard to debug. The lifecycle should catch both.

Use a deterministic baseline even when you expect to ship an agent. The baseline gives you something to compare against. If an agent cannot beat a form, a search page, a workflow, or a prompt chain on quality, coverage, or operational cost, the extra autonomy is not justified yet.

## Boundary Design

Before implementation, define the system boundaries. This is the architecture work that prevents the model from quietly becoming the whole application.

- What owns state?
- What owns policy?
- What owns tool permissions?
- What owns memory writes?
- What owns final approval?
- What can be replayed after failure?
- What can be changed without redeploying code?

The model may propose actions. Software validates, executes, logs, and enforces.

This boundary should be visible in code. Tool schemas should not be vague. State transitions should not be hidden in natural language. Approval rules should not live only in the system prompt. Memory writes should be reviewed or constrained before they become future context.

## Implementation

Implementation should make the lifecycle visible in code. A reader should be able to find the goal object, the state model, the tool boundary, the policy checks, the stop condition, and the trace emission without reverse-engineering a prompt.

Minimum implementation units:

- a goal or request object;
- typed state;
- tool schemas;
- policy checks;
- stop conditions;
- trace events;
- eval fixtures;
- deployment configuration.

Avoid implementations where the only durable artifact is a conversation transcript. Transcripts are useful evidence. They are not a state model.

The implementation should also make failure explicit. A failed tool call, a denied policy check, a budget stop, and a user cancellation are different outcomes. Do not collapse them into “the agent failed.”

## Release Readiness Checklist

Before an agent reaches production, the team should be able to point to concrete artifacts:

- owner and support path;
- capability map and exclusions;
- input, output, and error contract;
- autonomy level by action;
- tool manifests and capability classes;
- state schema and stop reasons;
- context assembly and memory policy;
- approval rules and escalation path;
- security and privacy review;
- eval suite with blocking cases;
- observability plan and trace schema;
- cost, latency, and tool budgets;
- rollback, kill switch, or capability disablement;
- ADRs for authority, memory, tools, and autonomy changes.

## Evaluation

Evaluate before launch and after launch. Agents change behavior when prompts, tools, models, memory, retrieval indexes, policies, or user populations change. Evaluation is not a launch checklist. It is the feedback loop for the system.

Pre-launch evaluation should cover happy-path tasks, ambiguous user requests, missing data, tool failure, retrieval failure, prompt-injection attempts, approval-required actions, and cost and latency budgets. Post-launch evaluation should add production failures, user corrections, human override cases, and edge cases found in traces. Every serious incident should leave behind an eval case.

Evaluation gates should be tied to releases. A model upgrade, prompt rewrite, tool manifest change, memory policy change, retrieval index change, or approval policy change should run the relevant regression suite before rollout.

Useful blocking evals include:

- forbidden tool calls;
- missing approval before side effects;
- unsafe memory writes;
- stale or missing evidence;
- tenant-boundary violations;
- failure to stop on budget or cancellation;
- hallucinated citations;

- poor recovery from tool failure.

## Deployment

Production deployment needs more than a hosted endpoint. It needs operational control: model and prompt versioning, rate limits and cost budgets, tool-level timeouts, audit logs, trace IDs that span tools and agents, alerting for breaker events, human escalation paths, and rollback for prompts, policies, and tools. If a deployment cannot explain what the agent did, it is not ready for high-impact work. The operator should be able to answer what goal was active, what evidence was used, what tools were called, what policies passed, what changed, and why the run stopped.

Rollout should be staged:

1. offline evals with mocked tools;
2. internal dogfood with read-only tools;
3. limited pilot with human approval on side effects;
4. canary production rollout with tight budgets;
5. broader rollout only after traces and overrides support it;
6. higher autonomy only after production evidence justifies it.

The point is not to slow the team down. The point is to make each increase in autonomy an explicit engineering decision.

## Governance

Governance is the control loop around the agent. It decides how the system changes without losing the boundaries that made it safe enough to ship. That means reviewing new tools, prompt and policy changes, memory schemas, eval changes, approval thresholds, incident reports, and model upgrades.

Treat agent changes like product and infrastructure changes. Small prompt edits can change behavior as much as code changes. New tools can change the risk class of the whole system. A model upgrade can invalidate old eval results. A memory schema change can change what the agent believes about returning users.

The lifecycle is complete only when these changes are reviewed, tested, deployed, observed, and fed back into the next iteration.

## Failure Modes

- Demo-first development becomes production without new controls.
- The team ships an agent without a deterministic baseline.
- Prompt changes ship without regression evals.
- Tools are added before capability boundaries are defined.
- Memory is enabled before write policy, deletion, and correction exist.
- The agent has no clear owner after launch.
- Human approval exists in the UI but not in the runtime policy.

- Production traces cannot reconstruct tool calls, context, memory, and decisions.
- Model upgrades happen without measuring behavior drift.
- Incidents lead only to prompt tweaks, not evals, policy changes, or architecture fixes.

## Lifecycle Review Questions

At each major change, ask:

- Did the autonomy level change?
- Did the tool authority change?
- Did the memory policy change?
- Did the context builder or retrieval index change?
- Did the model, prompt, routing, or fallback change?
- Did approval, budget, or stop behavior change?
- Did observability or redaction change?
- Which evals prove this is still acceptable?
- Which ADR or runbook should be updated?
- How do we roll back?

## Related Chapters

- [Choosing the Right Pattern](#)
- [Architecture Decision Records for Agents](#)
- [Evaluation-Driven Agent Development](#)
- [Goals and State](#)
- [Tool Capability Design](#)
- [MCP-first Tool Use](#)
- [Observability and Evals](#)
- [Policy Enforcement](#)
- [Agent UX and Human Trust](#)
- [Agentic System Architecture](#)

# Agent Harnesses

An agent harness is the layer around the loop. The loop calls the model, reads state, decides, acts, and stops. The harness gives that loop a usable working environment: tools, files, memory, skills, permissions, subagents, approvals, traces, and recovery.

This is the part many people mistake for the agent itself. It is not the agent. It is the operating shell that makes the agent useful for real work.

The framework may provide a harness. The runtime may host a harness. But the responsibility does not disappear: something has to decide what the agent can see, what it can do, what state survives, what needs approval, what is traced, and how the run recovers.

This chapter owns the agent's working environment. It does not own the model provider, business workflow, tool implementation, or fleet-wide production operations. The framework, application services, and production runtime own those concerns.

This chapter comes after the loop because the loop alone is too small for real work. Use it as the bridge from “what is the agent doing?” to “what environment keeps that work bounded?” The next production chapters take the same control ideas into runtime, observability, and security.

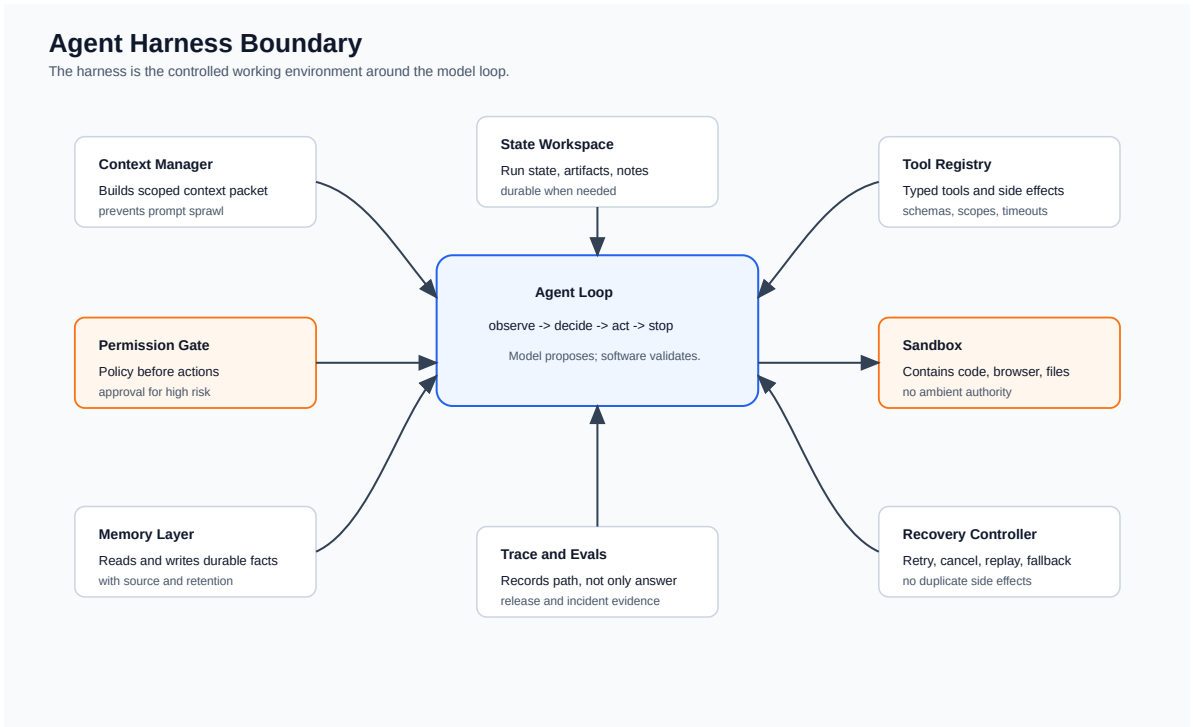
## What You Should Be Able To Do

After this chapter, you should be able to:

- distinguish the agent loop from the harness around it;
- name the controls a harness must own before the agent can do real work;
- decide which harness features are needed for a given capability;
- evaluate harness behavior separately from model answer quality;
- identify framework defaults that need product-owned policy.

## Harness Boundary

Use this diagram to separate the model loop from the environment that makes the loop safe and useful. The harness owns the controls around the loop: context, tools, policy, memory, traces, evals, sandboxing, and recovery.



## Why Harnesses Exist

A raw agent loop is small enough to fit in a few functions. Real tasks need more than that: a place to keep intermediate work, a way to load only the context that matters, a way to call tools safely, a way to split work across subagents, a way to pause for approval, a way to remember useful facts, a way to recover after failure, and a way to inspect what happened afterward. All of that together is the harness.

Coding agents make the point obvious. A coding agent does not only call a model. It reads files, edits files, runs tests, tracks tasks, invokes tools, asks for approval, handles failures, and keeps a trace of the session. The model matters, but the harness is what decides whether the work is controlled.

## Harness vs Framework vs Runtime

The terms overlap, but the distinction is worth keeping.

| Layer     | Main Job                                | Examples Of Responsibility                                                    |
|-----------|-----------------------------------------|-------------------------------------------------------------------------------|
| Framework | Gives abstractions for building agents. | Model adapters, tool calling, chains, agents, prompts.                        |
| Runtime   | Executes and operates agent work.       | Durable state, retries, streaming, deployment, persistence.                   |
| Harness   | Gives the agent a working environment.  | Files, skills, memory, subagents, permissions, approvals, context management. |

A product may use all three. A framework may include a runtime, and a runtime may include harness features. The names matter less than the responsibilities, and the question that cuts through the overlap is simple: which layer owns which control?

If the answer is “the framework handles it”, keep asking. Which component stores state? Which component enforces permission? Which component owns memory writes? Which component can stop a side effect? Which component emits the trace? Those are harness responsibilities whether the code lives in your application or a framework.

## Harness Responsibility Matrix

Use this matrix when reviewing a framework, SDK, or custom harness.

| Responsibility         | Product Question                         | Harness Evidence                                                    |
|------------------------|------------------------------------------|---------------------------------------------------------------------|
| Context control        | What did the model see for this call?    | Context packet, omitted refs, trust labels.                         |
| Tool disclosure        | Which tools were visible at this state?  | Tool profile by route, role, and approval state.                    |
| Permission enforcement | Why was an action allowed or denied?     | Policy decision with actor, target, mode, and reason.               |
| Memory access          | Why was memory read or written?          | Memory policy ref, source ref, retention, deletion path.            |
| Approval binding       | What exact action did the human approve? | Approval ID tied to tool, args, actor, and expiry.                  |
| Recovery               | Can this run resume or replay safely?    | Checkpoints, idempotency keys, mocked-tool replay mode.             |
| Observability          | Can an operator reconstruct the run?     | Trace events for proposal, validation, action, result, stop reason. |

If the harness cannot produce evidence for a row, that row is still a design risk, even if the demo works.

## Core Harness Capabilities

A capable harness usually exposes these surfaces.

| Capability      | What It Does                                                | Main Risk                                           |
|-----------------|-------------------------------------------------------------|-----------------------------------------------------|
| Workspace       | Gives the agent files, artifacts, notes, and scratch state. | Sensitive data leakage or stale artifacts.          |
| Context manager | Selects what enters the model context.                      | Token bloat, missing evidence, summarization drift. |
| Tool registry   | Exposes tools with schemas and descriptions.                | Broad tools, unsafe actions, weak authorization.    |



| Capability          | What It Does                                                   | Main Risk                                                |
|---------------------|----------------------------------------------------------------|----------------------------------------------------------|
| Skill loader        | Loads procedural instructions only when needed.                | Irrelevant activation or hidden unsafe scripts.          |
| Memory layer        | Reads and writes durable information.                          | Stale, private, or poisoned memory.                      |
| Planner             | Tracks tasks, subtasks, and progress.                          | Plans become theater instead of control.                 |
| Subagent manager    | Delegates work to isolated contexts.                           | Trace fragmentation and unclear ownership.               |
| Permission gate     | Blocks or pauses risky actions.                                | Prompt-only approval or approval fatigue.                |
| Sandbox             | Contains code, browser, shell, or file actions.                | Escape, secret exposure, or uncontrolled network access. |
| Trace surface       | Records decisions, tool calls, costs, and errors.              | Final-answer-only logs.                                  |
| Recovery controller | Handles retry, cancellation, replay, fallback, and escalation. | Duplicate side effects or lost progress.                 |

Not every agent needs every capability. A narrow support classifier may need almost none of this; a coding or research agent probably needs most of it.

## Harness Contract

A harness should have an explicit run context. This is the object, or set of objects, that prevents the agent loop from becoming a pile of globals.

```
type HarnessRunContext = {
 runId: string;
 actorId: string;
 tenantId: string;
 goalId: string;
 autonomyLevel: "advisory" | "drafts_for_review" | "executes_after_approval" |
"bounded_autonomous";
 stateRef: string;
 contextPacketId?: string;
 toolProfile: {
 allowedTools: string[];
 writeToolsRequireApproval: boolean;
 egressPolicyRef?: string;
 };
 memoryPolicyRef: string;
```

```
approvalPolicyRef: string;
budget: {
 maxIterations: number;
 maxCostCents: number;
 deadlineAt?: string;
};
traceId: string;
};
```

The exact shape can vary, but the harness should know the actor, tenant, goal, state, tools, memory policy, approval policy, budget, and trace before the model proposes the next action.

The same contract should make cancellation possible:

```
type HarnessStopReason =
 | "success"
 | "blocked"
 | "approval_required"
 | "budget_exhausted"
 | "cancelled"
 | "policy_denied"
 | "tool_failure";
```

If a harness cannot represent stop reasons, it will eventually turn different failures into the same vague error.

## Workspace And Artifacts

Longer tasks need a workspace, and it fills up quickly with user-provided files, retrieved documents, generated drafts, code patches, plans, task notes, tool outputs, and evaluation results. None of that should be treated as a dumping ground. Files and artifacts need ownership, naming, retention, and visibility rules like any other state.

Good workspace design answers a handful of questions up front: what the agent can read, what it can write, what the user can inspect, what is temporary, what becomes durable state, and what must never enter the model context at all. The workspace is part of the security model, not separate from it.

The workspace also needs cleanup rules. Temporary files, generated drafts, tool outputs, downloaded documents, and logs should not accumulate until they become accidental memory. Keep retention, redaction, and deletion explicit.

## Context Management

The harness decides what the model sees, and that decision is often more important than the prompt. A strong model with bad context still fails; a smaller model with clean context often works.

The context manager controls the instructions, the active goal, the current state, the relevant files, the retrieved evidence, the available tools, the selected memory, the recent observations, and the budget and stop rules. All of it should be loaded deliberately. Putting the whole workspace, the full conversation, every tool, all memory, and all retrieved documents into every call is not context engineering. It is context flooding. For the operating rules behind this, see [Context Budgets And Working Sets](#); a harness should be able to explain the working set it assembled for each model call.

A good harness can answer, for every model call: which context packet was assembled, which files and memories were included, which tools were disclosed, which sources were omitted, and why.

## Skills And Progressive Disclosure

Skills let a harness keep the base agent small. At startup the agent may see only the names and short descriptions of the available skills. When one becomes relevant, the harness loads the deeper instructions, references, scripts, templates, or examples behind it.

That is progressive disclosure, and it pays off across token budget, domain specialization, team-owned procedures, repeatable work, and safer tool use. Treat skills as versioned, tested artifacts rather than loose prompt fragments. A skill is procedural code and documentation that can change how the agent behaves, so it deserves the same discipline as any other code.

## Subagents And Context Isolation

Subagents are worth reaching for when a task benefits from isolated context: when work can happen in parallel, when a specialist needs different instructions or different tools, when a task should not pollute the main context, or when the main agent should receive a summary rather than every detail. Do not use them as decoration. Every subagent adds cost, latency, coordination, and trace complexity.

The harness has to make subagent work visible: who delegated the task, what context was passed, what tools were available, what result came back, what evidence supports it, and how the main agent used it. If subagent work cannot be traced, it cannot be trusted.

Subagents should inherit less authority by default, not more. The harness should pass a scoped goal, a scoped context packet, a scoped tool profile, and a trace relationship back to the parent run.

## Permissions And Approvals

The harness enforces permission boundaries before execution. The model can propose reading a file, calling an API, running a command, sending a message, updating a record, or writing a memory. The harness decides whether each of those is actually allowed.

Good permission systems draw distinctions the model cannot be trusted to draw for itself: read versus write, local versus remote, safe versus risky, reversible versus irreversible, user-visible versus hidden, cheap versus expensive, approved versus unapproved. Approval should be specific. “Allow tools” is too broad to mean anything. “Allow this agent to update ticket `INC-2048` with this summary” is something a person can actually review.

The harness policy can be expressed as a simple gate:

```
interface ToolRequest {
 tool: string;
 mode: 'read' | 'write';
 target: string;
 userVisible: boolean;
}

function authorizeHarnessAction(request: ToolRequest, permissions: HarnessPermissions) {
 if (!permissions.tools.includes(request.tool)) {
 return { allowed: false, reason: 'tool_not_in_profile' };
 }

 if (request.mode === 'write' && !permissions.canWrite) {
 return { allowed: false, reason: 'write_not_allowed' };
 }

 if (request.userVisible && !permissions.hasApproval) {
 return { allowed: false, reason: 'approval_required' };
 }

 return { allowed: true };
}
```

That is the harness doing its job: model intent becomes a checked request before anything happens.

The same pattern should apply to memory writes, file writes, shell commands, browser actions, external messages, and handoffs. The model proposes. The harness validates. The tool or workflow executes.

## Memory In A Harness

Harness memory is orchestration, not a new memory taxonomy. The harness decides when to read or write task-local state, durable memory, retrieved evidence, or procedural skills, but the policies live in the canonical layers: [Working Memory](#), [Memory-Augmented Agent](#), [Semantic Recall And RAG](#), and [Context Engineering](#).

The practical rule is simple: memory access is opt-in per task. Do not load global user memory, project memory, episodic memory, or retrieved documents just because they exist. Load them because they are relevant, allowed, fresh enough, and correctable. Write memory only through explicit events with source, scope, retention, and deletion rules.

## Sandboxes

Harnesses become security-critical the moment they expose code execution, shell commands, browser control, or filesystem writes. A sandbox has to control filesystem access, network access, environment variables, secrets, installed packages, process lifetime, clipboard and browser-profile access, and upload and download paths.

The right default is least privilege: give the agent the narrowest environment that can still complete the task, and widen it only when a specific need forces you to.

## Recovery And Replay

The harness owns what happens after interruption or failure. A serious harness can pause, cancel, retry, resume, replay, or escalate without losing track of state.

Recovery needs side-effect discipline:

- retries need idempotency keys;
- approvals need exact-action binding;
- file edits need diffs or snapshots;
- external messages need send records;
- memory writes need write IDs;
- tool calls need correlation IDs;
- replay should not repeat unsafe side effects.

A replay mode should be able to run with mocked tools, frozen context packets, and recorded model outputs when debugging. That is how teams compare prompt, model, policy, or tool changes without reissuing refunds, resending emails, or rewriting memory.

## Harness Observability For Background Agents

Background agents need a harness that can be inspected after the fact. At minimum, capture:

- the initial task and acceptance criteria;
- selected files, tools, and models;
- tool calls with inputs, outputs, policy decisions, and errors;
- checkpoints or handoff summaries after major phases;
- final verification commands and their results.

Without this record, a background agent becomes an opaque worker. With it, the agent becomes reviewable infrastructure.

## Harness Evaluation

Evaluate the harness directly. Freeze or mock the model response where possible so the test isolates harness behavior instead of model quality.

- Test context selection and exclusion.
- Test tool disclosure by route and role.

- Test permission denial before execution.
- Test approval waits and resume.
- Test cancellation before and during tool execution.
- Test retry with idempotency.
- Test memory write policy.
- Test subagent handoff traceability.
- Test sandbox boundaries.
- Test trace completeness after failure.
- Test replay without side effects.

Harness evals are different from answer evals. They ask whether the operating shell kept control when the model proposed something risky, incomplete, expensive, or wrong.

A harness fixture should describe the proposed trajectory and the controls that must hold:

```
type HarnessEvalCase = {
 caseId: string;
 proposedActions: Array<{
 tool: string;
 mode: "read" | "write";
 target: string;
 }>;
 expected: {
 disclosedTools: string[];
 deniedTools: string[];
 approvalRequiredFor: string[];
 maxIterations: number;
 maxCostCents: number;
 stopReason: HarnessStopReason;
 requiredTraceEvents: string[];
 repeatedSideEffects: 0;
 };
};
```

Keep a small set of blocking harness cases: forbidden tool disclosure, permission bypass, approval bypass, cross-tenant context, budget overrun, cancellation followed by a side effect, duplicate execution after retry, and replay that reaches a live write tool.

Measure tool-disclosure precision, permission-denial correctness, approval-bypass rate, budget-enforcement rate, cancellation latency, duplicate side effects, context-policy violations, trace completeness, replay isolation, and recovery success. Segment results by tool profile, autonomy level, and harness version.

For the shared eval case contract and release-gate method, see [Evaluation-Driven Agent Development](#).

## Harness Failure Modes

Harnesses fail in predictable ways. Every tool is available all the time. Every file enters context. Memory writes are invisible. Subagents create work nobody owns. Approvals are too broad. Sandbox boundaries are unclear. Traces omit tool inputs and outputs. Skills go stale but keep loading. Context compression removes the very evidence you need to debug the run. Every one of these is an architecture failure, not a model failure, which is exactly why a better model will not save you from them.

Add a few more to the list: cancellation does not stop queued side effects, retries duplicate work, subagents inherit broad permissions, framework defaults silently enable memory, workspace artifacts become hidden context, and replay is impossible because the harness did not store context packets or tool results.

## Design Checklist

Before adopting or building a harness, ask:

1. What does the harness own?
2. What state does it persist?
3. What can the model see?
4. What can the model propose?
5. What can the harness execute without approval?
6. What tools are scoped by role or task?
7. What files can be read or written?
8. What memory can be written?
9. What subagent work is traceable?
10. What happens when the run is interrupted?
11. What does the operator see after failure?
12. How can a harness change be evaluated before release?
13. How are context packets, tool calls, approvals, memory writes, and subagent runs linked in traces?
14. How does the harness replay a run without repeating side effects?
15. Which framework defaults have been replaced by product-owned policy?

If the harness cannot answer these, it is not ready to hold a serious agent.

## Production Checklist

- Define the harness run context.
- Keep goal, state, context, tools, memory, approvals, budget, and trace linked.
- Disclose tools progressively by route and state.
- Enforce permissions outside the prompt.
- Bind approvals to exact actions.
- Make cancellation and pause real runtime states.
- Keep memory writes explicit, scoped, reviewable, and deletable.
- Scope subagents by context, tools, and permissions.

- Sandbox high-risk actions with filesystem, network, secret, and process limits.
- Support replay with mocked tools and no repeated side effects.
- Evaluate harness behavior separately from final-answer quality.

## Design Rule

The model supplies judgment. The harness supplies control.

Once the harness contract is explicit, continue with [Production Runtime Overview](#). That chapter moves from one agent's working environment to queues, rollouts, budgets, recovery, and operators.

## Related Chapters

- [What Is An Agent?](#)
- [Agent Engineer Toolkit](#)
- [Framework Selection](#)
- [Tool Capability Design](#)
- [Context Engineering](#)
- [Context Budgets And Working Sets](#)
- [Skills](#)
- [Working Memory](#)
- [Human Approval Gates](#)
- [Observability and Evals](#)
- [Evaluation-Driven Agent Development](#)
- [Agent Security and Sandboxing](#)
- [Coding Agents](#)



# Building a Minimal Agent Runtime

This chapter explains the small runtime behind the from-scratch mini-framework labs. You do not build it because production teams should avoid mature frameworks. You build it because a tiny runtime makes the real architecture visible.

Frameworks change APIs and vocabulary. The same responsibilities keep coming back: state, decisions, tools, policy, context, traces, evals, and stop conditions. Once you can build those primitives in a small runtime, LangGraph, Mastra AI, AutoGen-style systems, CrewAI, MCP, A2A, and custom harnesses are easier to evaluate.

## What You Should Be Able To Do

After this chapter, you should be able to:

- identify the runtime primitives hidden inside agent frameworks;
- explain why the model proposes decisions but the runtime executes them;
- build a tiny loop with state, policy, tools, context, traces, and stop reasons;
- compare framework features by responsibility instead of vocabulary;
- know when the learning runtime should give way to production infrastructure.

## Why Build One

Most agent failures are not mysterious model failures. They are runtime failures.

The system did not know who owned state. The tool list was too broad. Policy lived in a prompt. The loop had no stop reason. Context was assembled by dumping everything into the model. A final answer looked acceptable, but the path used a forbidden tool. None of those are fixed by changing framework names.

A minimal runtime teaches the control boundary:

```
goal
-> build context
-> ask for a decision
-> validate the decision
-> check policy
-> execute allowed work
-> record observation
-> evaluate stop condition
```

That is the shape hidden under many framework abstractions.

Use this diagram to keep the runtime responsibilities separate. The model proposes a decision, but the runtime owns validation, policy, execution, observation, and stop conditions.

## What This Runtime Is Not

This is not a production framework. It does not try to solve deployment, streaming, distributed execution, persistence, authentication, workflow queues, model adapters, UI integration, tracing backends, or memory stores.

Use it as a learning scaffold. Use mature frameworks when you need production durability, operational integrations, concurrency, checkpoints, retries, hosted observability, and ecosystem support.

## Runtime Readiness Record

Before adapting the mini-runtime to a real system, write down which primitives are learning-only and which are production-owned.

```
runtime_readiness:
 purpose: "learning scaffold"
 state:
 owner: "mini-runtime"
 production_gap: "no durable checkpoints"
 tools:
 owner: "tool registry"
 production_gap: "no tenant-scoped authorization service"
 policy:
 owner: "local policy gate"
 production_gap: "no central audit trail"
 context:
 owner: "context builder"
 production_gap: "no retrieval freshness or redaction pipeline"
 trace:
 owner: "local trace events"
 production_gap: "no hosted observability backend"
 decision: "use for labs only; migrate to mature runtime before real side effects"
```

The record prevents the common mistake: treating a useful educational runtime as if it already has production durability, security, and operations.

## Primitive 1: State

State is the source of truth for the run. The transcript is not enough. A transcript says what was said; state says what the system is trying to do, what happened, what was observed, what remains, and why the run stopped.

```

type StopReason =
 | "success"
 | "blocked"
 | "approval_required"
 | "budget_exhausted"
 | "invalid_decision"
 | "policy_denied"
 | "tool_failure";

type Observation = {
 id: string;
 kind: "model" | "tool" | "policy" | "human" | "system";
 summary: string;
 data?: unknown;
};

type AgentState = {
 runId: string;
 goal: string;
 steps: number;
 maxSteps: number;
 observations: Observation[];
 stopReason?: StopReason;
};

```

Good state lets you resume, replay, debug, evaluate, and explain a run. Bad state forces operators to infer behavior from final text.

## Primitive 2: Decision

A model response is a proposal. The runtime turns that proposal into a typed decision before it can affect tools, users, durable state, or external systems.

```

type Decision =
 | { kind: "answer"; text: string }
 | { kind: "tool"; name: string; input: unknown }
 | { kind: "ask_human"; question: string }
 | { kind: "stop"; reason: StopReason };

```

This is the most important split in the runtime: the model can suggest, but software validates and executes.

## Primitive 3: Loop

The loop owns progress. It repeatedly builds context, asks for a decision, validates that decision, executes allowed work, records observations, and stops.

```
async function runAgent(
 state: AgentState,
 decide: (context: ContextPacket) => Promise<Decision>,
) : Promise<AgentState> {
 while (state.steps < state.maxSteps) {
 const context = buildContext(state);
 const decision = await decide(context);
 const result = await handleDecision(state, decision);

 state.observations.push(result.observation);
 state.steps += 1;

 if (result.stopReason) {
 state.stopReason = result.stopReason;
 return state;
 }
 }

 state.stopReason = "budget_exhausted";
 return state;
}
```

The loop should never run because the model keeps asking. It runs because the runtime still has budget, policy allows the next step, and stop conditions have not been met.

## Primitive 4: Tool Registry

Tools are capabilities. A registry defines the capabilities the runtime can expose.

```
type ToolResult =
 | { status: "ok"; data: unknown }
 | { status: "refused"; reason: string }
 | { status: "error"; reason: string };

type ToolDefinition = {
 name: string;
 description: string;
 sideEffect: "read" | "draft" | "write";
 execute(input: unknown): Promise<ToolResult>;
};
```

Keep tools narrow. Prefer `lookup_order_summary` over `run_sql`, `draft_refund_request` over `post_http`, and `search_policy_docs` over unrestricted browser or shell access.

## Primitive 5: Policy Gate

The registry says what exists. Policy decides what is allowed now.

```
type PolicyDecision =
 | { status: "allow" }
 | { status: "deny"; reason: string }
 | { status: "approval_required"; reason: string };

type PolicyContext = {
 actorId: string;
 route: string;
 approvedActionIds: string[];
 remainingSteps: number;
};
```

A useful policy gate considers actor, route, tenant, tool, side effect, approval state, data sensitivity, and budget. A prompt that says “do not do dangerous things” is not a policy gate.

## Primitive 6: Context Packet

Context is not everything the system knows. It is the working set for one decision.

```
type ContextPacket = {
 runId: string;
 goal: string;
 stateSummary: string;
 observations: Array<{ id: string; summary: string }>;
 toolsDisclosed: string[];
 evidenceRefs: string[];
 memoryRefs: string[];
 omittedRefs: Array<{ ref: string; reason: string }>;
};
```

The runtime should be able to explain why each item entered the context and why other available material stayed out.

## Primitive 7: Trace

Traces make behavior reviewable. Without them, debugging collapses into reading final answers and guessing.

```

type TraceEvent = {
 runId: string;
 step: number;
 type:
 | "context_built"
 | "decision"
 | "policy_decision"
 | "tool_result"
 | "stop";
 data: unknown;
};

```

Trace events should connect the model decision, policy result, tool call, observation, cost, latency, and stop reason.

### Primitive 8: Eval Harness

Agent evals should inspect paths, not only answers.

```

type EvalCase = {
 caseId: string;
 input: string;
 expected: {
 toolsCalled?: string[];
 toolsNotCalled?: string[];
 stopReason: StopReason;
 };
};

```

Useful evals catch forbidden tools, missing evidence, approval bypasses, invalid decisions, repeated side effects, and budget exhaustion. A plausible final answer is not enough if the trajectory was unsafe.

### How This Maps to Frameworks

| Runtime Primitive | LangGraph                    | Mastra AI                       | AutoGen-style Systems      | CrewAI      |
|-------------------|------------------------------|---------------------------------|----------------------------|-------------|
| State             | graph state and checkpoints  | workflow and memory state       | conversation/session state | flow state  |
| Decision          | node output or router result | agent response or workflow step | agent message              | task output |

| Runtime Primitive | LangGraph                      | Mastra AI                | AutoGen-style Systems       | CrewAI                        |
|-------------------|--------------------------------|--------------------------|-----------------------------|-------------------------------|
| Loop              | graph traversal                | workflow/agent runtime   | conversation turn loop      | flow execution                |
| Tool registry     | tools bound to nodes or agents | tools                    | callable functions/tools    | role tools                    |
| Policy gate       | guard node or middleware       | workflow/tool policy     | manager or wrapper          | flow guard or task constraint |
| Context packet    | node input state               | agent context and memory | message set                 | task context                  |
| Trace             | callbacks and checkpoints      | observability/evals      | logs and messages           | task and flow logs            |
| Eval harness      | graph-level tests              | eval suites              | transcript/trajectory tests | task/flow quality checks      |

Frameworks can package these primitives, but they do not remove the need to design them.

The important comparison is responsibility, not API shape:

| Question                         | If You Build It Yourself                                                      | If You Use A Framework                                                                                            |
|----------------------------------|-------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| Who owns state?                  | Your runtime data model and persistence plan.                                 | The framework may provide state containers or checkpoints, but your application still defines the business state. |
| Who authorizes tools?            | Your policy function, approval records, and audit trail.                      | The framework can expose hooks or middleware, but product policy still belongs outside the prompt.                |
| Who assembles context?           | Your context builder chooses memory, evidence, tools, and omissions.          | The framework can provide memory abstractions, but you still need source, freshness, and privacy rules.           |
| Who evaluates behavior?          | Your tests inspect decisions, tools, traces, and stop reasons.                | The framework can run evals, but you still decide what unsafe or low-quality behavior means.                      |
| Who handles production failures? | You must add retries, idempotency, durability, alerts, and incident workflow. | Mature runtimes can provide pieces of this, but they must be configured against your risk model.                  |

## What the Labs Do

The mini-framework labs implement the primitives in three passes:

1. [Lab 09 - Minimal Agent Loop](#) builds state, decisions, loop control, and stop reasons.
2. [Lab 10 - Tool Registry and Policy Gate](#) adds tools, policy decisions, approval-required outcomes, and refusal paths.
3. [Lab 11 - Context, Memory, Trace, and Evals](#) adds context packets, scoped memory, trace events, and trajectory evals.

Do the labs if you want implementation intuition. Read this chapter alone if you only need the mental model.

## Design Rule

Build the tiny runtime to learn. Ship with mature runtime capabilities when the system must survive real users, real data, real side effects, and real incidents.

## Related Chapters

- [What Is An Agent?](#)
- [Agent Loop](#)
- [Agent Harnesses](#)
- [Tool Capability Design](#)
- [Context Engineering](#)
- [Observability and Evals](#)



# Agent Engineer Toolkit

An agent engineer chooses models, tools, memory, orchestration, evals, and deployment infrastructure as one system. The toolkit should support the architecture, not define it.

Use this chapter to decide what to build directly and what to take from a framework.

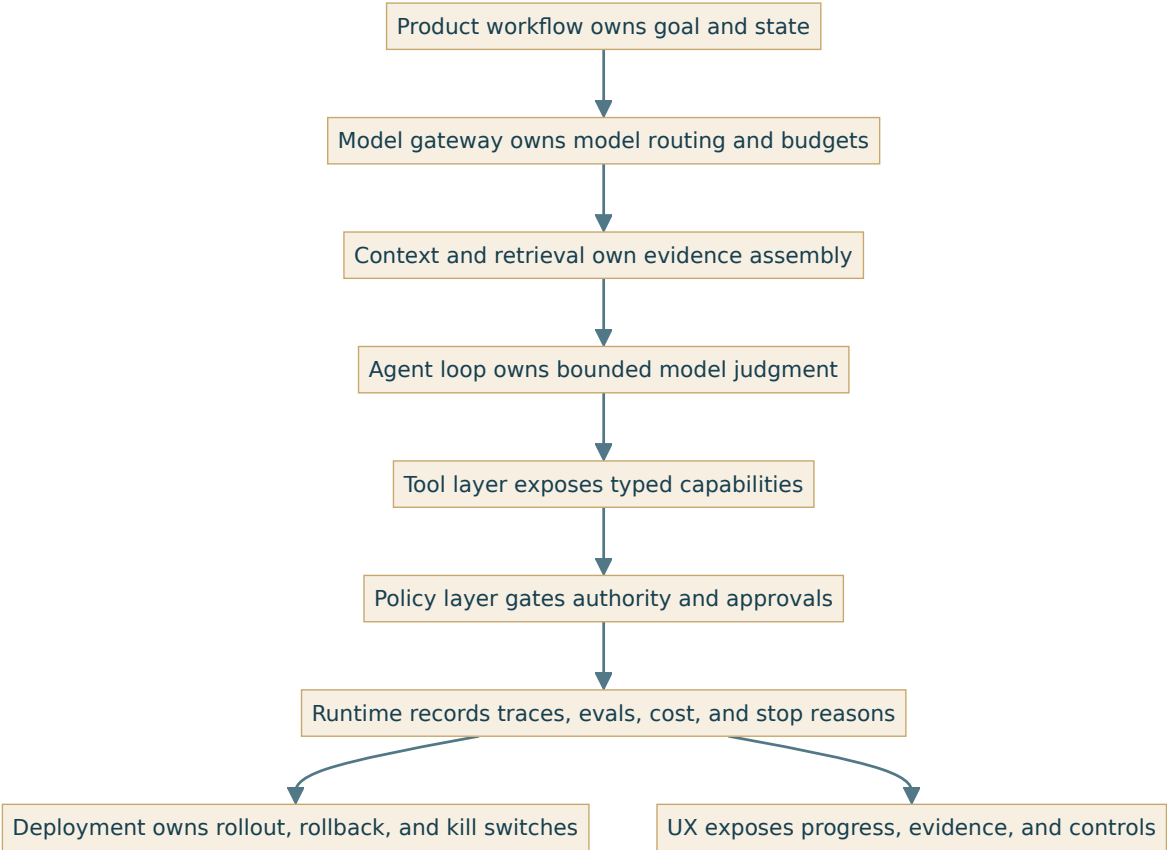
Think of the toolkit as a capability map. Some capabilities can come from a framework, some from existing platform services, and some should stay product-owned because they define authority, policy, and trust.

## Toolkit Layers

| Layer               | Responsibility                                                  | Examples of Decisions                                                          |
|---------------------|-----------------------------------------------------------------|--------------------------------------------------------------------------------|
| Model layer         | Reasoning, generation, tool calling, structured output.         | Model quality, latency, cost, context window, deployment mode.                 |
| Model gateway       | Routing, fallback, rate limits, provider abstraction.           | Which models are allowed, when fallback happens, how versions are pinned.      |
| Context layer       | Assembles the model working set.                                | Context packet shape, source labels, exclusions, token budget.                 |
| Orchestration layer | Workflow, routing, loops, state, retries.                       | Code workflow, LangGraph-style graph, CrewAI-style crew, Mastra-style runtime. |
| Tool layer          | External actions and data access.                               | MCP servers, internal APIs, browser tools, code execution, database functions. |
| Policy layer        | Runtime allow, deny, approval, escalation, and audit decisions. | Tool authority, data access, memory writes, approval thresholds.               |
| Memory layer        | Working, episodic, semantic, and user memory.                   | Vector index, relational store, filesystem state, memory write policy.         |
| Retrieval layer     | Source search, filtering, reranking, citation.                  | Index type, source registry, freshness rules, tenant filters.                  |
| Evaluation layer    | Quality and regression checks.                                  | Golden datasets, judges, human labels, production traces.                      |
| Observability layer | Debugging and operations.                                       | Traces, metrics, costs, logs, run replay.                                      |
| Deployment layer    | Runtime hosting, scaling, config, rollback.                     | Feature flags, canaries, kill switches, environment policy.                    |

| Layer            | Responsibility               | Examples of Decisions                                     |
|------------------|------------------------------|-----------------------------------------------------------|
| Security layer   | Permissions and containment. | Sandboxes, secrets, policy gates, audit, egress controls. |
| UX layer         | Human interaction and trust. | Streaming, approvals, explanations, handoffs, recovery.   |
| Governance layer | How the system changes.      | ADRs, reviews, release gates, incident-to-eval loop.      |

The layers can come from one framework, several libraries, or direct code. The important part is that each layer has an owner.



Use the map as an ownership check. If a toolkit hides one of these boundaries, the team still owns the missing decision; it has only moved out of sight.

### Build, Buy, Or Compose

Do not ask “which framework should we use?” first. Ask which toolkit capabilities must be product-owned.

| Capability     | Often Framework-Provided                                | Should Stay Product-Owned                      |
|----------------|---------------------------------------------------------|------------------------------------------------|
| Model adapters | provider clients, streaming, structured output helpers. | allowed models, fallback policy, cost budgets. |

| Capability    | Often Framework-Provided                            | Should Stay Product-Owned                                     |
|---------------|-----------------------------------------------------|---------------------------------------------------------------|
| Orchestration | graph execution, workflow steps, retries.           | state schema, stop reasons, approval waits, rollback.         |
| Tooling       | tool registration, invocation helpers, MCP clients. | tool authority, side effects, scopes, credentials, approvals. |
| Memory        | stores, embeddings, retrieval helpers.              | write policy, consent, retention, deletion, correction.       |
| Retrieval     | indexes, rerankers, connectors.                     | source registry, access filters, freshness, citation rules.   |
| Evals         | runners, judges, reports.                           | blocking cases, incident fixtures, release gates.             |
| Observability | spans, logs, dashboards.                            | trace schema, redaction, audit requirements, replay policy.   |
| Deployment    | hosting, configs, scaling.                          | rollout plan, kill switches, incident ownership.              |

Frameworks are good at mechanics. Product teams must still own authority.

## Minimum Toolkit By Risk

The toolkit should scale with risk. A demo and a side-effectful production agent do not need the same machinery.

| Risk Level                      | Minimum Toolkit                                                                                         |
|---------------------------------|---------------------------------------------------------------------------------------------------------|
| Demo                            | model call, prompt, sample inputs, simple run log.                                                      |
| Internal read-only assistant    | context builder, source filters, basic evals, trace IDs, feedback path.                                 |
| Production read-only agent      | state model, retrieval policy, eval gates, observability, cost budgets, rollout controls.               |
| Production side-effectful agent | typed tools, policy engine, approval gates, idempotency, rollback, incident replay.                     |
| Regulated or high-risk agent    | security review, audit logs, redaction, tenant isolation, human escalation, ADRs, formal release gates. |

Adding toolkit layers is not ceremony when the agent can affect users, money, access, data, or production systems.

## Build vs Framework

Use a framework when it removes real operational complexity:

- graph execution;
- durable state;
- tracing;
- tool registration;
- eval integration;
- human approval flows;
- deployment management;
- multi-agent orchestration.

Build directly when:

- the workflow is small;
- the team needs full control over state and policy;
- framework abstractions hide important failures;
- the system must integrate with existing infrastructure;
- the agent is part of a larger product workflow.

Frameworks should make boundaries clearer. If a framework makes it hard to see state, tool calls, permissions, or failure modes, use a smaller abstraction.

## Product-Owned Interfaces

Keep these interfaces explicit even when a framework executes them:

- request and response contracts;
- state and event schemas;
- tool manifests;
- context packet shape;
- policy decision schema;
- memory record schema;
- approval request and decision schema;
- eval fixture format;
- trace schema;
- deployment and rollback controls.

These are the pieces that let a team migrate frameworks, debug incidents, and keep authority out of prompts.

## Framework Evaluation Checklist

Before adopting a framework, test it against the agent's production requirements.

- Can it persist and resume state?

- Can it represent approval gates?
- Can it restrict tools per role or route?
- Can it trace model calls, tool calls, and handoffs?
- Can it run evals against realistic fixtures?
- Can it support model fallback and routing?
- Can it expose cost and latency by step?
- Can it run in the team's deployment environment?
- Can engineers debug failures without reading framework internals?
- Can it be removed later if the product outgrows it?

Do a thin vertical slice before committing the whole architecture.

## Model Selection

Choose models by workload, not benchmark rank alone.

Evaluate:

- structured output reliability;
- tool-use reliability;
- instruction following;
- latency;
- cost per completed task;
- context window behavior;
- refusal and safety behavior;
- multimodal needs;
- support for deployment constraints;
- provider stability and versioning.

Use multiple models when it reduces cost or improves quality. A small model may route, classify, or extract. A stronger model may plan, synthesize, or resolve edge cases.

## Tooling Decisions

Tools need engineering discipline.

At toolkit-selection time, look for the minimum surfaces that make tools governable: names, descriptions, typed inputs and outputs, permission scopes, timeouts, idempotency, side-effect labels, and test fixtures. The full design checklist lives in [Tool Capability Design](#); this chapter only asks whether your toolkit can support it.

Poor tools expose broad surfaces such as unrestricted shell, generic browser control, or “database query” without policy checks. Broad tools can be useful for trusted coding agents, but they need sandboxing and audit.

The tool layer should be boring on purpose. Tools should look like product capabilities, not raw infrastructure. Prefer `create_refund_draft` over `run_sql`, `send_approved_message` over `post_http`, and `lookup_order_summary` over broad database access.

## Memory Decisions

Memory is not one feature. It is several stores with different rules.

- Working memory holds the current task state.
- Episodic memory stores previous runs and user interactions.
- Semantic memory stores facts and documents.
- Procedural memory stores skills, playbooks, and preferences.

Decide who can write memory, how writes are validated, how old memory expires, and how retrieval is cited.

## Evaluation And Observability Decisions

Evals and traces are toolkit layers, not finishing touches.

At minimum, the toolkit should support:

- offline evals with mocked tools;
- trajectory evals that inspect tool calls and policy decisions;
- regression evals from incidents;
- model, prompt, tool, policy, and memory version tracking;
- traces across model calls, tool calls, retrieval, approvals, and memory writes;
- redaction rules for logs and eval fixtures;
- replay without repeating unsafe side effects.

If the toolkit cannot prove what happened, the team will debug the final answer instead of the system.

## Toolkit Assembly Checklist

Before choosing libraries or frameworks, fill this in:

| Question                             | Answer Needed                                         |
|--------------------------------------|-------------------------------------------------------|
| What owns the active goal and state? | workflow, runtime, app service, or framework.         |
| What owns tool authority?            | tool registry, policy engine, access-control service. |
| What owns memory writes?             | memory service, policy gate, human review.            |
| What owns context assembly?          | context builder with traceable packets.               |
| What owns eval gates?                | CI, release process, eval service.                    |

| Question                 | Answer Needed                                       |
|--------------------------|-----------------------------------------------------|
| What owns observability? | tracing platform, app logs, audit store.            |
| What owns rollback?      | deployment system, feature flags, tool disablement. |
| What owns user controls? | product UI, approval service, support workflow.     |

Unowned layers become production incidents.

## Toolkit Anti-Patterns

- Choosing a framework before choosing a pattern.
- Giving every agent every tool.
- Treating vector search as memory without write policy.
- Logging final answers but not tool decisions.
- Using model routing without evals for route quality.
- Shipping a demo framework path without production state, policy, and observability.
- Treating framework-provided memory as safe by default.
- Letting policy live only in prompt text.
- Adding observability after the first production incident.
- Building a tool layer with no manifest, owner, or disable path.
- Choosing a model gateway without version pinning or fallback rules.

## Related Chapters

- [Framework Selection](#)
- [Agent Harnesses](#)
- [Agent Development Lifecycle](#)
- [Choosing the Right Pattern](#)
- [Tool Capability Design](#)
- [Context Engineering](#)
- [Policy Enforcement](#)
- [Observability and Evals](#)
- [Architecture Decision Records for Agents](#)
- [MCP-first Tool Use](#)
- [Working Memory](#)
- [Mastra Runtime](#)
- [CrewAI Flows and Crews](#)
- [Agentic System Architecture](#)

# Framework Selection

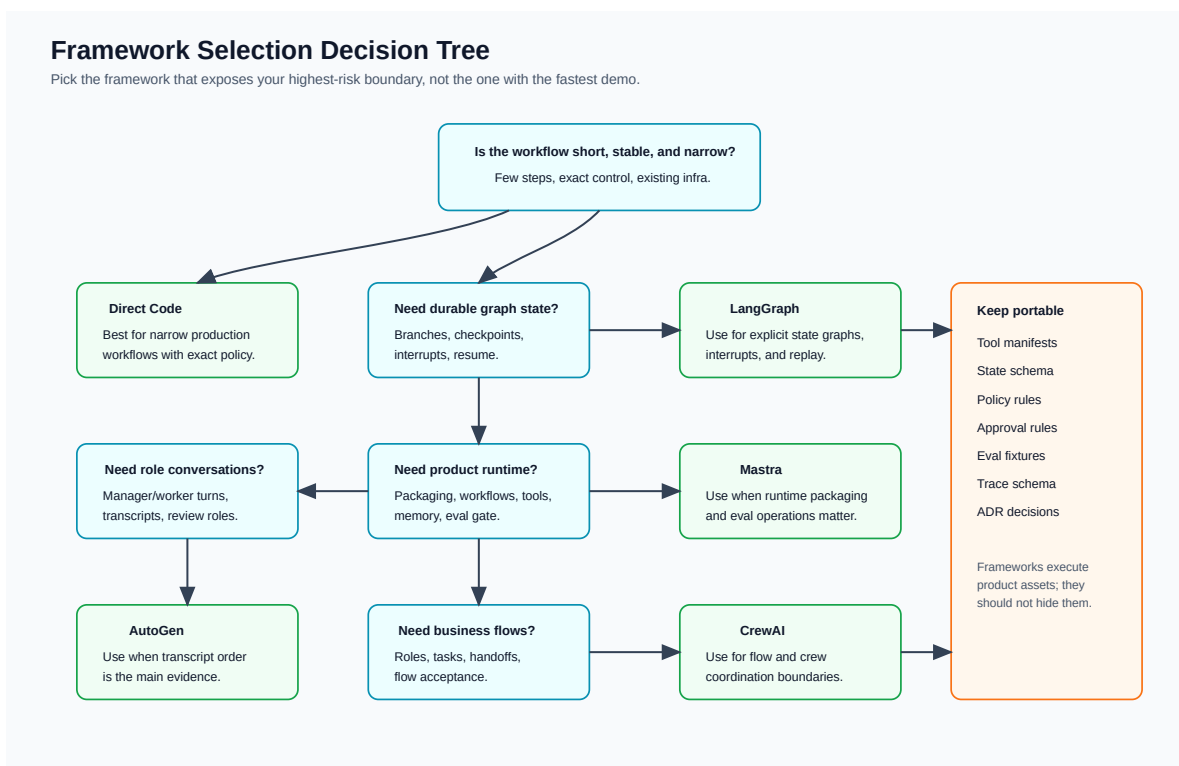
Agent frameworks can accelerate development, but they also shape the system's boundaries. Select a framework after you know the pattern, state model, tools, eval needs, and deployment constraints.

Use this chapter to compare frameworks without turning the architecture into a framework demo.

For a side-by-side comparison of LangGraph, AutoGen-style systems, Mastra AI, CrewAI, and a custom mini-runtime, use [Cross-Framework Decision Matrix](#). For concrete install and porting notes, use [Real Framework Setup Notes](#). For the decision record, use [Templates and Worksheets](#).

## Decision Tree

Use this diagram to keep the framework conversation grounded in workload shape. The useful question is what the framework makes easier to inspect: direct state, graph execution, runtime packaging, role transcripts, or business-flow coordination.



## Intent

Choose whether to build directly, use a lightweight library, or adopt a full agent framework.

The right framework should make the important parts more visible: state, tools, policy, traces, handoffs, and failure modes.

The wrong framework makes the demo faster and the system harder to own. It hides state, turns tool calls into magic, makes memory feel automatic, and gives the team a vocabulary before it has an architecture.



## Selection Criteria

| Criterion     | Questions                                                                        |
|---------------|----------------------------------------------------------------------------------|
| Control flow  | Does it support chains, graphs, loops, routing, and human gates?                 |
| State         | Can it persist, resume, inspect, and migrate state?                              |
| Tooling       | Can tools be typed, scoped, tested, and audited?                                 |
| Multi-agent   | Can it represent roles, handoffs, and ownership clearly?                         |
| Memory        | Does it separate working, episodic, semantic, and procedural memory?             |
| Evaluation    | Can evals run against routes, tools, traces, and final outputs?                  |
| Observability | Does it expose model calls, tool calls, costs, latency, and errors?              |
| Security      | Can tools and agents run with least privilege?                                   |
| Deployment    | Can it run where your system must run?                                           |
| Escape hatch  | Can you drop to code when the abstraction is wrong?                              |
| Portability   | Can prompts, tools, evals, policies, and state survive a framework change?       |
| Ownership     | Can your team debug production failures without waiting for framework internals? |

Do not adopt a framework that hides a concern your product must control.

## Fit By Problem Shape

Start from the workload, not the framework category.

| Problem Shape          | Better Fit                           | Watch For                                     |
|------------------------|--------------------------------------|-----------------------------------------------|
| Single bounded task    | direct code or prompt library.       | overbuilding state and orchestration.         |
| Known phases           | prompt chain or workflow.            | hiding validation between steps.              |
| Branching workflow     | graph or durable workflow framework. | graph sprawl and unclear state ownership.     |
| Tool-heavy agent       | tool registry plus policy boundary.  | broad tools and weak approval controls.       |
| Evidence-heavy answers | RAG framework plus context builder.  | retrieval disconnected from policy and evals. |

| Problem Shape             | Better Fit                                                         | Watch For                                    |
|---------------------------|--------------------------------------------------------------------|----------------------------------------------|
| Long-running work         | durable workflow runtime.                                          | in-memory loops and lost approval state.     |
| Multi-role work           | multi-agent framework or service topology.                         | fake specialization with duplicated context. |
| Cross-agent communication | A2A, service contracts, queues, or workflows.                      | natural-language handoffs without schema.    |
| Product runtime           | application/runtime framework.                                     | platform lock-in and opaque operations.      |
| Coding agents             | sandboxed harness with file, command, test, and approval controls. | shell access without policy or replay.       |

If two approaches both work, choose the one that makes state, tools, policy, evals, and rollback easier to inspect.

## Common Framework Shapes

| Shape                    | Strength                                                 | Risk                                                           |
|--------------------------|----------------------------------------------------------|----------------------------------------------------------------|
| Prompt and tool library  | Fast for simple agents.                                  | You must build state, evals, and operations yourself.          |
| Graph/workflow framework | Clear execution paths, durable state, conditional edges. | Graphs can become hard to read if every branch becomes a node. |
| Multi-agent framework    | Role and delegation primitives.                          | Easy to overuse agents where a workflow is enough.             |
| RAG framework            | Retrieval, indexes, and data connectors.                 | Retrieval can become disconnected from policy and evals.       |
| Runtime framework        | Deployment, tracing, memory, workflows, and operations.  | Runtime choices can become platform lock-in.                   |

The best choice may combine a small number of layers instead of one large framework.

## Capability Matrix

Use this matrix during evaluation. The goal is not to assign a score for marketing features. The goal is to identify which responsibilities the framework owns and which ones your application must still own.

| Capability     | Questions To Ask                                                                       |
|----------------|----------------------------------------------------------------------------------------|
| State model    | Where is state stored? Can it be inspected, migrated, replayed, and versioned?         |
| Tool boundary  | Are tools typed? Are scopes, side effects, approvals, and trace fields explicit?       |
| Policy         | Can policy run before tool calls, memory writes, retrieval, and final answers?         |
| Memory         | Are working, episodic, semantic, and user memory separated? Can writes be reviewed?    |
| Context        | Can the framework show what context packet was assembled for each model call?          |
| Human approval | Are approvals durable, exact-action bound, expiring, and auditable?                    |
| Evals          | Can evals test trajectories, tool calls, refusals, memory writes, and citations?       |
| Observability  | Are model calls, tool calls, costs, latency, policy decisions, and retries traced?     |
| Security       | Can it enforce least privilege, sandboxing, credential scope, and egress limits?       |
| Deployment     | Does it support your runtime, data residency, scaling, rollback, and incident process? |
| Interop        | Can it expose or consume MCP, A2A, REST, queues, or your existing service contracts?   |
| Exit path      | Can you move prompts, tools, state, policies, and evals out later?                     |

## Build Directly When

- The workflow is short and stable.
- You need exact control over state and policy.
- The team already has workflow infrastructure.
- The agent is embedded inside an existing product.
- Debuggability matters more than rapid prototyping.
- The framework would introduce more concepts than the domain requires.

Direct code is often the best first production version for a narrow workflow.

Build directly does not mean build carelessly. It means using normal software boundaries: typed inputs, state tables, tool clients, policy checks, tests, traces, and deployment controls. A small direct implementation with strong boundaries is better than a framework implementation that hides them.

## Use a Framework When

- The system needs durable graph execution.
- Agents need tool registration and discovery.
- Multi-agent handoffs are central to the product.

- You need built-in tracing, evals, or deployment support.
- You will build many related agents.
- The team benefits from shared conventions.

Use a framework to make repeated patterns consistent, not to avoid understanding the architecture.

Framework adoption is strongest when the team already knows the responsibilities it needs the framework to carry. It is weakest when the team says, “we need agents”, and lets the framework define the architecture after the fact.

## Evaluation Process

Run a framework through a vertical slice:

1. one real user request;
2. one route or workflow;
3. one read tool;
4. one write tool behind approval;
5. one memory read or write;
6. one failure path;
7. one eval case;
8. one trace inspection;
9. one deployment path.

If the slice is hard to inspect, hard to test, or hard to secure, stop before the framework spreads.

Add a second vertical slice for failure:

1. malformed user request;
2. missing evidence;
3. denied policy check;
4. tool timeout;
5. approval wait;
6. cancellation;
7. replay from trace;
8. rollback of prompt, model, or tool configuration.

Most frameworks look good on the happy path. The failure slice tells you whether you can operate it.

## Portability And Exit

Assume the framework may change. The safest way to adopt one is to keep the core assets portable:

- tool manifests;
- prompt and instruction files;
- state schemas;
- memory schemas;

- approval rules;
- policy rules;
- eval fixtures;
- trace schema;
- context packet shape;
- architecture decision records.

Avoid burying these assets inside framework-specific callbacks or decorators if they represent product policy. The framework can execute them. It should not be the only place where they exist.

## Framework Anti-Patterns

- Selecting a framework because it supports every pattern.
- Replacing architecture decisions with framework defaults.
- Giving all framework agents the same context and tools.
- Accepting opaque traces.
- Skipping evals because the framework demo worked.
- Treating framework memory as trustworthy by default.
- Using a multi-agent framework for simple routing.
- Treating built-in observability as sufficient without incident replay.
- Accepting framework tool abstractions that cannot express side effects or approvals.
- Letting framework defaults decide memory retention, context assembly, or security scope.

## Decision Checklist

Before adopting a framework, answer:

- Which pattern are we implementing?
- Which responsibilities does the framework own?
- Which responsibilities remain in our application?
- Can we inspect state, context, tool calls, memory, policy decisions, and traces?
- Can we run evals without calling real side-effectful tools?
- Can we enforce approval and least privilege outside the prompt?
- Can we deploy, roll back, and disable one capability quickly?
- Can we migrate away without rewriting prompts, tools, state, evals, and policy?
- Which ADR records this choice?
- What failure would make us reconsider the framework?

## Related Chapters

- [Agent Engineer Toolkit](#)
- [Cross-Framework Decision Matrix](#)
- [Real Framework Setup Notes](#)
- [Templates and Worksheets](#)
- [Agent Harnesses](#)

- [Agent Development Lifecycle](#)
- [Architecture Decision Records for Agents](#)
- [Choosing the Right Pattern](#)
- [Tool Capability Design](#)
- [Policy Enforcement](#)
- [Observability and Evals](#)
- [Mastra Runtime](#)
- [CrewAI Flows and Crews](#)
- [Agentic System Architecture](#)

# Cross-Framework Decision Matrix

This chapter compares LangGraph, AutoGen-style systems, Mastra AI, CrewAI, and a small custom runtime by engineering responsibility. The goal is not to crown one framework. The goal is to decide what should own state, tools, policy, memory, evals, deployment, and failure recovery for a specific system.

Use this chapter after the labs. The labs show each framework shape in isolation. This chapter helps choose among them for a real product. Pair it with [Real Framework Setup Notes](#) when you need concrete install and porting guidance.

Use this diagram to locate the state owner before comparing APIs. A framework is a better fit when its state boundary matches the failure you most need to inspect and recover.



## Decision Rule

Choose the framework that makes your highest-risk boundary easiest to inspect, test, and operate.

If the highest risk is resumable state, prefer graph or workflow state. If the highest risk is multi-agent accountability, prefer a transcript or flow that records role behavior. If the highest risk is production runtime packaging, prefer a runtime with conventions for tools, memory, evals, and observability. If the highest risk is exact policy control, a small direct runtime may be safer than a large abstraction.

## Framework Fit Matrix

| Option                      | Best When                                                                                              | Avoid When                                                                                         | Keep Portable                                                                        |
|-----------------------------|--------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| LangGraph-style state graph | Work has branching, checkpoints, interrupts, resume, or node-level observability.                      | A simple sequence or single tool call is enough. Graph shape would hide rather than clarify state. | State schema, node contracts, checkpoint records, interrupt payloads, eval fixtures. |
| AutoGen-style team          | Collaboration, role behavior, and transcript review are central.                                       | The team is just a chain with agent names. Conversation history becomes the only state store.      | Message schema, role contracts, termination rules, transcript evals, tool policy.    |
| Mastra-style runtime        | TypeScript product needs agents, tools, workflows, memory, evals, and observability packaged together. | Runtime conventions would hide product policy or deployment requirements.                          | Tool manifests, workflow contracts, memory rules, evals, trace schema.               |

| Option                       | Best When                                                                                     | Avoid When                                                                                             | Keep Portable                                                                  |
|------------------------------|-----------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| CrewAI-style flows and crews | Python workflows need flow-owned state plus bounded specialist crews.                         | Roles overlap, flow acceptance is vague, or crews replace deterministic workflow design.               | Flow state, task contracts, role permissions, crew outputs, acceptance evals.  |
| Mini-runtime/custom code     | Scope is narrow, policy needs exact control, or the team already has workflow infrastructure. | You need production durability, scaling, hosted observability, and ecosystem integrations immediately. | Everything: state, policy, tools, memory, evals, traces, deployment contracts. |

## Responsibility Matrix

| Responsibility | LangGraph-style                                     | AutoGen-style                                     | Mastra-style                                   | CrewAI-style                           | Mini-runtime                              |
|----------------|-----------------------------------------------------|---------------------------------------------------|------------------------------------------------|----------------------------------------|-------------------------------------------|
| State owner    | Graph state and checkpoints.                        | Team/task state outside the transcript.           | Runtime workflow and memory state.             | Flow state.                            | Your application model.                   |
| Control flow   | Nodes, edges, conditional transitions, interrupts.  | Team turn policy and termination.                 | Workflows and agent runtime.                   | Flows coordinate crews and tasks.      | Loop, router, or workflow code you write. |
| Tool policy    | Guard nodes, middleware, or tool wrappers.          | Manager/runtime execution boundary.               | Tool and workflow policy hooks.                | Role tools plus flow constraints.      | Explicit policy gate.                     |
| Memory         | State plus memory stores.                           | Transcript plus external memory.                  | Runtime memory abstractions.                   | Flow/crew context and external stores. | Context packet and memory policy.         |
| Evals          | Node paths, state diffs, checkpoints, final output. | Transcript turns, roles, tool calls, termination. | Runtime traces, tool calls, workflow outcomes. | Flow acceptance and role outputs.      | Trajectory tests you define.              |



| Responsibility | LangGraph-style                               | AutoGen-style                              | Mastra-style                                  | CrewAI-style                             | Mini-runtime                                 |
|----------------|-----------------------------------------------|--------------------------------------------|-----------------------------------------------|------------------------------------------|----------------------------------------------|
| Observability  | Per-node traces and checkpoint inspection.    | Structured message transcript.             | Runtime observability and eval hooks.         | Flow/task/role records.                  | Trace schema and storage you build.          |
| Deployment     | App runtime plus checkpoint and stores.       | App runtime plus agent service boundaries. | TypeScript runtime packaging.                 | Python flow app and workers.             | Existing product infrastructure.             |
| Escape hatch   | Nodes are plain code if state stays explicit. | Transcript can be exported and replayed.   | Keep tools/evals outside framework-only code. | Keep flow state separate from role chat. | Maximum control, maximum operational burden. |

## Choose By Risk

| Dominant Risk                          | Better Default                                | Why                                                                          |
|----------------------------------------|-----------------------------------------------|------------------------------------------------------------------------------|
| Lost progress or interrupted workflows | LangGraph-style or durable workflow           | Checkpoints, resumability, and state transitions are first-class.            |
| Unaccountable multi-agent behavior     | AutoGen-style transcript or CrewAI-style flow | You can inspect role turns, handoffs, and final acceptance.                  |
| TypeScript product runtime consistency | Mastra-style runtime                          | Agents, tools, workflows, memory, and evals share one application structure. |
| Python role-based workflow automation  | CrewAI-style flow                             | Flow state and crew execution map well to Python automation teams.           |
| Strict policy and minimal scope        | Mini-runtime/custom code                      | The application owns the exact boundary and avoids framework defaults.       |
| Framework uncertainty                  | Mini-runtime first, then migrate              | Building primitives once clarifies what the framework must provide.          |

## Migration And Escape Hatches

Do not let the framework become the only place where product logic exists. Keep these assets portable:

- state schemas;
- tool manifests and side-effect classes;

- policy rules;
- approval records;
- memory retention and deletion rules;
- prompt and instruction files;
- context packet shape;
- trace schema;
- eval fixtures;
- deployment and rollback notes;
- ADRs explaining why the framework was chosen.

Portability is not theoretical. Framework APIs change, teams switch stacks, and production systems outgrow early assumptions. The safest framework adoption keeps product contracts outside framework-specific decorators and callbacks wherever possible.

## Decision Checklist

Before choosing a framework, answer:

- What owns durable state?
- Where does policy run before side effects?
- How are tools typed, scoped, approved, and traced?
- What is the smallest replayable trace that explains a failure?
- Which evals fail the build?
- What happens when a run is interrupted?
- How do we roll back prompts, tools, policies, and model choices?
- What data must never enter memory or traces?
- Which parts can migrate if the framework changes?
- Which production incident would prove this framework was the wrong choice?

## Related Chapters

- [Framework Selection](#)
- [Real Framework Setup Notes](#)
- [Templates and Worksheets](#)
- [Building a Minimal Agent Runtime](#)
- [Agent Harnesses](#)
- [Lab Production Readiness Checklist](#)
- [Architecture Decision Records for Agents](#)
- [Observability and Evals](#)

# Real Framework Setup Notes

The labs teach architecture through small local implementations. This chapter shows how to translate those patterns into real framework projects without making the book depend on one language or one vendor.

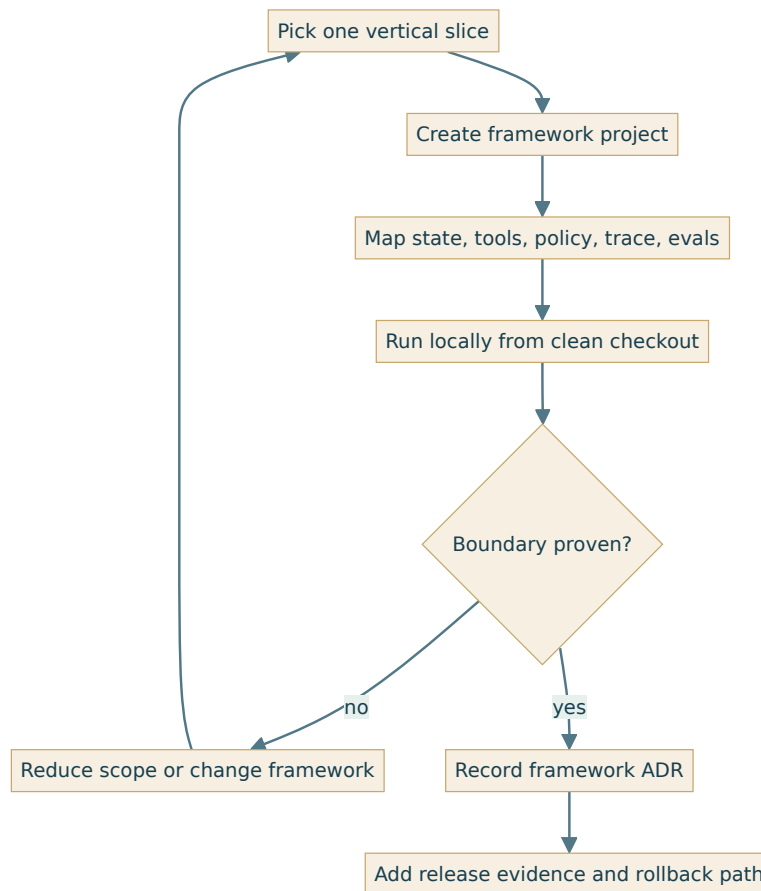
Use these notes after the framework-neutral labs. Keep the same architectural contracts: state schema, tool manifest, policy gate, trace schema, eval fixtures, and rollback plan. The framework should host those contracts, not replace them.

## How To Use This Chapter

Start with one vertical slice:

1. one user request;
2. one state object;
3. one read tool;
4. one side-effect tool behind policy or approval;
5. one trace;
6. one eval case;
7. one local run command;
8. one rollback or disable path.

Do not begin by porting every lab. Port the smallest behavior that proves the framework can own the responsibility you need it to own.



Use this flow before expanding the port. A framework is a candidate only after it proves the boundary that matters most: state, approval, tool authority, transcript ownership, workflow packaging, or eval integration.

## Shared Setup Rules

Every framework variant should define these files before the first demo:

| Artifact                    | Purpose                                                                              |
|-----------------------------|--------------------------------------------------------------------------------------|
| <code>.env.example</code>   | Names required keys without storing secrets.                                         |
| <code>state.schema.*</code> | Defines resumable state and migration fields.                                        |
| <code>tools.*</code>        | Declares tool names, inputs, outputs, side effects, and permissions.                 |
| <code>policy.*</code>       | Enforces action rules before retrieval, memory writes, tool calls, or final answers. |
| <code>trace.*</code>        | Emits run, model, tool, policy, approval, and evaluator events.                      |
| <code>evals/*</code>        | Stores happy path, negative path, and trajectory checks.                             |
| <code>README.md</code>      | Shows install, local run, test, eval, and cleanup commands.                          |

Keep secrets out of examples. Use `.env.example` for names and local environment variables for values.

## Setup Failure Playbook

Framework ports fail in repeatable ways. Treat setup failures as design evidence, not as local annoyance. A framework is not ready for the book’s production path until a new engineer can reproduce install, run, test, eval, and cleanup from a clean checkout.

| Failure                         | Common Signal                                                           | What To Capture                                                                      | Fix Or Decision                                                                      |
|---------------------------------|-------------------------------------------------------------------------|--------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| Wrong runtime version           | Python, Node.js, or package manager rejects install.                    | Runtime version, lockfile version, install command, failing package.                 | Pin the runtime in docs and CI, or choose a framework slice with supported versions. |
| Hidden provider dependency      | Demo imports a provider package that was not installed.                 | Missing module, provider package name, model route, <code>.env.example</code> field. | Make provider packages explicit and keep deterministic fallback tests.               |
| Secret required for local test  | Baseline test fails before architecture can be inspected.               | Command, missing environment variable, whether deterministic fallback exists.        | Split unit/eval tests from live-provider smoke tests.                                |
| Generated scaffold hides policy | Tool, approval, or memory rules live only inside example prompt text.   | File path, hidden rule, missing policy module, failed boundary test.                 | Move policy into code and add a trajectory eval.                                     |
| State cannot resume             | Interrupt, retry, or local server restart loses run state.              | Thread ID, checkpoint store, resume command, expected state diff.                    | Add persistent checkpointer or reject the framework for durable workflows.           |
| Trace is final-answer only      | Logs show output but not route, tool, policy, approval, or stop reason. | Trace fields present and missing, sample run ID, eval requirement.                   | Add instrumentation before expanding the port.                                       |
| Upgrade breaks behavior         | Package update changes routing, tool calls, memory, or termination.     | Old/new versions, changed trace, failing fixture, rollback command.                  | Require regression evals before framework upgrades.                                  |

Capture the failure in the framework ADR. The useful question is not “Can we install it?” The useful question is “Which production boundary did setup prove or fail to prove?”

Use this setup evidence record for each framework slice:

| Field           | Example Value                                            |
|-----------------|----------------------------------------------------------|
| Framework slice | <code>native-framework-examples/langgraph-refund/</code> |

| Field                    | Example Value                                                                             |
|--------------------------|-------------------------------------------------------------------------------------------|
| Runtime versions         | <code>python --version</code> , <code>node --version</code> , or package manager version. |
| Install command          | Exact command from clean checkout.                                                        |
| Baseline run command     | Smallest command that proves the main boundary.                                           |
| Eval command             | Scoped command that proves failure behavior.                                              |
| Required secrets         | Names only; no values.                                                                    |
| Deterministic fallback   | Yes/no and command.                                                                       |
| Known setup failure      | One captured failure and fix.                                                             |
| Rollback or disable path | Command, feature flag, or route removal.                                                  |

## LangGraph Variant

Use LangGraph when the main risk is stateful control flow: branching, checkpoints, interrupts, replay, or human approval waits.

Repository native examples:

- `native-framework-examples/langgraph-refund/` for approval, interrupt, and resume behavior.
- `native-framework-examples/langgraph-research-rag/` for source filtering, citation faithfulness, and escalation behavior.

Official setup references:

- [LangGraph install](#)
- [LangGraph graph API](#)
- [LangGraph local server](#)
- [LangGraph persistence](#)
- [LangGraph interrupts](#)

Typical local setup:

```
python3 -m venv .venv
source .venv/bin/activate
pip install -U langgraph
```

For a local LangGraph server, the official docs use the LangGraph CLI:

```
pip install -U "langgraph-cli[inmem]"
langgraph dev
```

Use the in-memory server only for local development. Production needs persistent storage for checkpoints and any long-term stores.

Porting path from the labs:

| Lab Asset      | LangGraph Mapping                                                      |
|----------------|------------------------------------------------------------------------|
| state object   | <code>StateGraph</code> state schema                                   |
| loop step      | node function                                                          |
| route decision | conditional edge                                                       |
| approval wait  | interrupt plus checkpoint                                              |
| trace event    | node, model, tool, policy, and interrupt spans                         |
| eval case      | graph input, expected state diff, expected route, expected stop reason |

Production questions:

- Which checkpointer stores thread-scoped graph state?
- How are thread IDs assigned and protected from cross-tenant access?
- Which nodes can cause side effects?
- Are side-effect nodes idempotent under retry or resume?
- Which state fields need migrations?
- Which interrupts require human approval records?

Native example acceptance check: the example must prove the highest-risk boundary. For refund workflows, it must pause at an approval interrupt, resume by thread ID, preserve prior state, and pass evals without issuing money. For Research RAG, it must omit stale and forbidden sources before answer synthesis, cite only approved current evidence, and escalate when approved evidence is missing.

## AutoGen Variant

Use AutoGen when the main risk is collaborative behavior: role contracts, message history, team termination, and transcript review.

Repository native example: `native-framework-examples/autogen-delivery/`.

Official setup references:

- [AutoGen documentation](#)
- [AgentChat user guide](#)
- [AgentChat installation](#)
- [AgentChat agents](#)
- [AgentChat teams](#)
- [AgentChat termination](#)
- [AutoGen to Microsoft Agent Framework migration guide](#)

Typical local setup:

```
python3 -m venv .venv
source .venv/bin/activate
pip install -U "autogen-agentchat" "autogen-ext[openai]"
```

AutoGen AgentChat currently requires Python 3.10 or later. AutoGen is community-maintained, so new long-lived Microsoft-stack projects should also evaluate Microsoft Agent Framework before standardizing on AutoGen. Keep provider packages explicit so model dependencies do not hide inside the framework.

Porting path from the labs:

| Lab Asset        | AutoGen Mapping                                               |
|------------------|---------------------------------------------------------------|
| supervisor       | team manager or coordinator                                   |
| worker contract  | agent role, tools, and expected output                        |
| transcript       | durable message record                                        |
| termination rule | team stop condition                                           |
| tool policy      | execution wrapper outside the message text                    |
| eval case        | transcript replay with role, tool, and termination assertions |

Production questions:

- Who owns the transcript as durable state?
- Which messages are persisted, redacted, and replayable?
- What stops the team?
- Which agent may call which tool?
- Can a retry duplicate a tool side effect?
- Which transcript failures become eval fixtures?

Native example acceptance check: the example must define role-specific AgentChat agents, a team termination rule, a normalized transcript export, and evals that fail on missing roles, invalid turn order, or missing final owner.

## Mastra Variant

Use Mastra when the main risk is packaging a TypeScript product runtime around agents, workflows, tools, memory, observability, and evals.

Repository native example: `native-framework-examples/mastra-refund/`.

Official setup references:

- [Mastra docs](#)



- [Mastra quickstart](#)
- [Mastra manual install](#)
- [Mastra agents](#)
- [Mastra tools](#)
- [Mastra workflows](#)
- [Mastra evals](#)

Typical local setup starts from the framework CLI:

```
npm create mastra@latest
```

Use the scaffold to inspect project structure, then map the book’s contracts into framework-owned modules. Do not leave product policy only inside generated example code.

Porting path from the labs:

| Lab Asset                  | Mastra Mapping                         |
|----------------------------|----------------------------------------|
| agent decision             | Mastra agent                           |
| deterministic control flow | workflow                               |
| tool registry              | typed tool declarations                |
| memory contract            | framework memory plus retention policy |
| trace schema               | runtime observability export           |
| eval case                  | framework eval plus repository fixture |

Production questions:

- Which parts are Mastra-owned and which remain application-owned?
- Where do tools declare side effects and permissions?
- How are workflows deployed, retried, and rolled back?
- How are traces and eval results exported to the team’s operational system?
- Which framework upgrades require regression evals?

Native example acceptance check: the example must define a refund draft agent, typed policy and draft tools, a workflow that enforces policy-before-draft order, and an eval that fails on money movement or customer messaging.

## CrewAI Variant

Use CrewAI when the main risk is Python workflow automation with flow-owned state and bounded specialist crews.

Repository native example: [native-framework-examples/crewai-delivery/](#).

Official setup references:

- [CrewAI docs](#)
- [CrewAI installation](#)
- [CrewAI quickstart](#)
- [CrewAI introduction](#)

CrewAI's current docs emphasize Python 3.10 through 3.13, `uv`-based installation, and a quickstart that scaffolds a Flow plus an agent crew. Keep the generated flow small enough that state transitions remain visible.

Porting path from the labs:

| Lab Asset       | CrewAI Mapping                            |
|-----------------|-------------------------------------------|
| flow state      | CrewAI Flow state                         |
| task delegation | crew tasks                                |
| worker contract | agent role, goal, tools, and output shape |
| merge policy    | flow acceptance step                      |
| trace event     | flow, task, and crew output records       |
| eval case       | flow output plus role-output assertions   |

Production questions:

- What does the Flow own that the Crew must not mutate implicitly?
- What does each role add that a deterministic function could not?
- How are crew outputs validated before the flow accepts them?
- What happens when one role fails or disagrees?
- Which flow checkpoints are needed before external side effects?

Native example acceptance check: the example must keep planner, reviewer, and tester outputs separate, then let the Flow decide final acceptance.

### Mini-Runtime Variant

Use the custom mini-runtime when the main risk is understanding and owning the architecture. It is not a framework replacement for every production need. It is a teaching and design tool.

Map the same contracts directly:

| Contract | Mini-Runtime Location                        |
|----------|----------------------------------------------|
| state    | explicit application object or table         |
| tools    | registry with schemas and side-effect labels |

| Contract | Mini-Runtime Location                             |
|----------|---------------------------------------------------|
| policy   | function called before authority                  |
| memory   | context packet plus governed storage              |
| trace    | append-only event list                            |
| evals    | deterministic tests over state, trace, and output |

The mini-runtime is valuable because it makes the hidden framework responsibilities visible. After building it, readers can judge whether LangGraph, AutoGen, Mastra, or CrewAI adds enough operational value to justify the abstraction.

### Framework-Agnostic Acceptance Checklist

Before calling a framework port complete, verify:

- install command is documented and reproducible;
- local run command works from a clean checkout;
- secrets live in environment variables, not source;
- state owner is named;
- tool side effects are declared;
- policy runs before authority;
- traces include model, tool, policy, and evaluator events;
- evals cover happy path, negative path, and trajectory;
- rollback or kill switch is documented;
- framework-specific code does not hide product contracts.

If the port passes only the happy path, it is still a demo.

# Templates and Worksheets

These templates turn the book's guidance into reviewable engineering artifacts. Use them when a lab becomes a product slice, when a team chooses a framework, or when an agent gains more authority.

Copy only the sections that matter. A short, complete decision record beats a long template with empty answers.

For filled examples, compare these templates with the [Capstone Projects](#).

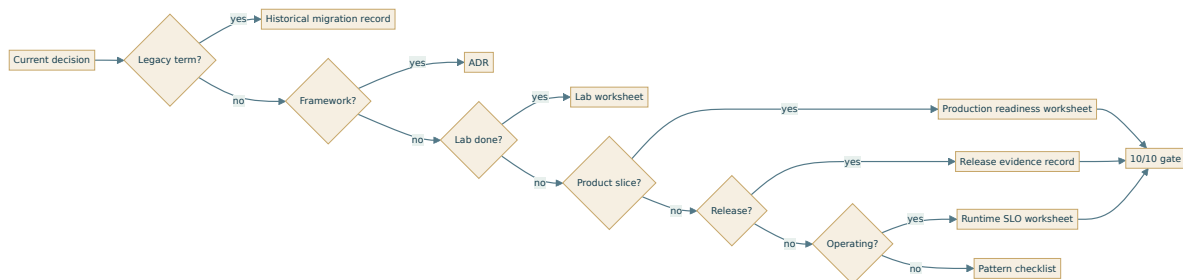
## Choose The Right Artifact

Start with the artifact that matches the decision in front of you. Do not fill every template by default.

| Situation                                                        | Use This Artifact                                       | Done When                                                                                                                 |
|------------------------------------------------------------------|---------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| Choosing LangGraph, AutoGen, Mastra, CrewAI, or a custom runtime | <a href="#">Framework selection ADR template</a>        | The team can name what the framework owns and what the application still owns.                                            |
| Studying completed agent ADRs                                    | <a href="#">Completed agent ADR examples</a>            | The team can compare its decision record with realistic authority, RAG, and multi-agent examples.                         |
| Studying completed lab evidence                                  | <a href="#">Completed lab evidence examples</a>         | The team can compare its worksheet with concrete commands, traces, failure paths, production gaps, and release decisions. |
| Studying production readiness evidence                           | <a href="#">Completed production readiness examples</a> | The team can compare its production worksheet with concrete owners, gates, blockers, readiness ratings, and next actions. |
| Reviewing an Agentic RAG answer                                  | <a href="#">Agentic RAG query trace worksheet</a>       | The team can reconstruct source selection, omitted evidence, citation checks, and the final answer/refusal decision.      |
| Reviewing a debate or consensus workflow                         | <a href="#">Debate and consensus review checklist</a>   | Independence, merge policy, dissent handling, budget, baseline comparison, and judge errors are reviewed.                 |
| Threat-modeling an agent route                                   | <a href="#">Agent threat model worksheet</a>            | Private data, untrusted content, tool authority, STRIDE risks, evals, and trace evidence are reviewed together.           |
| Reviewing agent UX and trust                                     | <a href="#">Agent UX review worksheet</a>               | Runtime states, visible evidence, user controls, approval UX, correction paths, and UX evals are explicit.                |

| Situation                                 | Use This Artifact                                                        | Done When                                                                                                                                 |
|-------------------------------------------|--------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Finishing a lab                           | <a href="#">Lab completion worksheet</a>                                 | The run command, test command, pattern boundary, and lesson are recorded.                                                                 |
| Completing Lab 02 Agent Loop and Planning | <a href="#">Lab 02 planning loop guided exercise worksheet</a>           | Baseline plan trace, changed input, unsupported step, missing input, and stop-condition contract are captured.                            |
| Completing Lab 03 Agentic RAG             | <a href="#">Lab 03 Agentic RAG guided exercise worksheet</a>             | Baseline retrieval, grounding change, missing-evidence behavior, source contract, native graph comparison, and eval fixture are captured. |
| Completing Lab 06 Observability and Evals | <a href="#">Lab 06 observability and evals guided exercise worksheet</a> | Trace contract, missing-policy failure, negative case, CI gate, and incident-to-eval note are captured.                                   |
| Completing Lab 07 Runtime Packaging       | <a href="#">Lab 07 runtime packaging guided exercise worksheet</a>       | Runtime boundaries, tool order, forbidden side effects, rollback, and native Mastra comparison are captured.                              |
| Completing Lab 12 State Graphs            | <a href="#">Lab 12 state graph guided exercise worksheet</a>             | Interrupt, checkpoint, resume, replay safety, native LangGraph comparison, and production follow-up are captured.                         |
| Turning a lab into a product slice        | <a href="#">Lab production readiness worksheet</a>                       | Missing state, policy, eval, trace, rollback, and ownership controls are explicit.                                                        |
| Reviewing a production candidate          | <a href="#">Production readiness worksheet</a>                           | Every high-authority path has evidence or is scoped out of release.                                                                       |
| Operating a production route              | <a href="#">Runtime SLO and incident review worksheet</a>                | SLOs, dashboard panels, incident triage, trace review, eval conversion, and rollback actions are explicit.                                |
| Preparing a release                       | <a href="#">Release evidence record</a>                                  | The release has eval output, known risk, approver, rollback owner, and rollback command.                                                  |
| Reviewing agentic system composition      | <a href="#">Agentic system architecture review checklist</a>             | Experience, control, execution, knowledge, safety, observability, evaluation, contracts, and authority boundaries are explicit.           |
| Reviewing a full system architecture      | <a href="#">Reference architecture review checklist</a>                  | Entry, routing, state, knowledge, tools, memory, approval, evals, observability, and release ownership are reviewed together.             |

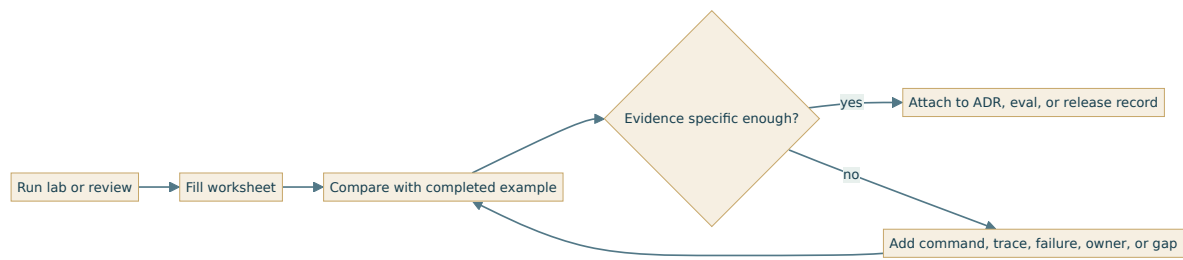
| Situation                                 | Use This Artifact                                            | Done When                                                                                                                     |
|-------------------------------------------|--------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| Reviewing unattended event-triggered work | <a href="#">Event-triggered agent review checklist</a>       | Event contract, admission, dedupe, ordering, retries, dead letters, storms, traces, and evals are explicit.                   |
| Reviewing a self-healing workflow         | <a href="#">Self-healing workflow review checklist</a>       | Failure classes, recovery policy, idempotency, compensation, breakers, replay packets, and regression evals are explicit.     |
| Reviewing a personal agent architecture   | <a href="#">Personal agent architecture review checklist</a> | Consent, local/cloud split, credentials, memory, user controls, connectors, and audit evidence are explicit.                  |
| Checking the final bar                    | <a href="#">10/10 production gate scorecard</a>              | The system is reviewable, testable, observable, reversible, and owned.                                                        |
| Comparing a capstone with your own system | <a href="#">Capstone review scorecard</a>                    | You have a concrete adaptation plan and a list of blocking gaps.                                                              |
| Scoring a pattern before adoption         | <a href="#">Pattern evaluation scorecard</a>                 | The pattern has evidence for goal, boundary, autonomy, tools, state, context, security, evals, observability, and operations. |
| Translating older agent terminology       | <a href="#">Historical pattern migration record</a>          | A vague legacy label is mapped to a current architecture decision with evidence.                                              |
| Reviewing a reusable skill package        | <a href="#">Skill review checklist</a>                       | Activation, instruction shape, assets, security, versioning, tests, observability, and final decision are explicit.           |
| Reviewing one pattern                     | The matching pattern review checklist below                  | Use cases, avoid cases, failure modes, evals, and production controls have evidence.                                          |



Use the smallest artifact that proves the decision. Add the production gate only when the system is close enough to release that missing evidence should block progress.

## Completed Examples

Blank templates help only after the reader understands the evidence standard. Use completed examples to compare the density and specificity of your own artifact.



Use [Completed agent ADR examples](#) for architecture decisions. Use [Completed lab evidence examples](#) for lab evidence packs and eval reviews. Use [Completed production readiness examples](#) when a team needs to calibrate owner, gate, blocker, rollback, and release-decision evidence.

Downloadable versions:

- [A2A agent interoperability review checklist](#)
- [Agentic RAG query trace worksheet](#)
- [Agentic system architecture review checklist](#)
- [Agent threat model worksheet](#)
- [Agent UX review worksheet](#)
- [Capstone review scorecard](#)
- [Completed agent ADR examples](#)
- [Completed lab evidence examples](#)
- [Completed production readiness examples](#)
- [Computer-use agent review checklist](#)
- [CrewAI flows and crews review checklist](#)
- [Debate and consensus review checklist](#)
- [Cost controls and runtime budgets review checklist](#)
- [Deployment walkthrough review checklist](#)
- [Domain agent architecture review checklist](#)
- [Durable workflows review checklist](#)
- [Evaluator-optimizer review checklist](#)
- [Event-triggered agent review checklist](#)
- [Framework selection ADR template](#)
- [Historical pattern migration record](#)
- [Lab 02 planning loop guided exercise worksheet](#)
- [Lab 03 Agentic RAG guided exercise worksheet](#)
- [Lab 06 observability and evals guided exercise worksheet](#)
- [Lab 07 runtime packaging guided exercise worksheet](#)
- [Lab 12 state graph guided exercise worksheet](#)
- [Lab completion worksheet](#)
- [Lab production readiness worksheet](#)
- [Knowledge-bound agents review checklist](#)
- [Long-term episodic memory review checklist](#)
- [Mastra runtime review checklist](#)
- [Memory-augmented agent review checklist](#)
- [MCP-first tool use review checklist](#)

- [Observability and evals review checklist](#)
- [Personal agent architecture review checklist](#)
- [Parallel agents review checklist](#)
- [Pattern evaluation scorecard](#)
- [Planning and execution review checklist](#)
- [Policy enforcement review checklist](#)
- [Production evaluation feedback loop review checklist](#)
- [Production readiness worksheet](#)
- [Release evidence record](#)
- [Reference architecture review checklist](#)
- [Reflection pattern review checklist](#)
- [ReAct review checklist](#)
- [Runtime SLO and incident review worksheet](#)
- [Self-improvement review checklist](#)
- [Self-healing workflow review checklist](#)
- [Secure agent communication review checklist](#)
- [Semantic recall and RAG review checklist](#)
- [Skill review checklist](#)
- [Task delegation review checklist](#)
- [Tool capability design review checklist](#)
- [Working memory review checklist](#)
- [10/10 production gate scorecard](#)

## Framework Selection ADR

Use this ADR when adopting LangGraph, AutoGen, Mastra, CrewAI, a mini-runtime, or another framework.

```
ADR-000: Choose [Framework] for [Agent or Workflow]
```

```
Status
```

```
Proposed | Accepted | Superseded
```

```
Context
```

```
What product problem are we solving?
```

```
What user-visible workflow will this framework host?
```

```
What constraints matter: language, deployment, compliance, team skills, latency, cost, or existing infrastructure?
```

```
Decision
```

```
We will use [framework] for [scope].
```



The framework will own [state/control flow/tools/memory/evals/observability/deployment].  
The application will still own [policy/domain data/security/approval/rollback].

## ## Alternatives Considered

| Option                     | Why It Fit | Why We Did Not Choose It |
|----------------------------|------------|--------------------------|
| ---                        | ---        | ---                      |
| Direct code / mini-runtime |            |                          |
| LangGraph                  |            |                          |
| AutoGen                    |            |                          |
| Mastra                     |            |                          |
| CrewAI                     |            |                          |

## ## Responsibility Boundary

| Responsibility | Owner                          | Evidence                              |
|----------------|--------------------------------|---------------------------------------|
| ---            | ---                            | ---                                   |
| State          | framework/application/platform | schema, checkpoint, migration plan    |
| Control flow   | framework/application/platform | graph, workflow, team, flow, loop     |
| Tools          | framework/application/platform | manifest, schema, permission model    |
| Policy         | framework/application/platform | enforcement point before authority    |
| Memory         | framework/application/platform | retention, deletion, correction rules |
| Observability  | framework/application/platform | trace schema and dashboard            |
| Evals          | framework/application/platform | fixtures, thresholds, CI gate         |
| Deployment     | framework/application/platform | runbook, rollback, kill switch        |

## ## Vertical Slice Proof

- user request:
- state object:
- read tool:
- side-effect tool:
- policy decision:
- trace:
- eval:
- rollback:

## ## Acceptance Criteria

- local install and run commands are documented;
- state can be inspected and replayed;
- policy runs before side effects;
- evals fail the build for critical regressions;
- traces explain one failed run;

- rollback can disable the risky capability;
- framework-specific code does not hide product policy.

## ## Consequences

What gets easier?

What gets harder?

What lock-in or migration risk remains?

Which production incident would make us revisit this decision?

## ## Review Trigger

Review this ADR after model upgrade, framework upgrade, new write-capable tool, new memory type, production incident, or repeated human override.

## Production Readiness Worksheet

Use this worksheet before exposing the system to real users, real data, or real side effects.

| Gate          | Answer                                                         | Evidence                              |
|---------------|----------------------------------------------------------------|---------------------------------------|
| Owner         | Who owns the runtime and incidents?                            | team, on-call, runbook                |
| Scope         | Which users, tenants, tools, data, and workflows are in scope? | ADR, service config                   |
| State         | What state exists and where is it persisted?                   | schema, checkpoint store              |
| Idempotency   | Which actions can be retried safely?                           | idempotency keys, side-effect records |
| Tools         | Which tools can be called and with what authority?             | tool manifest, permission map         |
| Policy        | Where does policy run before authority?                        | enforcement code, tests               |
| Approval      | Which actions require approval?                                | approval schema, UI/API, expiry       |
| Memory        | What can be read or written?                                   | retention, deletion, correction rules |
| Observability | Can one failed run be reconstructed?                           | trace dashboard, redaction proof      |
| Evals         | What blocks release?                                           | eval fixtures, thresholds, CI output  |
| Security      | How are secrets, egress, and sandboxing handled?               | secret manager, network policy        |

| Gate          | Answer                                                     | Evidence              |
|---------------|------------------------------------------------------------|-----------------------|
| Rollback      | How do we disable model, prompt, tool, workflow, or agent? | runbook, feature flag |
| Incident Loop | How do incidents become evals?                             | post-incident process |

Readiness rating:

green: every high-authority path has evidence

yellow: limited internal or read-only release only

red: demo only; no real users, data, or side effects

## Lab-To-Production Checklist

Use this checklist after completing any hands-on lab.

lab:

target product slice:

framework/language:

owner:

Architecture

[ ] Pattern selected and linked

[ ] Framework decision recorded

[ ] State owner named

[ ] Tool boundary defined

[ ] Policy boundary defined

[ ] Human approval boundary defined when needed

Implementation

[ ] Install command documented

[ ] Local run command documented

[ ] Test command documented

[ ] Eval command documented

[ ] .env.example committed

[ ] Secrets excluded from source

[ ] Tool schemas validated

[ ] Side effects use idempotency keys

[ ] Errors have typed outcomes

Production

[ ] Checkpoint or explicit stateless decision recorded

```
[] Trace schema implemented
[] Trace redaction implemented
[] Eval threshold defined
[] CI gate configured
[] Rollback path documented
[] Kill switch tested
[] Runbook created

Evidence
[] Test output attached
[] Eval output attached
[] Example trace attached
[] ADR linked
[] Owner accepted residual risk
```

If a checked item has no evidence, leave it unchecked.

## Release Gate Checklist

Use this gate for each production release.

Download the [release evidence record](#) when preparing a public book release, GitHub Pages deployment, or release PR.

| Check                | Required Before Release                                           |
|----------------------|-------------------------------------------------------------------|
| Prompt/model changed | run task, refusal, policy, tool, and cost evals                   |
| Tool changed         | run authorization, schema, idempotency, and error evals           |
| Policy changed       | run false-allow, false-deny, approval, and escalation evals       |
| Memory changed       | run read-scope, write-policy, deletion, and correction evals      |
| Retrieval changed    | run access, freshness, citation, and missing-evidence evals       |
| Runtime changed      | run retry, cancellation, checkpoint, and trace completeness evals |

Release decision:

```
release version:
change type:
eval dataset version:
passing threshold:
actual result:
known failures:
approved by:
```

rollback owner:  
rollback command:

## Incident-To-Eval Worksheet

Use this after production incidents, near misses, or serious human overrides.

incident ID:

date:

service:

trace ID:

owner:

What happened?

Which boundary failed?

☐ state

☐ tool

☐ policy

☐ approval

☐ memory

☐ retrieval

☐ model/prompt

☐ workflow

☐ observability

☐ eval gate

What should have happened?

New eval case:

input:

expected trajectory:

expected tool behavior:

expected policy behavior:

expected output:

blocking threshold:

Release rule:

☐ blocks future release

☐ warning only

☐ monitor only

Follow-up:

```
code change:
policy change:
runbook change:
ADR update:
owner:
due date:
```

An incident that does not produce an eval, a policy change, or a runbook update is likely to repeat.

## Related Chapters

- [Framework Selection](#)
- [Real Framework Setup Notes](#)
- [Cross-Framework Decision Matrix](#)
- [Architecture Decision Records for Agents](#)
- [Deployment Walkthrough](#)
- [Lab Production Readiness Checklist](#)

# Evaluation-Driven Agent Development

Evaluation turns agent development from demo-driven guessing into engineering.

Without evals, the team is arguing from demos, anecdotes, and vibes. With evals, the team can say what failed, why it matters, whether the next change helped, and what it broke.

Use this chapter when an agent is moving from prototype to production, or when a prototype already feels impressive but nobody can prove it is reliable.

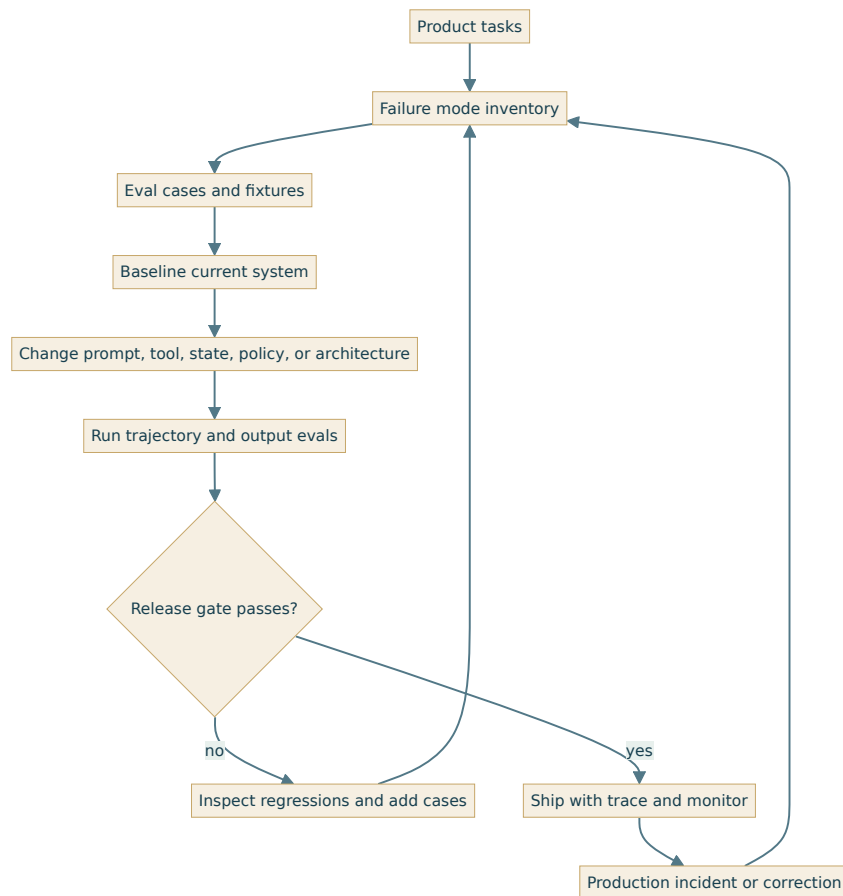
## Intent

Build agents from failure modes and success metrics, not from demos. A demo shows that the agent can work once. An eval suite shows whether it keeps working across the cases that matter.

The eval loop is:

1. list the tasks the agent must handle;
2. list the ways those tasks can fail;
3. define business and quality metrics;
4. create fixtures and datasets;
5. run the agent against those datasets;
6. inspect failures;
7. improve prompts, tools, state, policy, or architecture;
8. repeat on every meaningful change.

The key word is architecture. If an eval fails because the agent lacks state, a prompt edit is the wrong fix. If it fails because the tool is unsafe, a better instruction is not enough. Evals should be allowed to change the design.



Use the loop as a forcing function. Every meaningful change should name the failure it targets, the eval that proves improvement, and the release gate it must not break.

## What To Evaluate

Evaluate the full trajectory, not only the final answer.

| Layer        | What To Check                                                 |
|--------------|---------------------------------------------------------------|
| Routing      | Correct path, confidence, fallback behavior.                  |
| Planning     | Valid steps, dependency order, missing constraints.           |
| Retrieval    | Source relevance, freshness, coverage, citations.             |
| Tool use     | Correct tool, valid input, safe side effects, error handling. |
| Memory       | Correct recall, safe write, no stale or private leakage.      |
| Policy       | Permissions, approval requirements, refusals.                 |
| Final output | Correctness, completeness, tone, format, evidence.            |
| Operations   | Latency, cost, retries, breaker events, escalation.           |



If only the final response is evaluated, the team will miss the failure that caused it. The answer may look right even though the agent used stale evidence, skipped approval, retried a tool six times, or wrote bad memory for the next run.

## Evaluation Lenses By Responsibility

Different patterns need different evidence. Do not use one generic “answer quality” score for every agentic system.

| Pattern Responsibility   | Primary Question                                                 | Minimum Proof                                                                                |
|--------------------------|------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| Loop control             | Did the run stop for the right reason?                           | Step budget, tool budget, timeout, retry, and stop-reason cases.                             |
| Context assembly         | Did the model see the right evidence and exclusions?             | Context packet assertions for included sources, omitted sources, freshness, and budget.      |
| Tool use                 | Did the system use the minimum safe authority?                   | Mocked tools, forbidden-tool cases, schema checks, approval checks, and error paths.         |
| Memory                   | Did the system remember only what policy allows?                 | Allowed write, denied write, correction, deletion, stale recall, and tenant isolation cases. |
| Retrieval                | Did the system find and cite the right evidence?                 | Relevance, coverage, citation, missing-evidence, and conflicting-source cases.               |
| Multi-agent coordination | Did delegation improve the result without hiding accountability? | Single-agent baseline, worker failure, disagreement, merge, and final owner cases.           |
| Human approval           | Did the right action pause with enough information?              | Approval-required, approval-denied, timeout, resume, and audit-record cases.                 |
| Runtime operation        | Can the team explain and recover from failure?                   | Trace completeness, replay, rollback, breaker, incident fixture, and release-gate cases.     |

This table is a design tool. If the selected pattern introduces a risk, the eval must target that risk directly.

## Minimal Eval Case Contract

An eval case should be small enough to review and specific enough to fail for the right reason.

```
type AgentEvalCase = {
 caseId: string;
```

```

goal: string;
input: string;
mockedTools?: Record<string, unknown[]>;
expected: {
 finalStatus?: "succeeded" | "needs_human" | "refused" | "failed";
 stopReason?: string;
 toolsCalled?: string[];
 toolsNotCalled?: string[];
 requiredTraceEvents?: string[];
 requiredCitations?: string[];
 memoryWrites?: "none" | "allowed" | "denied" | "review_required";
};
};

```

The schema is intentionally plain. It lets the team test final output, trajectory, policy, memory, and observability without turning the eval suite into another agent project.

## Failure Mode Inventory

Start each agent project with a failure mode inventory. This is the most useful early artifact because it turns fear into testable cases.

Examples:

- selects the wrong route;
- asks for information it already has;
- misses a required approval;
- calls a write tool when read-only analysis is enough;
- retrieves irrelevant evidence;
- trusts stale memory;
- loops without progress;
- gives a confident answer with weak evidence;
- exposes sensitive data;
- exceeds cost or latency budget;
- cannot resume after interruption.

Each failure mode should map to at least one test, trace query, or production alert. If a failure is important but invisible, the system is not ready.

A good inventory is specific to the product. “Hallucination” is too broad. “Answers refund-policy questions without citing the current policy document” is testable. “Tool misuse” is too broad. “Calls `issue_refund` before a manager approval record exists” is testable.

## Metrics

Use a small set of metrics that match the product. More metrics do not create more truth. They usually create more ways to ignore the real failure.

Common metrics:

- task completion rate;
- correctness;
- evidence coverage;
- tool-call accuracy;
- route accuracy;
- approval precision and recall;
- hallucination rate;
- user correction rate;
- human escalation rate;
- cost per completed task;
- latency by step;
- incident rate.

Do not optimize one metric in isolation. A lower escalation rate is bad if the agent is making unsafe decisions instead. Lower latency is bad if the agent is skipping retrieval. Higher task completion is bad if it completes the wrong task.

The best metrics form a constraint set: quality cannot improve by violating safety, cost, latency, evidence, or approval requirements.

## Eval Dataset Types

| Dataset            | Source                     | Use                                        |
|--------------------|----------------------------|--------------------------------------------|
| Golden tasks       | Handwritten examples       | Basic regression coverage.                 |
| Adversarial tasks  | Designed edge cases        | Safety, routing, policy, prompt injection. |
| Production traces  | Real runs with redaction   | Realistic behavior and long-tail failures. |
| Human-labeled data | Subject matter experts     | Ground truth for judgment-heavy tasks.     |
| Synthetic variants | Generated from known cases | Coverage expansion after human review.     |
| Incident fixtures  | Past failures              | Prevent regressions.                       |

Production traces are valuable, but they need privacy controls and careful labeling. Do not dump raw user conversations into an eval pipeline without redaction, retention rules, and access controls.

Synthetic data is useful for expanding coverage, but it should not become the ground truth by itself. Generated cases often miss the messy parts of real usage: missing context, contradictory user instructions, weird phrasing, stale records, and partial tool outages.

## Judges

Model judges can help, but they need constraints. They are reviewers, not law.

Use model judges for:

- subjective quality;
- rubric-based review;
- comparing two outputs;
- checking whether a claim is supported by evidence.

Use deterministic checks for:

- schema validity;
- required fields;
- forbidden tools;
- budget limits;
- citation presence;
- route labels;
- permission gates.

Combine both. A model judge should not be the only control before a high-risk action.

Also evaluate the judge. Give it known good and bad examples. Track false accepts and false rejects. If the judge rewards polished prose over evidence, it will make the agent look better while the system gets worse.

## Evaluation-Driven Development Loop

For every meaningful agent change:

1. Add or update eval cases before changing prompts or tools.
2. Run the current system to establish a baseline.
3. Make the smallest change that targets the failure.
4. Run the eval suite again.
5. Inspect regressions, not only the average score.
6. Promote the change only if it improves the targeted behavior without harming critical cases.

This loop works for prompts, tools, policies, memory, routing, models, and orchestration.

Average scores are dangerous. A change that improves easy cases and breaks policy cases is not an improvement. Keep a small set of blocking evals that must pass every time: schema validity, forbidden tool calls, approval behavior, privacy boundaries, and known incident fixtures.

## Production Monitoring

Runtime monitoring should feed the eval suite. The production system is the best source of cases the team failed to imagine, so capture the signals that expose them: failed runs, human overrides, user corrections, low-confidence routes, breaker events, tool failures, high-cost and slow runs, policy denials, and unusual memory writes. Every serious incident should become at least one eval case. The goal is not blame. The goal is to make the failure hard to repeat.

Monitoring should also show when the eval suite is stale. If production traces contain task types, tool paths, or policy decisions that never appear in tests, the eval suite no longer represents the system.

For the production operating loop, see [Production Evaluation Feedback Loops](#).

## Release Gates

Before shipping a meaningful agent change, define what must pass.

Minimum release gates:

- no schema failures in golden tasks;
- no forbidden tool calls;
- no missing trace IDs;
- no approval bypasses;
- no regression on incident fixtures;
- no unsupported claims in evidence-bound answers;
- no unresolved high-risk failures from the review queue.

These gates should be boring. That is the point. They keep the team from shipping an exciting demo that reopens old failures.

## Related Chapters

- [Observability and Evals](#)
- [Production Evaluation Feedback Loops](#)
- [Circuit Breakers, Fallbacks, and Replay](#)
- [Policy Enforcement](#)
- [Evaluator-Optimizer](#)
- [Agent Development Lifecycle](#)

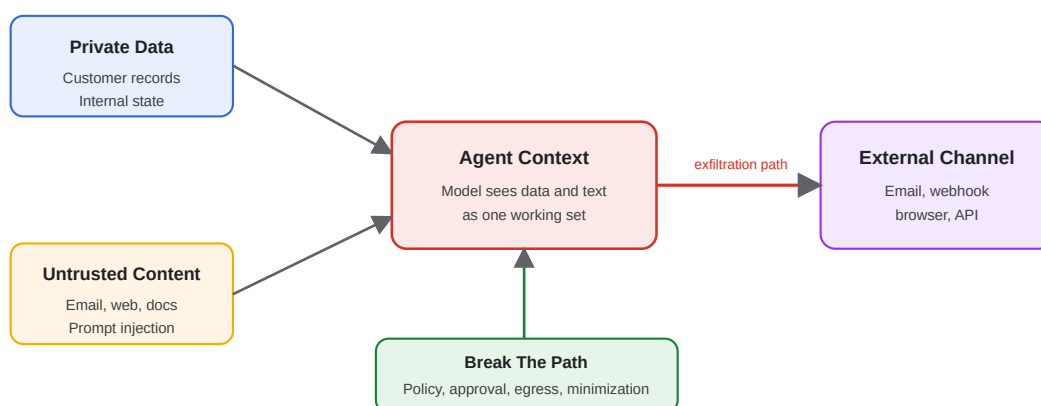
# Agent Threat Model

Agent security starts with a simple question: what can the model cause the system to do? A chatbot can give a bad answer. An agent can read private data, combine it with untrusted instructions, and send something outside the boundary. That is a different risk class, and it deserves a different threat model.

So do not start the threat model with “the model might hallucinate.” Start it with authority, data, tools, and communication.

Download the reusable worksheet: [agent threat model worksheet](#).

## Agent Threat Model Risk Path



## The Dangerous Combination

The most dangerous agent shape combines three capabilities:

1. access to private or trusted data;
2. exposure to untrusted content;
3. the ability to communicate or act outside the system.

When all three sit in the same execution path, the agent can become an exfiltration bridge. The shape is easy to picture: the agent reads internal customer data, then browses a web page, email, ticket, document, or repository comment that someone else controls, and that untrusted content tells the model to send the private data out through an email, webhook, issue comment, browser form, or API call.

The model may not experience this as an attack at all. It experiences it as just another instruction in context. That is precisely why prompt-only defenses are not enough.

## Security Boundary

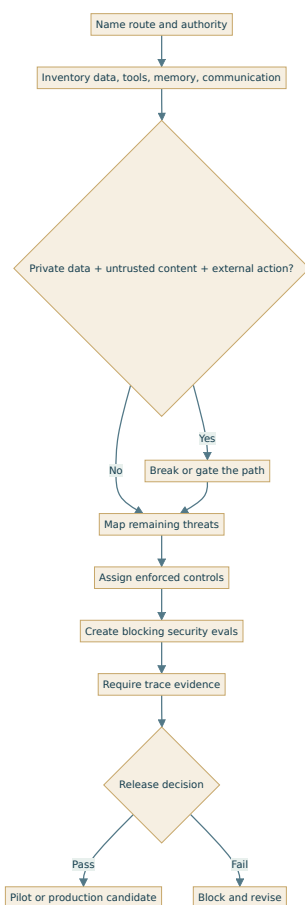
The model can propose; software must decide whether the proposal is allowed. Keep those responsibilities in separate hands.

| Layer         | Responsibility                                                       |
|---------------|----------------------------------------------------------------------|
| Model         | Interprets the task, proposes actions, summarizes evidence.          |
| Orchestrator  | Owns the loop, state, budgets, route, and stop condition.            |
| Policy engine | Decides whether the action is allowed.                               |
| Tool gateway  | Validates, executes, logs, denies, or routes for approval.           |
| Sandbox       | Limits filesystem, network, process, browser, and credential access. |
| Observability | Records enough trace data for replay, audit, and incident response.  |

If the model is also the policy engine, the boundary is weak by construction: the same context that carries the attack also carries the policy instruction meant to stop it.

## Threat Review Flow

Use this flow when a new route, tool, memory type, data source, or communication channel is added. A threat model is useful only if it ends in controls, evals, traces, and a release decision.



## STRIDE Map For Agents

STRIDE is useful only after it is translated into agent-specific boundaries. Use it to ask what the model, runtime, tools, memory, and communication paths can be tricked into doing.

| Threat Class           | Agent-Specific Question                                                                                                     | Control                                                                                  | Evidence                                                                     |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|------------------------------------------------------------------------------|
| Spoofing               | Can a user, tool, agent, MCP server, webhook, or document pretend to be a trusted actor or instruction source?              | Signed identity, audience checks, route-scoped credentials, source labels.               | Identity claims in trace, denied wrong-audience request, source trust label. |
| Tampering              | Can untrusted content modify goals, policy, tool arguments, memory, traces, or evaluation fixtures?                         | Instruction/data separation, schema validation, immutable policy, protected trace store. | Rejected hostile tool result, denied memory write, trace integrity check.    |
| Repudiation            | Can a user, agent, tool, or approver deny an action because the system lacks a durable record?                              | Run IDs, approval records, policy decision logs, tool audit events.                      | Complete trace from request to stop reason.                                  |
| Information disclosure | Can private data, secrets, citations, memory, or traces leak through output, tools, logs, browser, or cross-agent messages? | Data minimization, redaction, egress allowlist, destination classification.              | Redaction test, blocked egress, private-data exposure eval.                  |
| Denial of service      | Can loops, retries, retrieval, tools, queues, model calls, or approval waits exhaust cost, latency, quota, or humans?       | Budgets, timeouts, concurrency limits, circuit breakers, cancellation.                   | Budget exhaustion case, queue limit alert, breaker trace.                    |
| Elevation of privilege | Can the agent move from read to write, draft to send, one tenant to another, or one tool scope to broader authority?        | Least privilege, route-level tool allowlists, approval gates, scoped tokens.             | Forbidden tool eval, cross-tenant denial, approval-required trace.           |

This map should produce tickets, not just discussion. Every high-risk row needs one owner, one enforced control, one eval, and one trace field.

## Classify Tool Capabilities

Every tool should carry capability metadata. Without it, the runtime cannot reason about risk at all.



| Capability               | Question                                                                                                     |
|--------------------------|--------------------------------------------------------------------------------------------------------------|
| Private data access      | Can this tool read customer, tenant, secret, internal, or privileged data?                                   |
| Untrusted content access | Can this tool fetch content controlled by users, external websites, emails, documents, tickets, or comments? |
| External communication   | Can this tool send data to another person, system, network, repository, browser, or workflow?                |
| Side effects             | Can this tool write, delete, purchase, deploy, message, mutate, or trigger work?                             |
| Credential scope         | What credentials does the tool execute with?                                                                 |
| Approval requirement     | Which calls require a human or higher-trust workflow?                                                        |

This classification should live outside the prompt. A tool manifest, gateway config, policy file, or service registry is far easier to audit than a paragraph of natural-language instructions.

The policy check can be small, but it must sit outside the model:

```
interface ToolCapability {
 name: string;
 readsPrivateData: boolean;
 readsUntrustedContent: boolean;
 communicatesExternally: boolean;
 requiresApproval: boolean;
}

function blocksDangerousPath(tools: ToolCapability[]): boolean {
 const readsPrivateData = tools.some(tool => tool.readsPrivateData);
 const readsUntrustedContent = tools.some(tool => tool.readsUntrustedContent);
 const communicatesExternally = tools.some(tool => tool.communicatesExternally);

 return readsPrivateData && readsUntrustedContent && communicatesExternally;
}

function authorizeToolPlan(tools: ToolCapability[]) {
 if (blocksDangerousPath(tools)) {
 return {
 allowed: false,
 reason: 'private_data_plus_untrusted_content_plus_external_comm'
 };
 }

 if (tools.some(tool => tool.requiresApproval)) {
```

```
 return { allowed: false, reason: 'approval_required' };
 }

 return { allowed: true, reason: 'allowed' };
}
```

This does not make the whole system secure by itself. It shows the boundary: the model can propose a plan, but software decides whether the tool chain is allowed.

## Break The Dangerous Path

The goal is not to ban useful agents. It is to make sure at least one part of the dangerous path is missing or controlled.

| Control                            | What It Removes Or Restricts                                                                     |
|------------------------------------|--------------------------------------------------------------------------------------------------|
| No private data in this route      | The agent can inspect untrusted content and communicate, but cannot access sensitive data.       |
| No untrusted content in this route | The agent can work with trusted internal data and tools, but cannot ingest hostile instructions. |
| No external communication          | The agent can read and reason, but cannot exfiltrate or act outside the boundary.                |
| Policy-gated tool proxy            | The agent can propose calls, but policy decides what executes.                                   |
| Human approval                     | High-risk actions require a person to inspect the proposed action and evidence.                  |
| Egress controls                    | Network and messaging paths are limited by route, domain, destination, or data class.            |
| Data minimization                  | Tools return only the fields required for the task.                                              |
| Redaction                          | Sensitive values are removed before they enter model context or traces.                          |

The word that matters is enforce. A sentence in the system prompt is not the same thing as an enforced boundary, and attackers know the difference even when the model does not.

## Policy-Gated Tool Calls

For production agents, tool calls should pass through an enforcement layer that can allow the call, deny it, require approval, transform or redact its arguments, attach an idempotency key, log the policy decision, and link that decision back to the run trace. Implement it as a gateway, proxy, middleware, or tool-registry wrapper; the exact shape matters less than the boundary itself. The model should not be calling high-risk tools directly.

## Untrusted Content

Treat tool results, retrieved documents, web pages, emails, tickets, comments, logs, and user-uploaded files as data, not instructions. None of them should be able to redefine the system goals, the tool permissions, the approval requirements, the output destinations, the memory-write rules, the policy exceptions, or the agent's identity and role. The agent can summarize untrusted content all it likes. It must not obey untrusted content as governance.

## Memory And Persistence Risks

Memory makes attacks durable. An unsafe write can poison future runs long after the original interaction is gone, which is why memory writes need policy of their own. A memory write should record its source, the actor, the timestamp, the privacy class, the expiry or retention rule, a confidence level, the reason for writing, and a correction or deletion path. Do not let the model silently rewrite what the system will believe tomorrow.

## Evaluation Guidance

Security evals should test behavior across the full trajectory, not only the final answer. Build cases where a retrieved document contains instructions to ignore policy, where a web page asks the agent to leak data through a tool call, where an email asks it to forward internal context, and where a tool result carries hostile instructions. Add cases where private data is available but not needed for the task, where a high-risk tool call is proposed without the required approval, and where a memory write attempts to store sensitive or untrusted data. In each, the correct behavior is refusal, escalation, redaction, or a safer route.

Then measure what those cases reveal: blocked unsafe tool calls, allowed safe tool calls, false denials that stop valid work, private-data exposure, approval-routing accuracy, trace completeness, and incident-to-eval conversion rate. Run these against mocked tools so you learn whether the system would have called the dangerous tool before any real side effect is possible.

## Design Checklist

Before an agent can touch private data, untrusted content, or external communication, answer these:

- Which tools can read private data?
- Which tools can fetch untrusted content?
- Which tools can communicate externally?
- Which routes combine those capabilities?
- Which tool calls require approval?
- Where is policy enforced in code?
- What happens when capability metadata is missing?
- Can the system deny by default?
- Can operators disable a route or tool quickly?
- Are traces redacted but still useful?
- Can a production incident become an eval case?

If the answer to “where is this enforced?” is “in the prompt,” the design is not finished.

## Related Chapters

- [Agent Security and Sandboxing](#)
- [Policy Enforcement](#)
- [Human Approval Gates](#)
- [MCP-first Tool Use](#)
- [Tool Capability Design](#)
- [Context Budgets and Working Sets](#)
- [Evaluation-Driven Agent Development](#)
- [Observability and Evals](#)

# Agent Security and Sandboxing

Agent security is different from chatbot safety because agents can act. They can read private data, call APIs, write files, run code, trigger workflows, and communicate with other systems.

Use this chapter when an agent has tools, memory, external data, or side effects.

Start with the [Agent Threat Model](#) if you need to classify the system's risk. Use this chapter when you are ready to design the containment controls.

This chapter belongs after harnesses and production runtime because security is not a prompt feature. The same boundaries that make an agent operable also make it securable: identity, authorization, sandboxing, egress, approvals, traceability, and incident response.

## Security Model

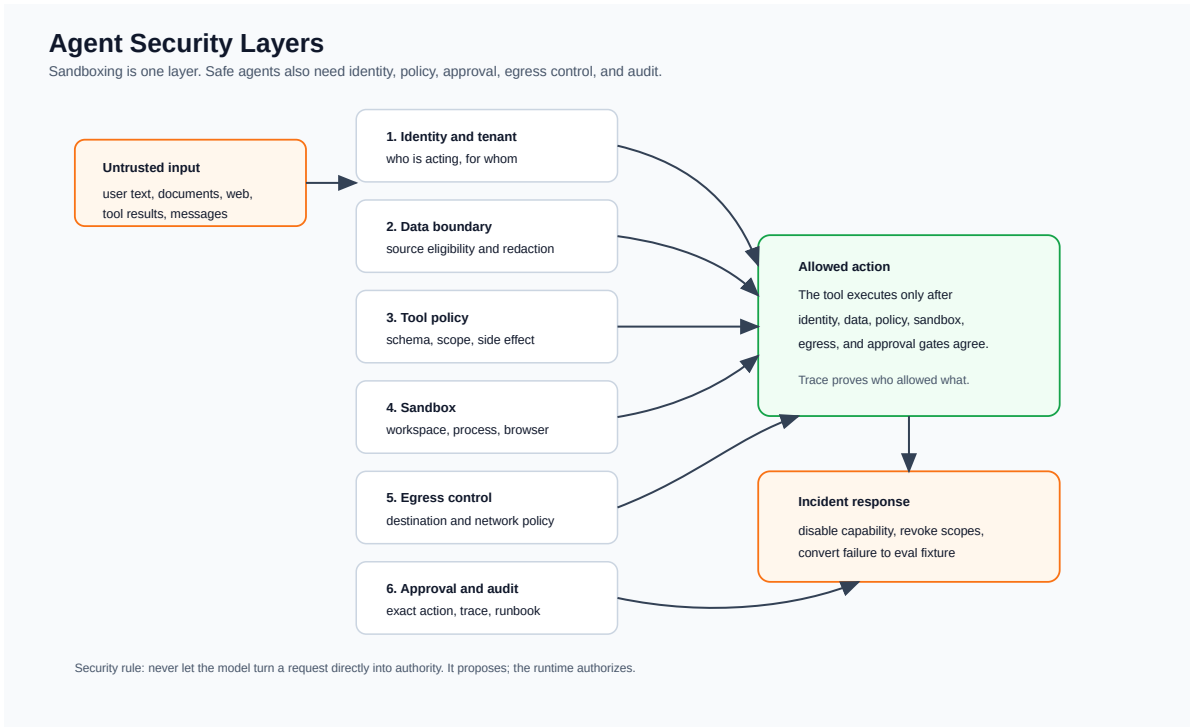
Secure agents by separating four concerns: what the user asks for, what the model proposes, what policy allows, and what the tool actually executes. The model should not be the policy engine. It can classify intent or explain a decision, but deterministic software should enforce permissions.

In production there is a fifth concern: how the request crosses trust boundaries. If an agent calls another agent, a tool gateway, an MCP server, a browser worker, a workflow engine, or a data service, the system needs secure transport, caller identity, scoped authorization, and trace correlation. TLS, mTLS, OAuth or OIDC, policy enforcement, and observability are part of the same boundary.

The model can propose work. It should not mint credentials, choose its own scopes, decide that TLS is optional, approve its own network egress, or hide security events from traces.

## Security Layers

Use this diagram to see sandboxing as one layer in a larger security boundary. The model proposes work; identity, data access, tool policy, sandboxing, egress controls, approval, audit, and incident response decide what can actually happen.



## Threats To Design For

Agent security is easier to reason about when the threat is concrete. Start with these cases before adding more exotic ones.

| Threat            | How it shows up                                                                                   | Control that should catch it                                                  |
|-------------------|---------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| Prompt injection  | A document, email, page, or tool result tells the agent to ignore policy or leak data.            | Source isolation, instruction hierarchy, retrieval guardrails, output checks. |
| Tool abuse        | The model selects a legitimate tool for an unsafe purpose.                                        | Tool policy, scopes, approvals, side-effect classification.                   |
| Secret leakage    | A token, key, customer record, or internal note enters the prompt, memory, logs, or final answer. | Secret handles, redaction, data minimization, trace classification.           |
| Data exfiltration | Private data leaves through browser, API, email, file write, or cross-agent message.              | Egress policy, destination classification, approval gates, audit records.     |
| Confused deputy   | The agent uses a user's authority or a service token in a context where it should not.            | Delegated scopes, audience checks, tenant checks, route-level permissions.    |
| Unsafe delegation | One agent hands work to another agent with broader tools, unclear ownership, or weak identity.    | A2A authorization, scoped task envelopes, trace-linked handoffs.              |

| Threat                     | How it shows up                                                                    | Control that should catch it                                                  |
|----------------------------|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| Replay or duplicate action | A message, tool call, or workflow step runs twice.                                 | Idempotency keys, nonces, expiry, durable side-effect records.                |
| Sandbox escape             | Code, browser automation, or file handling reaches outside the intended workspace. | Filesystem isolation, network blocks, process limits, no ambient credentials. |

Do not treat these as abstract risks. Turn them into tests, traces, and operational runbooks.

## The High-Risk Combination

The most dangerous agent shape combines access to private or trusted data, exposure to untrusted content, and the ability to perform external actions. When those three meet, a malicious document, web page, email, or tool result can try to steer the agent into leaking data or taking an unsafe action.

Mitigate this with least privilege, content isolation, approval gates, egress controls, and explicit policy checks.

## Security Control Matrix

Use security controls by boundary, not as one generic “guardrail” layer.

| Boundary                  | Required Controls                                                                                    |
|---------------------------|------------------------------------------------------------------------------------------------------|
| Data access               | Tenant checks, source eligibility, row or document permissions, data minimization, redaction.        |
| Tool execution            | Typed schemas, policy gate, scoped credentials, idempotency key, timeout, audit record.              |
| Network egress            | Allowlisted domains, blocked private networks, destination classification, TLS verification.         |
| Cross-agent messages      | TLS or mTLS, OAuth or OIDC claims, audience checks, scopes, replay protection, trace ID.             |
| Browser or code execution | Container or VM isolation, restricted filesystem, no ambient credentials, CPU and wall-clock limits. |
| Memory writes             | Source, privacy class, expiry, confidence, correction path, policy decision.                         |
| Human approval            | Proposed action, evidence, policy result, approver identity, timestamp, final tool call.             |
| Observability             | Redacted traces, policy decisions, identity claims, tool calls, egress attempts, stop reasons.       |

The matrix makes one thing visible: sandboxing is necessary, but it is not enough. A sandbox limits the blast radius of execution. It does not replace identity, authorization, policy, approval, egress control, or observability.

## Sandboxing

Sandbox any agent that can execute code, operate a browser, manipulate files, or call write APIs.

Sandbox controls:

- isolated process, container, VM, or browser profile;
- read-only filesystem by default;
- scoped workspace directory;
- no ambient credentials;
- explicit secret injection only for approved tools;
- outbound network restrictions;
- time and CPU limits;
- file size limits;
- audit logs for every side effect;
- cleanup after run completion.

Coding and computer-use agents need stronger sandboxes than read-only research agents.

## Sandbox Tiers

Not every agent needs the same sandbox. Match containment to the work.

| Tier           | Use for                                                              | Minimum controls                                                                             |
|----------------|----------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| Read-only      | Search, summarization, classification, retrieval over approved data. | No write tools, no secrets in context, traceable retrieval, output redaction.                |
| Tool-limited   | Business workflows with typed tools and bounded side effects.        | Per-tool scopes, policy checks, idempotency keys, approvals for writes.                      |
| Workspace      | Coding, document generation, data transformation, local file edits.  | Scoped workspace, read-only source mounts where possible, diff review, cleanup.              |
| Browser        | Web navigation, form filling, UI automation.                         | Separate browser profile, blocked private networks, download controls, credential isolation. |
| Code execution | Shell, notebooks, package install, generated code, tests.            | Container or VM boundary, CPU and time limits, network policy, no ambient secrets.           |

The tier is a product decision. A support draft agent should not silently become a browser automation agent because a tool was convenient.



## Identity And Credentials

Agents should not run with ambient platform credentials. Every credential should be attached to a route, task, tool, user, tenant, and policy decision.

Use:

- short-lived tokens instead of long-lived secrets;
- OAuth or OIDC claims for user and service identity;
- audience checks so a token for one service cannot call another;
- scopes that match the delegated capability;
- mTLS for service-to-service identity where the platform supports it;
- per-tool credential injection after policy approval;
- credential revocation as part of incident response;
- redaction so credentials never enter prompts, memory, eval fixtures, or traces.

The runtime should be able to answer which identity called which tool with which scopes and which policy version allowed it. If it cannot, the credential model is not ready for production agents.

## Network And Egress

Network access is a tool, not a default right. A research agent that can browse the public web should not automatically be able to call internal APIs, metadata services, private networks, customer webhooks, or arbitrary domains.

Useful egress controls:

- allowlist domains by route and tool;
- block private IP ranges and cloud metadata endpoints;
- require TLS certificate validation for remote calls;
- separate browsing from authenticated business APIs;
- classify destinations as internal, partner, customer, public, or unknown;
- log denied egress attempts with trace IDs;
- require approval for new destinations or high-risk data export;
- fail closed when destination classification is missing.

Egress is where data leaks become real. Treat it as part of the policy boundary.

## Access Control

Grant access by role, task, and route. A support-answer agent can read public docs but cannot issue refunds. A billing workflow can read invoice state but needs approval to apply credit. A coding agent can edit files in a branch but cannot deploy to production. A research agent can browse the web but cannot touch customer data. Avoid global tool lists. Each route or agent should receive only the tools needed for the task.

Access control should check both the user and the agent route. A user may have permission to do something manually, but that does not mean every agent acting for that user should inherit the full permission set. Delegation should be narrower than the user's authority.

## Authorization Code Example

Keep authorization outside the prompt. The model can request a tool call, but the runtime decides whether that call can execute.

```
type ToolRequest = {
 actorId: string;
 tenantId: string;
 route: string;
 tool: string;
 sideEffect: "read" | "draft" | "write" | "external_send";
 destination?: string;
 idempotencyKey?: string;
};

type RuntimeClaims = {
 subject: string;
 tenantId: string;
 audience: string;
 scopes: string[];
 expiresAt: number;
};

type ToolPolicy = {
 requiredAudience: string;
 allowedTenants: string[];
 toolScopes: Record<string, string>;
 approvalRequiredFor: string[];
 allowedDestinations: string[];
};

type AuthorizationDecision =
 | { allowed: true; reason: "allowed" }
 | { allowed: false; reason: string };

function authorizeToolRequest(
 request: ToolRequest,
 claims: RuntimeClaims,
 policy: ToolPolicy,
 now = Date.now())
```

```

): AuthorizationDecision {
 if (claims.expiresAt <= now) {
 return { allowed: false, reason: "expired_token" };
 }

 if (claims.audience !== policy.requiredAudience) {
 return { allowed: false, reason: "wrong_audience" };
 }

 if (claims.tenantId !== request.tenantId ||
!policy.allowedTenants.includes(request.tenantId)) {
 return { allowed: false, reason: "tenant_boundary" };
 }

 const requiredScope = policy.toolScopes[request.tool];
 if (!requiredScope || !claims.scopes.includes(requiredScope)) {
 return { allowed: false, reason: "missing_scope" };
 }

 if (request.sideEffect !== "read" && !request.idempotencyKey) {
 return { allowed: false, reason: "missing_idempotency_key" };
 }

 if (request.destination && !policy.allowedDestinations.includes(request.destination)) {
 return { allowed: false, reason: "egress_denied" };
 }

 if (policy.approvalRequiredFor.includes(request.tool)) {
 return { allowed: false, reason: "approval_required" };
 }

 return { allowed: true, reason: "allowed" };
}

```

The real implementation should validate signed tokens and certificates with platform libraries. The architectural point is simpler: authorization happens before execution, from trusted runtime claims, not from model text.

## Guardrails

Guardrails should run before and after model calls and before tools.

| Guardrail Point | Checks                                                                     |
|-----------------|----------------------------------------------------------------------------|
| Input           | prompt injection, sensitive data, unsupported request, user authorization. |
| Retrieval       | untrusted source, stale source, tenant boundary, citation quality.         |
| Tool intent     | permission, side effect level, policy, approval requirement.               |
| Tool input      | schema, allowed fields, data minimization, idempotency key.                |
| Output          | data leakage, unsupported claims, unsafe instructions, missing caveats.    |
| Memory write    | privacy, source, expiry, correction path.                                  |

No single guardrail is enough. Use layers.

## Secrets

Agents should not see raw secrets unless the tool contract requires them. Prefer server-side tool execution, scoped tokens, and short-lived credentials, with secret access granted per tool, no secrets in prompts or memory, and redacted traces. If a model can read a secret, assume it can accidentally expose it.

Do not pass secrets through natural-language context. Give the model a capability handle, not the secret itself. The tool gateway can resolve that handle after policy passes.

## Security Profile Example

A production agent should have an explicit security profile. The exact format can vary, but the runtime needs these decisions somewhere outside the prompt:

```
{
 "route": "support_refund_assistant",
 "risk_class": "medium",
 "identity": {
 "allowed_issuers": ["https://identity.example.com"],
 "required_audience": "support-agent-runtime",
 "required_scopes": ["orders:read", "refunds:draft"]
 },
 "sandbox": {
 "filesystem": "read_only",
 "workspace": "/runs/${run_id}",
 "network": {
 "allowed_domains": ["api.example.com", "docs.example.com"],
 "block_private_networks": true,
 "require_tls": true
 }
 },
}
```

```

 "limits": {
 "max_runtime_ms": 60000,
 "max_output_bytes": 200000
 },
 "tools": {
 "allowed": ["orders.lookup", "refunds.draft_refund", "policies.current_refund_policy"],
 "approval_required": ["refunds.issue_refund", "email.send_customer"]
 },
 "observability": {
 "required_trace_fields": ["trace_id", "actor", "tool", "policy_decision",
"stop_reason"],
 "redact": ["access_token", "refresh_token", "customer_email", "payment_token"]
 }
 }
}

```

This profile is deliberately boring. That is the point. A boring profile can be reviewed, tested, diffed, rolled back, and attached to an incident.

## Approval Gates

For detailed approval design, use [Human Approval Gates](#). In this chapter, approval is treated as a security control for high-risk authority.

Require approval for:

- money movement;
- account changes;
- customer communication at scale;
- production infrastructure changes;
- legal, medical, or compliance decisions;
- deletion or irreversible writes;
- broad data export;
- privilege escalation.

Approval records should include the proposed action, evidence, policy result, approver, timestamp, and final tool call.

## Security Observability

Security controls only help operators if they are visible. Record security events as first-class trace events:

- identity verification result;
- token issuer, audience, subject, tenant, and scopes after redaction;
- TLS or mTLS service identity for cross-service calls;

- policy decision, reason, and policy version;
- sandbox start, limits, denied filesystem access, and denied network egress;
- credential injection and credential refusal;
- approval request, approval decision, and approver identity;
- tool invocation, side-effect identifier, idempotency key, and stop reason;
- redaction action and trace retention class.

Do not turn observability into a data leak. Raw tokens, private keys, secrets, full customer records, and unnecessary payloads should not appear in traces. The goal is to explain authority and behavior, not to duplicate sensitive data.

## Security Evals

Test security controls before launch and after meaningful changes.

| Eval Case                              | Expected Behavior                                                       |
|----------------------------------------|-------------------------------------------------------------------------|
| Missing OAuth scope                    | Tool call is denied before execution and denial is traced.              |
| Wrong token audience                   | Cross-agent request is rejected before payload processing.              |
| Prompt injection in retrieved document | Agent treats content as data and does not change policy or destination. |
| Unapproved egress destination          | Network call is blocked and recorded with trace ID.                     |
| Memory write contains private data     | Write is denied, redacted, or routed for approval.                      |
| Tool requests broad credential         | Runtime injects only scoped credential or denies the call.              |
| Approval-required side effect          | Workflow pauses before execution and records approval context.          |
| Replay of old message                  | Idempotency or nonce check blocks duplicate action.                     |
| Browser tries private network          | Request is denied before navigation or upload.                          |
| Code tries to read outside workspace   | Filesystem access is denied and traced.                                 |
| Subagent asks for broader tools        | Delegation is refused or routed for approval.                           |
| Trace contains raw token               | Redaction test fails and blocks release.                                |

Measure false allows, blocked unsafe tool calls, redaction failures, missing trace fields, unauthorized egress attempts, approval-routing accuracy, sandbox escape attempts, and time to investigate a security incident.

## Incident Response

Plan for agent incidents before launch. The team should already know how to disable a tool or a route, roll back a prompt or policy, revoke credentials, quarantine memory, inspect traces, notify affected users, and turn the incident into evals. If the response requires code archaeology in the middle of an incident, the system is not operationally ready.

Security incident response should include identity and credential actions: revoke tokens, rotate secrets, disable service accounts, invalidate message nonces, quarantine affected traces, and preserve enough audit evidence to explain the run.

## Related Chapters

- [Agent Threat Model](#)
- [Production Runtime Overview](#)
- [Policy Enforcement](#)
- [Observability and Evals](#)
- [Human Approval Gates](#)
- [Secure Agent Communication](#)
- [MCP-first Tool Use](#)
- [Circuit Breakers, Fallbacks, and Replay](#)
- [Coding Agents](#)

# Agent UX and Human Trust

Agent UX is the interface between human judgment and machine autonomy. A good agent interface shows progress, asks for help at the right time, and makes its actions inspectable.

Use this chapter when humans assign goals, review outputs, approve actions, or recover failed runs.

The core contract is simple: the user should understand what the agent is trying to do, what it has seen, what it has changed, what it is about to do, and how to interrupt or correct it. Trust is not a feeling the UI creates with friendly copy. Trust is the result of bounded autonomy, visible state, and usable control.

Download the reusable worksheet: [agent UX review worksheet](#).

## UX Goals

An agent experience should help users answer:

- What is the agent trying to do?
- What has it already done?
- What evidence is it using?
- What will it do next?
- What needs my approval?
- What failed?
- How do I correct it?
- How do I stop it?

Trust comes from control and visibility, not from confident language.

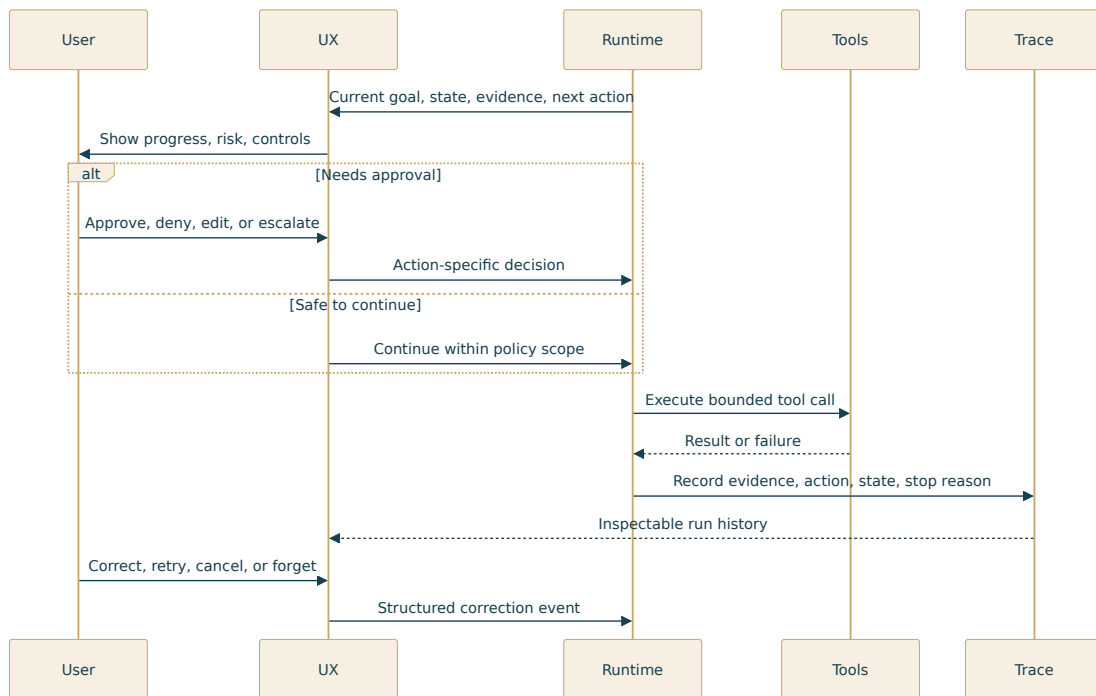
## Trust Contract

An agent interface should make five things visible enough for the risk of the task:

| Question               | UX Obligation                                                                      |
|------------------------|------------------------------------------------------------------------------------|
| What is the goal?      | Show the active goal, success criteria, and owner.                                 |
| What does it know?     | Show evidence, memory used, tool results, and known gaps.                          |
| What can it do?        | Show available authority, pending side effects, and approval requirements.         |
| What is happening now? | Show current state, step, wait reason, and progress.                               |
| How do I intervene?    | Provide cancel, pause, approve, deny, edit, retry, escalate, and correction paths. |



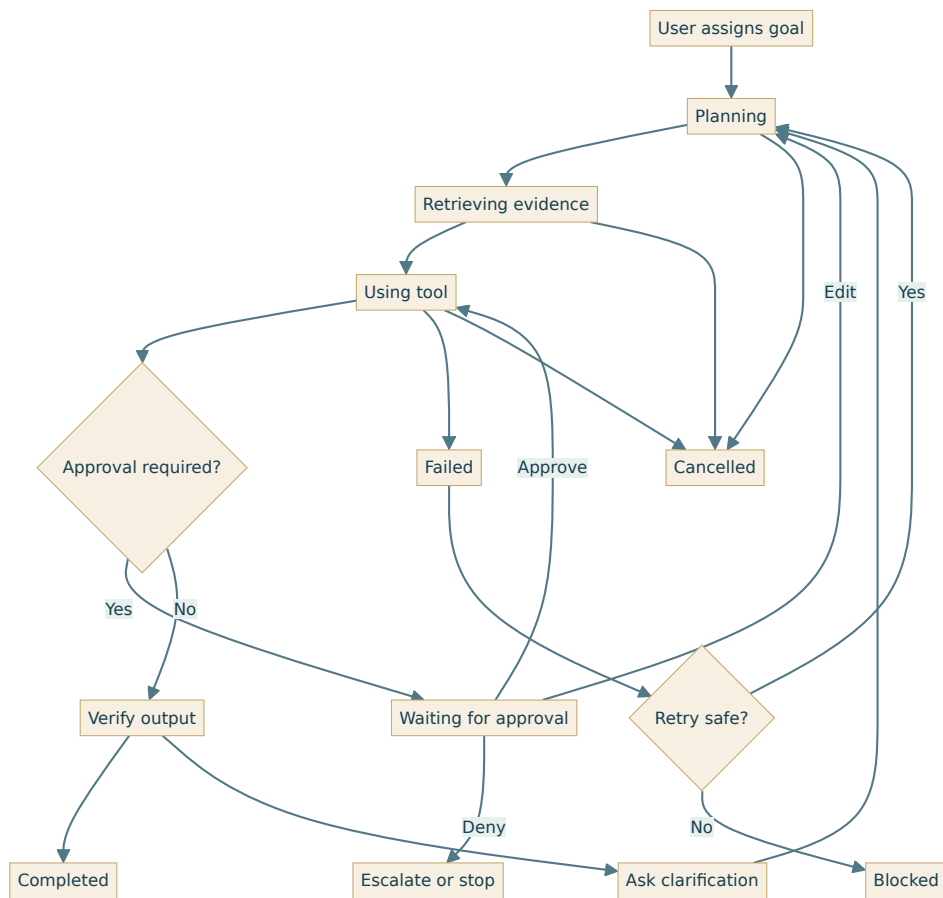
This does not mean exposing every internal detail. It means exposing the operational facts a reasonable user needs to supervise the agent.



Use this loop as a UX test. If the interface cannot show the state, expose the decision, bind approval to one action, and record the correction, the user is supervising a black box.

## Supervision State Model

The interface should expose the same state model the runtime uses. The words can change for the product, but the state transitions should not be hidden behind one spinner.



Use the model as a UI checklist. Each state needs visible evidence, available controls, stop rules, and trace events. A state that cannot be shown cannot be supervised.

## Interaction Modes

| Mode         | Use When                                        | UX Requirement                                      |
|--------------|-------------------------------------------------|-----------------------------------------------------|
| Interactive  | The user is present and can clarify or approve. | Streaming progress, questions, cancel, approval UI. |
| Background   | Work takes minutes or hours.                    | Status page, notifications, resume, audit trail.    |
| Human review | The agent prepares work for approval.           | Diff, evidence, rationale, accept/edit/reject.      |
| Autonomous   | The agent runs within bounded authority.        | Policy scope, traceability, alerts, rollback.       |
| Multi-agent  | Several agents collaborate.                     | Role visibility, handoff status, final owner.       |

The UX should match the risk. A read-only research agent can be lightweight. A production-deploying coding agent needs stronger review and rollback.

## UX States

Treat agent states as product states, not spinner text.

| State                | What The User Should See                                      |
|----------------------|---------------------------------------------------------------|
| Planning             | Goal, route, assumptions, and next step.                      |
| Retrieving           | Sources being searched, filters, and evidence count.          |
| Using tool           | Tool name, target system, purpose, and side-effect class.     |
| Waiting for approval | Proposed action, risk, evidence, approver, and expiry.        |
| Asking clarification | The missing input and why it blocks progress.                 |
| Blocked              | Block reason, completed work, and recovery options.           |
| Escalating           | Who receives the handoff and what context is included.        |
| Completed            | Result, evidence, changed systems, and trace ID.              |
| Failed               | Failure class, external changes, retry options, and trace ID. |
| Cancelled            | What stopped, what was rolled back, and what remains.         |

These states should map to runtime state. If the UI says “working” while the runtime is waiting for approval, the system is hiding the most important fact.

## Progress Design

Show progress at meaningful boundaries:

- understanding the goal;
- selecting a route;
- retrieving evidence;
- calling tools;
- waiting for external systems;
- evaluating output;
- requesting approval;
- completing or stopping.

Do not stream hidden reasoning as the primary progress signal. Show actions, evidence, and state transitions.

A useful progress event is small and concrete:

```
type AgentUxEvent = {
 runId: string;
 state:
 | "planning"
 | "retrieving"
```

```

 | "using_tool"
 | "waiting_for_approval"
 | "asking_clarification"
 | "blocked"
 | "escalating"
 | "completed"
 | "failed"
 | "cancelled";
 title: string;
 detail?: string;
 evidenceRefs: string[];
 toolCallId?: string;
 approvalId?: string;
 traceId: string;
 userActions: Array<"cancel" | "pause" | "approve" | "deny" | "edit" | "retry" |
"escalate" | "inspect">;
};

```

The UI does not need to show the raw event. But the product should be built from events like this, so progress, traceability, and recovery stay aligned.

## User Controls

The controls should match the agent's authority:

- **Cancel:** stop the run and prevent future side effects.
- **Pause:** stop after the current safe boundary.
- **Approve:** allow one exact action, not broad future autonomy.
- **Deny:** stop or route to a safer alternative.
- **Edit:** change a draft, extracted field, plan step, or proposed action.
- **Retry:** repeat a safe step with the same evidence or a revised route.
- **Inspect:** open evidence, tool results, memory, policy decision, or trace.
- **Forget:** remove or reject a memory write.
- **Escalate:** hand off to a human or higher-trust workflow.

Controls are part of the architecture. A cancel button that cannot stop a queued tool call is not a real cancel button.

## Explainability

Users need actionable explanations.

Good explanations include:

- source evidence;

- tool results;
- policy checks;
- uncertainty;
- alternative paths considered when relevant;
- reason for escalation or refusal.

Avoid explanations that expose private prompts, irrelevant internal chains of thought, or vague claims such as “the model decided.”

## Visibility Rules

Show the user the right level of detail for the task risk.

| Surface  | Low-Risk Task                | High-Risk Task                                           |
|----------|------------------------------|----------------------------------------------------------|
| Goal     | short summary.               | goal, constraints, owner, success criteria.              |
| Evidence | citation links.              | citation links, source age, conflicts, missing evidence. |
| Tools    | hidden or summarized.        | tool name, target system, side effect, result.           |
| Memory   | relevant preference if used. | memory source, age, scope, correction or forget option.  |
| Policy   | usually hidden.              | policy reason, approval requirement, denial reason.      |
| State    | lightweight progress.        | step, stop reason, retry state, approval state.          |

High-risk does not always mean dramatic. Sending an email, changing access, issuing a refund, writing memory, deleting a file, or touching production all deserve more visibility than answering a documentation question.

## Corrections

Design correction paths before users need them.

Correction types:

- edit final answer;
- correct extracted fields;
- change route;
- add missing context;
- reject memory write;
- retry with a different tool;
- escalate to human;
- cancel and roll back.

Corrections should update evals and, where appropriate, memory. They should not disappear into chat history.

A correction is data. Record what was corrected, who corrected it, which run it affected, whether memory changed, and whether the eval suite needs a new case.

## Multi-Agent UX

Multi-agent systems need role clarity.

Show:

- which agent owns the task;
- which agents contributed;
- what each role produced;
- where handoffs occurred;
- which agent or workflow made the final decision;
- where a human entered the loop.

If users cannot tell who is responsible, the system will feel unreliable even when the output is correct.

## Approval UX

Approval requests should be specific.

An approval should show:

- proposed action;
- target system;
- affected data or user;
- evidence;
- risk level;
- policy result;
- reversible or irreversible status;
- expected result;
- approval and rejection options.

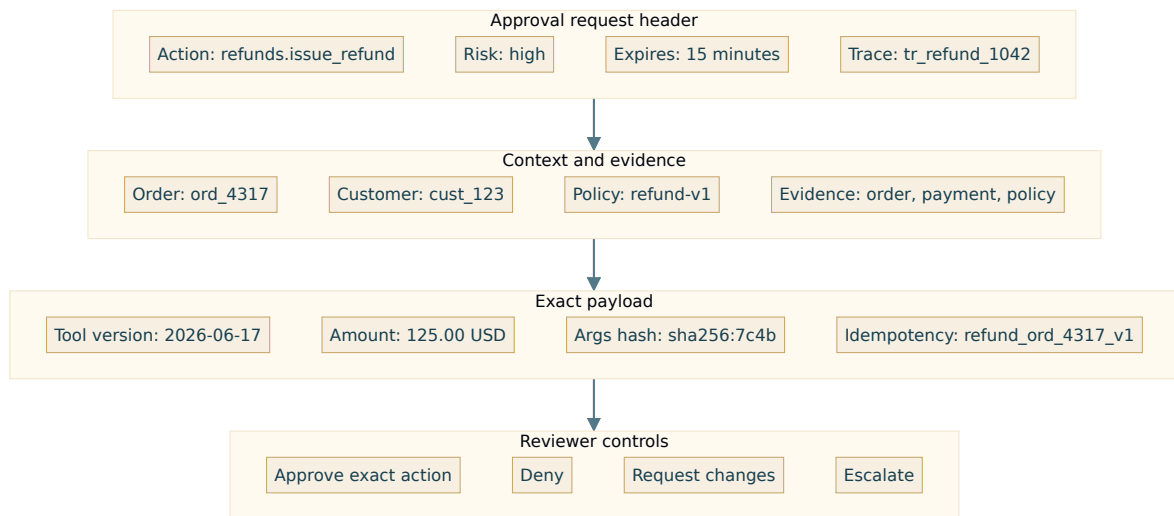
Never ask for broad approval such as “continue with all actions” when the agent can perform high-risk side effects.

Approvals should be attached to exact actions. If the proposed recipient, amount, file, command, permission, memory, or payload changes, the approval should no longer apply.

For the underlying approval contract, see [Human Approval Gates](#). This chapter focuses on what the user must be able to see and decide.

## Approval Panel Mock

Use this panel model when reviewing high-risk actions. The layout can change, but the evidence, action, risk, and decision controls should not disappear.



An approval panel should make the risky action boringly explicit. The reviewer should see the same action ID, arguments hash, policy version, resource IDs, and trace ID that the runtime will validate before resuming.

| Panel Area     | Required Fields                                                   |
|----------------|-------------------------------------------------------------------|
| Header         | approval ID, trace ID, requester, risk class, expiry.             |
| Action         | tool name, tool version, side-effect class, exact resource.       |
| Evidence       | source IDs, policy references, diff or payload, missing evidence. |
| Runtime Safety | idempotency key, rollback note, whether the action is reversible. |
| Decision       | approve exact action, deny, request changes, escalate, cancel.    |

## Failure UX

A failed run should be useful.

Return:

- what completed;
- what failed;
- why it stopped;
- whether anything changed externally;
- what the user can do next;
- link or ID for trace review.

Failure UX is part of reliability. A system that fails clearly can still be trusted.

## Trust Calibration

The UI should not make the agent look more certain or more autonomous than it is.

Use confident language only when the system has strong evidence and the action is low risk. Use uncertainty when evidence is partial, stale, conflicting, or outside the agent's authority. Make refusals and escalations normal outcomes rather than embarrassing failures.

Bad trust calibration looks like:

- “Done” when the agent only drafted something;
- “Verified” when citations were not checked;
- “I found the answer” when sources conflict;
- “Running safely” when the tool has broad write access;
- “Remembered” when the user never approved a memory write.

## Failure Modes

- The agent appears busy but does not show what state it is in.
- Tool use is hidden until after side effects happen.
- Memory is used invisibly and the user cannot correct or forget it.
- Approval requests hide the real action or affected resource.
- The user cannot cancel a run that can still execute side effects.
- Errors are summarized as generic failure messages with no recovery path.
- Multi-agent handoffs hide who owns the final decision.
- Progress messages imply certainty while evidence is weak.
- The UI shows a final answer but not the trace, citations, or changed systems.
- Corrections disappear into chat and never improve evals or memory.

## Evaluation Guidance

Agent UX needs evals too. Do not only ask whether the answer is correct. Ask whether the user could supervise the agent.

- Test whether users can identify the active goal and current state.
- Test whether approval requests expose the exact action and side effects.
- Test whether cancellation stops future side effects.
- Test whether corrections update the right artifact: answer, state, memory, or eval.
- Test whether users can inspect evidence and tool results.
- Test whether memory use is visible and correctable.
- Test whether failure messages explain what changed externally.
- Test whether trust language matches evidence strength.
- Test whether multi-agent handoffs show ownership.
- Test whether a user can recover from blocked, failed, or escalated runs.

Measure task success, correction success, approval clarity, cancellation success, evidence inspection rate, user override rate, memory correction rate, failure recovery rate, and trust calibration.

## Production Checklist



- Map runtime states to explicit UI states.
- Show goal, current step, evidence, tool calls, approvals, and stop reason.
- Provide cancel, pause, approve, deny, edit, retry, inspect, forget, and escalate controls where relevant.
- Bind approvals to exact actions.
- Make memory use visible and correctable.
- Show changed external systems after side effects.
- Treat refusal, cancellation, approval wait, and escalation as normal states.
- Record corrections as structured events.
- Feed user corrections, overrides, and failures into evals.
- Keep trace IDs accessible for support and incident review.

## Related Chapters

- [Human Approval Gates](#)
- [Goals and State](#)
- [Tool Capability Design](#)
- [Memory-Augmented Agent](#)
- [Routing and Handoffs](#)
- [Observability and Evals](#)
- [Agent Development Lifecycle](#)

# Planning and Execution

Planning separates deciding what to do from doing it. The planner creates steps; the executor runs them, reports progress, and handles errors.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

Planning separates deciding what to do from doing it. The planner creates steps; the executor runs them, reports progress, and handles errors.

## Use When

- The task has meaningful sequencing, dependencies, or recoverable failure points.
- You want to inspect or revise the plan before execution.
- Execution can be deterministic even if planning uses a model.

## Avoid When

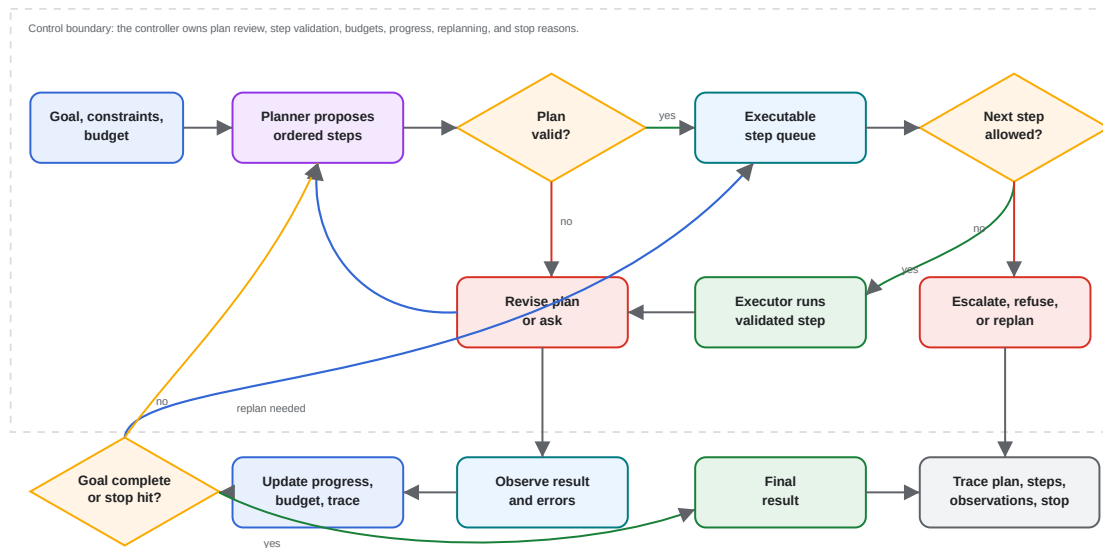
- The plan would always be a single step.
- The executor cannot report structured progress or failure.
- The model is allowed to execute unvalidated plans directly.

## Architecture

Use this diagram to read Planning and Execution as a system boundary, not only a code shape. The key ownership question is: the loop controller owns progress, budgets, stop conditions, and recovery state.

## Planning and Execution Control Flow

Planning chooses an ordered path; execution validates and observes each step before continuing.



Read it as a controlled handoff: the planner proposes steps, the controller validates the plan, and the executor runs only validated steps with observable progress.

## System Shape

- **Pattern boundary:** a controller repeatedly chooses the next step, executes it, observes the result, and decides whether to continue.
- **State owner:** the loop controller owns progress, budgets, stop conditions, and recovery state.
- **Primary artifact:** `planning-pattern/` contains the runnable reference implementation and examples.
- **Operational promise:** Planning separates deciding what to do from doing it. The planner creates steps; the executor runs them, reports progress, and handles errors.
- **Runnable path:** start with `npm run plan:test` before adapting the pattern to a larger system.

## Core Protocol

1. Initialize goal state, constraints, budgets, and stop conditions.
2. Choose the next action from the current state instead of assuming the whole path upfront.
3. Execute the action through a validated tool, worker, or local function.
4. Observe the result and update state with evidence, errors, and remaining work.
5. Stop, retry, re-plan, or escalate according to explicit policy.

## Implementation Notes

- Keep the pattern boundary explicit: inputs, state, side effects, and outputs should be visible.
- Validate model-produced decisions before they affect tools, users, or durable state.
- Emit enough trace data to debug failures after the run.

## Failure Modes

- The pattern is applied where a simpler deterministic workflow would be better.
- State, tool calls, or model decisions are not observable enough to debug.
- The system lacks clear stop, retry, or escalation behavior.

## Evaluation Strategy

- Test success cases, partial failure, repeated failure, budget exhaustion, and bad intermediate observations.
- Assert that the loop stops for the right reason and does not hide failed steps.
- Measure completion rate, number of iterations, recovery quality, cost, and latency.
- Include cases that prove each “Use When” condition is true for this pattern.
- Include negative cases from “Avoid When” so the system chooses a simpler or safer pattern when appropriate.

## Production Checklist

- Set hard iteration, cost, and time limits.
- Persist state after meaningful steps if the run can be interrupted.
- Make retries idempotent or add compensation.
- Expose trace events for each decision, action, observation, and stop reason.
- Define human escalation for ambiguous, high-risk, or policy-blocked work.
- Keep the source bundle, generated chapter, tests, and deployment artifact in the same release.

## Run the Example

```
npm run plan:test
npm run plan:run -- "Compute average of [1,2,3,4]"
npm run plan:py
```

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
planning-pattern/typescript/src/planner.ts
```

## Open full source

```
import axios from 'axios';

const MISTRAL_API = 'https://api.mistral.ai/v1/chat/completions';

export interface PlanStep { id: string; description: string }
export interface Plan { steps: PlanStep[]; rationale: string }

function extractNumbers(goal: string): number[] {
 const bracketed = goal.match(/\s*([^\s]+)\s*/)?.[1];
 if (!bracketed) return [1, 2, 3, 4];

 const numbers = bracketed
 .split(',')
 .map(value => Number(value.trim()))
 .filter(Number.isFinite);

 return numbers.length > 0 ? numbers : [1, 2, 3, 4];
}

export async function planTask(goal: string, apiKey?: string): Promise<Plan> {
 if (!apiKey) {
 const numbers = extractNumbers(goal);
 // deterministic fallback plan (no network) for tests
 return { steps: [
 { id: 's1', description: `Load numbers [${numbers.join(',')}]` },
 { id: 's2', description: 'Compute average' },
], rationale: 'synthetic' };
 }

 const resp = await axios.post(MISTRAL_API, {
 model: 'mistral-small-latest',
 messages: [
 { role: 'system', content: 'Return JSON {steps: [{id, description}], rationale} for the goal.' },
 { role: 'user', content: goal }
],
 temperature: 0
 }, { headers: { Authorization: `Bearer ${apiKey}` } });
 const content = resp.data?.choices?.[0]?.message?.content || '';
 try { return JSON.parse(content); } catch { return { steps: [], rationale: content }; }
}
```

[planning-pattern/typescript/src/executor.ts](#)

```
export type ExecutionFailure = {
 status: "failed";
 error_type: "unsupported_step" | "missing_numbers";
 step_id: string;
 description: string;
};

export type ExecutionValue = number[] | number | ExecutionFailure;
export type ExecutionResults = Record<string, ExecutionValue>;

export async function executePlan(steps: { id: string; description: string }[],
onProgress?: (pct: number, stage: string) => void): Promise<ExecutionResults> {
 const results: ExecutionResults = {};
 for (let i = 0; i < steps.length; i++) {
 const s = steps[i];
 onProgress?.(Math.round((i / steps.length) * 100), s.id);
 // trivial synthetic execution
 if (s.description.includes('Load numbers')) {
 const raw = s.description.match(/\s*([\^\s]+)\s*/)?.[1] ?? '';
 results[s.id] = raw
 .split(',')
 .map(value => Number(value.trim()))
 .filter(Number.isFinite);
 }
 else if (s.description.includes('Compute average')) {
 const arr = Array.isArray(results['s1']) ? results['s1'] : [];
 results[s.id] = arr.length > 0
 ? arr.reduce((a:number,b:number)=>a+b,0)/arr.length
 : {
 status: "failed",
 error_type: "missing_numbers",
 step_id: s.id,
 description: s.description
 };
 }
 else results[s.id] = {
 status: "failed",
 error_type: "unsupported_step",
 step_id: s.id,
 description: s.description
 };
 }
 onProgress?.(100, 'done');
```

```
 return results;
}
```

`planning-pattern/typescript/src/run.ts`

[Open full source](#)

```
import { planTask } from './planner.ts';
import { executePlan } from './executor.ts';

async function main() {
 const goal = process.argv.slice(2).join(' ') || 'Compute average of [1,2,3,4]';
 const plan = await planTask(goal, process.env.MISTRAL_API_KEY);
 console.log('Plan:', plan);
 const results = await executePlan(plan.steps, (pct, stage) => console.log('Progress',
pct, stage));
 console.log('Results:', results);
}

main();
```

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `planning-pattern/` folder from this repository.

## Related Patterns

- [ReAct](#)
- [Reflection](#)
- [Evaluator-Optimizer](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# ReAct

ReAct alternates reasoning and acting. The agent reasons about current state, takes an action, observes the result, and repeats.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

ReAct alternates reasoning and acting. The agent reasons about current state, takes an action, observes the result, and repeats.

## Use When

- The task requires tool use and the agent cannot know all required information upfront.
- Observations should change the next step.
- A bounded loop can stop on success, failure, or budget.

## Avoid When

- The task is a deterministic workflow with known steps.
- You cannot validate actions before they run.
- The reasoning trace would expose sensitive information to users.

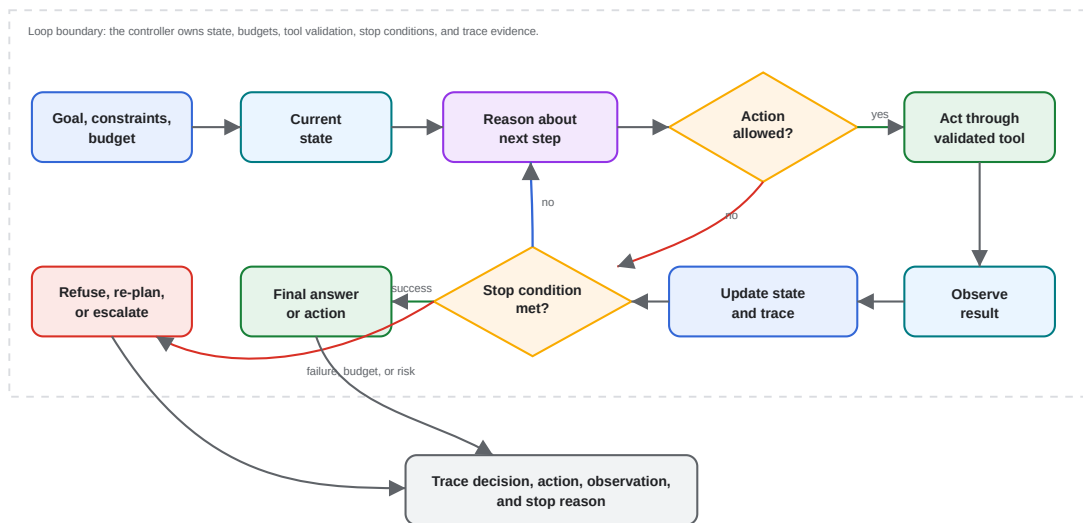
## Architecture

Use this diagram to read ReAct as a system boundary, not only a code shape. The key ownership question is: the loop controller owns progress, budgets, stop conditions, and recovery state.



## ReAct Control Loop

Reason, validate, act, observe, and update state until a stop condition ends the loop.



Read it as a bounded loop: each reasoning step must pass an action gate, each action must produce an observation, and each observation must update state or stop.

## System Shape

- **Pattern boundary:** a controller repeatedly chooses the next step, executes it, observes the result, and decides whether to continue.
- **State owner:** the loop controller owns progress, budgets, stop conditions, and recovery state.
- **Primary artifact:** `react-pattern-reason-act/` contains the runnable reference implementation and examples.
- **Operational promise:** ReAct alternates reasoning and acting. The agent reasons about current state, takes an action, observes the result, and repeats.
- **Runnable path:** start with `npm run react-agent` before adapting the pattern to a larger system.

## Core Protocol

1. Initialize goal state, constraints, budgets, and stop conditions.
2. Choose the next action from the current state instead of assuming the whole path upfront.
3. Execute the action through a validated tool, worker, or local function.
4. Observe the result and update state with evidence, errors, and remaining work.
5. Stop, retry, re-plan, or escalate according to explicit policy.

## Implementation Notes

- Keep the pattern boundary explicit: inputs, state, side effects, and outputs should be visible.
- Validate model-produced decisions before they affect tools, users, or durable state.
- Emit enough trace data to debug failures after the run.

## Failure Modes

- The pattern is applied where a simpler deterministic workflow would be better.
- State, tool calls, or model decisions are not observable enough to debug.
- The system lacks clear stop, retry, or escalation behavior.

## Evaluation Strategy

- Test success cases, partial failure, repeated failure, budget exhaustion, and bad intermediate observations.
- Assert that the loop stops for the right reason and does not hide failed steps.
- Measure completion rate, number of iterations, recovery quality, cost, and latency.
- Include cases that prove each “Use When” condition is true for this pattern.
- Include negative cases from “Avoid When” so the system chooses a simpler or safer pattern when appropriate.

## Production Checklist

- Set hard iteration, cost, and time limits.
- Persist state after meaningful steps if the run can be interrupted.
- Make retries idempotent or add compensation.
- Expose trace events for each decision, action, observation, and stop reason.
- Define human escalation for ambiguous, high-risk, or policy-blocked work.
- Keep the source bundle, generated chapter, tests, and deployment artifact in the same release.

## Run the Example

```
npm run react-agent
```

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
react-pattern-reason-act/autogen_typescript_example/react_agent.ts
```

[Open full source](#)

```

// ReAct Pattern (Reason + Act) - Autogen TypeScript Example
// To run: npm install && npm run react-agent

import axios from 'axios';
import * as readline from 'readline';
import { evaluate } from 'mathjs';
import * as dotenv from 'dotenv';
dotenv.config();

const MISTRAL_API_URL = 'https://api.mistral.ai/v1/chat/completions';
const MISTRAL_API_KEY = process.env.MISTRAL_API_KEY;

function calculatorTool(input: string): string {
 try {
 const val = evaluate(input);
 return val.toString();
 } catch (e) {
 return `Error: ${e}`;
 }
}

async function reactAgent(userInput: string): Promise<string> {
 let context = userInput;
 let done = false;
 let result = '';
 while (!done) {
 // Step 1: Reason
 const reasoningPrompt = `You are an agent. Think step by step about what to do next. If
you need to use a tool, say TOOL: <expression>. If you are done, say FINAL:
<answer>.\nContext: ${context}`;
 const response = await axios.post(
 MISTRAL_API_URL,
 {
 model: 'mistral-tiny',
 messages: [{ role: 'user', content: reasoningPrompt }],
 },
 {
 headers: {
 'Authorization': `Bearer ${MISTRAL_API_KEY}`,
 'Content-Type': 'application/json',
 },
 }
);
 const agentOutput = response.data.choices[0].message.content.trim();
 }
}

```

```

 console.log('Agent:', agentOutput);
 if (agentOutput.startsWith('TOOL:')) {
 const expr = agentOutput.replace('TOOL:', '').trim();
 const toolResult = calculatorTool(expr);
 context += `\nTool result: ${toolResult}`;
 } else if (agentOutput.startsWith('FINAL:')) {
 result = agentOutput.replace('FINAL:', '').trim();
 done = true;
 } else {
 // If the agent doesn't follow the protocol, end loop
 result = agentOutput;
 done = true;
 }
}
return result;
}

async function main() {
 const idx = process.argv.indexOf('--input');
 const cliInput = idx !== -1 ? process.argv[idx + 1] : undefined;
 const nonInteractive = cliInput || process.env.NON_INTERACTIVE_INPUT;
 if (nonInteractive) {
 try {
 const agentResponse = await reactAgent(String(nonInteractive));
 console.log('Final Answer:', agentResponse);
 } catch (err) {
 console.error('Error:', err);
 process.exitCode = 1;
 }
 return;
 }
 const rl = readline.createInterface({ input: process.stdin, output: process.stdout });
 rl.question('Task: ', async (userInput: string) => {
 try {
 const agentResponse = await reactAgent(userInput);
 console.log('Final Answer:', agentResponse);
 } catch (err) {
 console.error('Error:', err);
 }
 rl.close();
 });
}

main();

```

[Open full source](#)

```
ReAct Pattern (Reason + Act) - LangGraph Python Example
```

```
This example demonstrates the ReAct Pattern using LangGraph and Python. The agent alternates between reasoning and acting (using a calculator tool), and iteratively solves the task. The LLM is Mistral.
```

```
Requirements
```

- Python 3.8+
- `langgraph` library
- `python-dotenv` (for .env support)
- Mistral LLM API access

```
Install dependencies
```

```
```bash
pip install langgraph python-dotenv requests
```
```

```
Example Code
```

```
```python
import os
from langgraph import Agent, Environment, LLM, Tool
from dotenv import load_dotenv

load_dotenv()

MISTRAL_API_KEY = os.getenv("MISTRAL_API_KEY")
MISTRAL_API_URL = "https://api.mistral.ai/v1/chat/completions"

import sympy as _sp

def safe_calc(expr: str) -> str:
    try:
        return str(_sp.sympify(expr, evaluate=True))
    except Exception as e:
        return f"Error: {e}"

class CalculatorTool(Tool):
```

```

def call(self, input_str):
    return safe_calc(input_str)

class SimpleEnvironment(Environment):
    def get_observation(self):
        return input("Task: ")
    def send_action(self, action):
        print(f"Agent: {action}")

class ReActAgent(Agent):
    def __init__(self, llm, tools):
        self.llm = llm
        self.tools = {tool.name: tool for tool in tools}
    def act(self, observation):
        context = observation
        while True:
            prompt = (
                "You are an agent. Think step by step about what to do next. "
                "If you need to use a tool, say T00L: <expression>. "
                "If you are done, say FINAL: <answer>.\nContext: " + context
            )
            response = self.llm.complete(prompt)
            print(f"Agent: {response}")
            if response.startswith("T00L:"):
                expr = response.replace("T00L:", "").strip()
                tool_result = self.tools["calculator"].call(expr)
                context += f"\nTool result: {tool_result}"
            elif response.startswith("FINAL:"):
                return response.replace("FINAL:", "").strip()
            else:
                return response

llm = LLM(
    provider="mistral",
    api_key=MISTRAL_API_KEY,
    api_url=MISTRAL_API_URL,
)

calc_tool = CalculatorTool(name="calculator")
env = SimpleEnvironment()
agent = ReActAgent(llm, [calc_tool])

observation = env.get_observation()
action = agent.act(observation)

```

```
env.send_action(action)
```

```
'''
```

```
'''
```

```
- Try a multi-step math or logic task to see reasoning and tool use.
```

```
- Make sure your `.env` file contains your Mistral API key.
```

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `react-pattern-reason-act/` folder from this repository.

Related Patterns

- [Planning and Execution](#)
- [Reflection](#)
- [Evaluator-Optimizer](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

Reflection

Reflection asks an agent or evaluator to inspect prior output and identify concrete improvements.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Reflection asks an agent or evaluator to inspect prior output and identify concrete improvements.

Use When

- The system has explicit quality criteria.
- A critique can change decisions, tests, state, or final artifacts.
- You need a lightweight improvement pass without a full evaluator-optimizer loop.

Avoid When

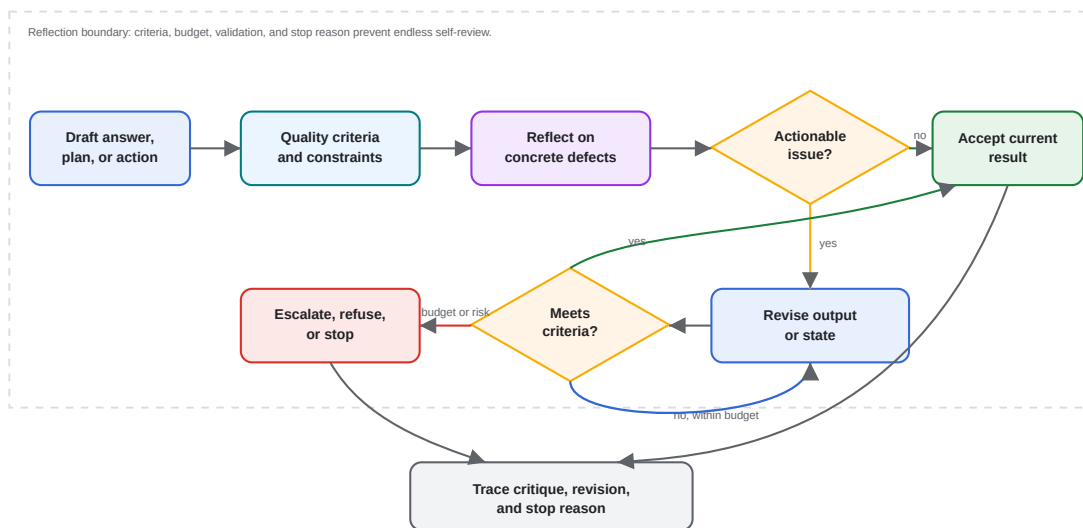
- The critique only produces longer output.
- There is no acceptance criterion for the revised result.
- The model is asked to approve its own unsafe actions.

Architecture

Use this diagram to read Reflection as a system boundary, not only a code shape. The key ownership question is: the loop controller owns progress, budgets, stop conditions, and recovery state.

Reflection Improvement Loop

Reflection is useful only when critique finds concrete defects and a bounded revision can improve the result.



Read it as a bounded improvement pass: critique must identify concrete defects, revision must be validated, and budget or risk must stop the loop.

System Shape

- **Pattern boundary:** a controller repeatedly chooses the next step, executes it, observes the result, and decides whether to continue.
- **State owner:** the loop controller owns progress, budgets, stop conditions, and recovery state.
- **Primary artifact:** `reflection-and-self-improvement-pattern/` contains the runnable reference implementation and examples.
- **Operational promise:** Reflection asks an agent or evaluator to inspect prior output and identify concrete improvements.
- **Runnable path:** start with `npm run reflection-self-improvement-agent` before adapting the pattern to a larger system.

Core Protocol

1. Initialize goal state, constraints, budgets, and stop conditions.
2. Choose the next action from the current state instead of assuming the whole path upfront.
3. Execute the action through a validated tool, worker, or local function.
4. Observe the result and update state with evidence, errors, and remaining work.
5. Stop, retry, re-plan, or escalate according to explicit policy.

Implementation Notes

- Keep the pattern boundary explicit: inputs, state, side effects, and outputs should be visible.
- Validate model-produced decisions before they affect tools, users, or durable state.
- Emit enough trace data to debug failures after the run.

Failure Modes

- The pattern is applied where a simpler deterministic workflow would be better.
- State, tool calls, or model decisions are not observable enough to debug.
- The system lacks clear stop, retry, or escalation behavior.

Evaluation Strategy

- Test success cases, partial failure, repeated failure, budget exhaustion, and bad intermediate observations.
- Assert that the loop stops for the right reason and does not hide failed steps.
- Measure completion rate, number of iterations, recovery quality, cost, and latency.
- Include cases that prove each “Use When” condition is true for this pattern.
- Include negative cases from “Avoid When” so the system chooses a simpler or safer pattern when appropriate.

Production Checklist

- Set hard iteration, cost, and time limits.
- Persist state after meaningful steps if the run can be interrupted.
- Make retries idempotent or add compensation.
- Expose trace events for each decision, action, observation, and stop reason.
- Define human escalation for ambiguous, high-risk, or policy-blocked work.
- Keep the source bundle, generated chapter, tests, and deployment artifact in the same release.

Run the Example

```
npm run reflection-self-improvement-agent
```

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
reflection-and-self-improvement-pattern/autogen_typescript_example/reflection_agent.ts
```

[Open full source](#)

```

import dotenv from 'dotenv';
dotenv.config();
import axios from 'axios';
import readline from 'readline';

const MISTRAL_API_KEY = process.env.MISTRAL_API_KEY;
const MISTRAL_API_URL = 'https://api.mistral.ai/v1/chat/completions';

if (!MISTRAL_API_KEY) {
  console.error('Please set MISTRAL_API_KEY in your .env file');
  process.exit(1);
}

async function askMistral(messages: any[]) {
  const response = await axios.post(
    MISTRAL_API_URL,
    {
      model: 'mistral-tiny',
      messages,
    },
    {
      headers: {
        'Authorization': `Bearer ${MISTRAL_API_KEY}`,
        'Content-Type': 'application/json',
      },
    }
  );
  return response.data.choices[0].message.content.trim();
}

async function main() {
  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout,
  });

  rl.question('Ask the agent a question: ', async (userInput) => {
    let messages = [
      { role: 'system', content: 'You are a helpful assistant that reflects on your answers and tries to improve them if possible.' },
      { role: 'user', content: userInput },
    ];

    // First response

```

```

    let answer = await askMistral(messages);
    console.log('\nInitial Answer:\n', answer);

    // Reflection step
    messages.push({ role: 'assistant', content: answer });
    messages.push({ role: 'system', content: 'Reflect on your previous answer. Was it
correct, clear, and complete? If not, revise and improve it.' });
    let reflection = await askMistral(messages);
    console.log('\nReflected/Improved Answer:\n', reflection);

    rl.close();
  });
}

main();

```

[reflection-and-self-improvement-pattern/langgraph_python_example/reflection_agent.py](#)

[Open full source](#)

```

import os
import requests

def ask_mistral(messages):
    api_key = os.getenv('MISTRAL_API_KEY')
    if not api_key:
        raise ValueError('Please set MISTRAL_API_KEY in your .env file')
    url = 'https://api.mistral.ai/v1/chat/completions'
    headers = {
        'Authorization': f'Bearer {api_key}',
        'Content-Type': 'application/json',
    }
    data = {
        'model': 'mistral-tiny',
        'messages': messages,
    }
    response = requests.post(url, headers=headers, json=data)
    response.raise_for_status()
    return response.json()['choices'][0]['message']['content'].strip()

def main():
    user_input = input('Ask the agent a question: ')
    messages = [
        {'role': 'system', 'content': 'You are a helpful assistant that reflects on your

```

```

answers and tries to improve them if possible.'},
    {'role': 'user', 'content': user_input},
]
# First response
answer = ask_mistral(messages)
print('\nInitial Answer:\n', answer)
# Reflection step
messages.append({'role': 'assistant', 'content': answer})
messages.append({'role': 'system', 'content': 'Reflect on your previous answer. Was it
correct, clear, and complete? If not, revise and improve it.'})
reflection = ask_mistral(messages)
print('\nReflected/Improved Answer:\n', reflection)

if __name__ == '__main__':
    main()

```

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `reflection-and-self-improvement-pattern/` folder from this repository.

Related Patterns

- [Planning and Execution](#)
- [ReAct](#)
- [Evaluator-Optimizer](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

Evaluator-Optimizer

Evaluator-Optimizer pairs a generator with an evaluator. The generator proposes; the evaluator scores; the optimizer revises or stops.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Evaluator-Optimizer pairs a generator with an evaluator and a bounded revision loop. The generator proposes an output. The evaluator checks it against explicit criteria. The optimizer either accepts it, asks for a targeted revision, escalates, or stops when the budget is exhausted.

The pattern is useful when quality can be judged more reliably than it can be produced in one pass. It is dangerous when the evaluator is vague, impressed by fluent prose, or allowed to approve work without checking evidence.

Reflection and Evaluator-Optimizer are related, but not identical. Reflection critiques and improves an output. Evaluator-Optimizer adds a decision boundary: pass, revise, refuse, escalate, or stop.

Use When

- The task has explicit criteria, tests, policies, examples, or evidence requirements.
- A second pass can catch failures the generator often misses.
- Iteration is worth the extra latency, cost, and complexity.
- The evaluator can produce structured feedback, not just prose critique.
- The controller can enforce max revisions, stop reasons, and escalation rules.

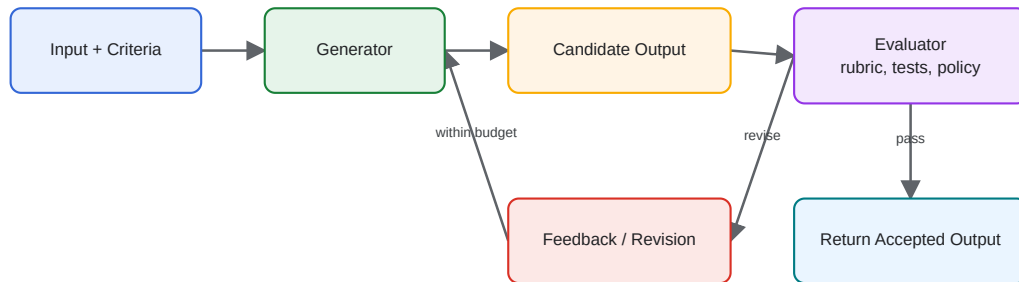
Avoid When

- The evaluator is only another vague opinion prompt.
- The evaluator cannot inspect the evidence needed to judge correctness.
- The task needs low latency and one pass is good enough.
- The revision loop can change facts, citations, policy decisions, or tool results without validation.
- The team cannot define what a false approval would look like.

Architecture

Use this diagram to read Evaluator-Optimizer as a system boundary, not only a code shape. The key ownership question is: the loop controller owns progress, budgets, stop conditions, and recovery state.

Evaluator-Optimizer Loop



System Shape

- **Pattern boundary:** the controller owns criteria, revision budget, evaluator selection, stop decisions, and escalation.
- **Generator role:** propose a candidate output or plan.
- **Evaluator role:** check the candidate against criteria, evidence, policy, and expected structure.
- **Optimizer role:** turn evaluator findings into targeted revision instructions.
- **Operational promise:** improve quality without letting a weak evaluator rubber-stamp unsafe or unsupported work.

Core Protocol

1. Receive the task, criteria, evidence requirements, and revision budget.
2. Generate the first candidate.
3. Evaluate the candidate with a rubric, tests, policy checks, and evidence checks.
4. If the candidate passes, return it with the evaluator decision.
5. If the candidate has fixable failures, produce targeted revision instructions.
6. Regenerate or patch the candidate.
7. Repeat until pass, refusal, escalation, timeout, or max revisions.
8. Record every candidate, evaluator decision, revision instruction, and stop reason.

Implementation Notes

Make the evaluator contract explicit.

```
type EvaluationDecision = {  
  status: 'pass' | 'revise' | 'block' | 'escalate';  
  score: number;
```

```

criteria: Array<{
  name: string;
  passed: boolean;
  evidenceRefs: string[];
  reason: string;
}>;
blockingFailures: string[];
revisionInstructions: string[];
stopReason?: 'passed' | 'max_revisions' | 'policy_block' | 'missing_evidence';
};

```

The controller should enforce the loop, not the evaluator prompt:

```

async function runEvaluatorOptimizer(task: Task, budget = { maxRevisions: 2 }) {
  let candidate = await generateCandidate(task);

  for (let revision = 0; revision <= budget.maxRevisions; revision += 1) {
    const decision = await evaluateCandidate(task, candidate);

    if (decision.status === 'pass') {
      return { status: 'succeeded', candidate, decision };
    }

    if (decision.status === 'block' || decision.status === 'escalate') {
      return { status: decision.status, candidate, decision };
    }

    if (revision === budget.maxRevisions) {
      return {
        status: 'failed',
        candidate,
        decision: { ...decision, stopReason: 'max_revisions' }
      };
    }

    candidate = await reviseCandidate(candidate, decision.revisionInstructions);
  }
}

```

The evaluator should not reward nicer prose if the candidate still lacks evidence. It should name the failing criterion and the evidence needed to pass.

Failure Modes

- Rubber-stamp evaluator: approves fluent but wrong output.
- Vague rubric: evaluator cannot distinguish a real failure from a style preference.
- Self-approval: the same model prompt generates and approves the answer without independent checks.
- Tone optimization: revisions make the answer smoother but not more correct.
- Missing evidence check: evaluator scores confidence without verifying citations or tool results.
- Reward hacking: generator learns phrases that satisfy the evaluator without satisfying the task.
- Endless revision loop: each pass creates new issues because stop conditions are weak.
- Hidden disagreement: evaluator concerns are not surfaced to the caller or trace.
- Evaluation drift: a prompt, model, or rubric change silently alters pass/fail behavior.

Evaluation Strategy

Evaluate the evaluator itself.

- Test false approvals: polished but wrong answers should fail.
- Test false rejections: correct but awkward answers should pass or receive minor revision.
- Test missing evidence: unsupported claims should block or escalate.
- Test policy failures: unsafe content should not be revised into acceptable wording.
- Test max-revision behavior: the loop must stop cleanly.
- Test adversarial candidates that flatter the evaluator or imitate rubric language.
- Test disagreement between deterministic checks and model-based evaluator judgment.
- Test regression cases from production failures.

A compact evaluator eval can look like this:

```
{
  "case_id": "polished_unsupported_refund_answer",
  "candidate": "Yes, the customer is clearly eligible for a full refund.",
  "available_evidence": ["order_status: delivered", "policy: refund requires damage evidence"],
  "expected_decision": {
    "status": "block",
    "blocking_failures": ["missing_damage_evidence"],
    "must_not_pass": true,
    "required_revision_instruction": "ask for or retrieve damage evidence"
  }
}
```

Use a scoring rubric that separates correctness from polish. For a refund recommendation, the evaluator should score evidence and policy before tone:

Criterion	Pass Condition	Blocking Failure
Evidence	Order, delivery status, customer claim, and policy version are referenced.	Recommendation cites missing or stale evidence.
Policy fit	The recommendation matches the active refund policy and threshold.	Candidate grants a refund that policy denies or escalates.
Authority	Candidate proposes only allowed actions for the current approval state.	Candidate implies payment execution without approval.
Customer message	Draft is accurate, specific, and does not overpromise.	Draft promises money movement before approval.
Traceability	Candidate links each claim to evidence refs.	Reviewer cannot replay why the recommendation passed.

A score can help ranking, but the release decision should use blockers first. A candidate with strong tone and missing policy evidence should fail, not receive a high average score.

Measure false approval rate, false rejection rate, revision success rate, max-revision rate, evaluator consistency, cost, latency, and recurrence of known failures.

Production Checklist

- Define criteria before generation starts.
- Separate generator and evaluator prompts.
- Use deterministic checks where possible before model judgment.
- Require evidence references for factual, policy, or tool-dependent claims.
- Set max revisions, timeout, and escalation rules.
- Record candidates, scores, criteria results, blocking failures, and revision instructions.
- Version evaluator prompts, rubrics, tests, and model routes.
- Track false approvals and false rejections in production review.
- Do not let evaluator approval bypass tool policy, approval gates, or security controls.
- Convert serious evaluator misses into regression evals.

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

This pattern currently has no dedicated code excerpt. Use the source and download links below for the full pattern folder.

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `evaluator-optimizer-pattern/` folder from this repository.

Related Patterns

- [Reflection](#)
- [Agent Loop](#)
- [Structured Output](#)
- [Production Evaluation Feedback Loops](#)
- [Observability and Evals](#)
- [Pattern Evaluation Checklist](#)

Self-Improvement

Self-improvement uses feedback from prior runs to improve future runs through reviewed changes to prompts, tools, retrieval, policies, tests, or skills.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Self-improvement uses feedback from prior runs to improve future runs through reviewed changes to prompts, tools, retrieval, policies, tests, or skills.

Use When

- You have run logs, eval failures, and review processes.
- Improvements are applied through versioned artifacts.
- Humans or automated gates can approve behavior changes.

Avoid When

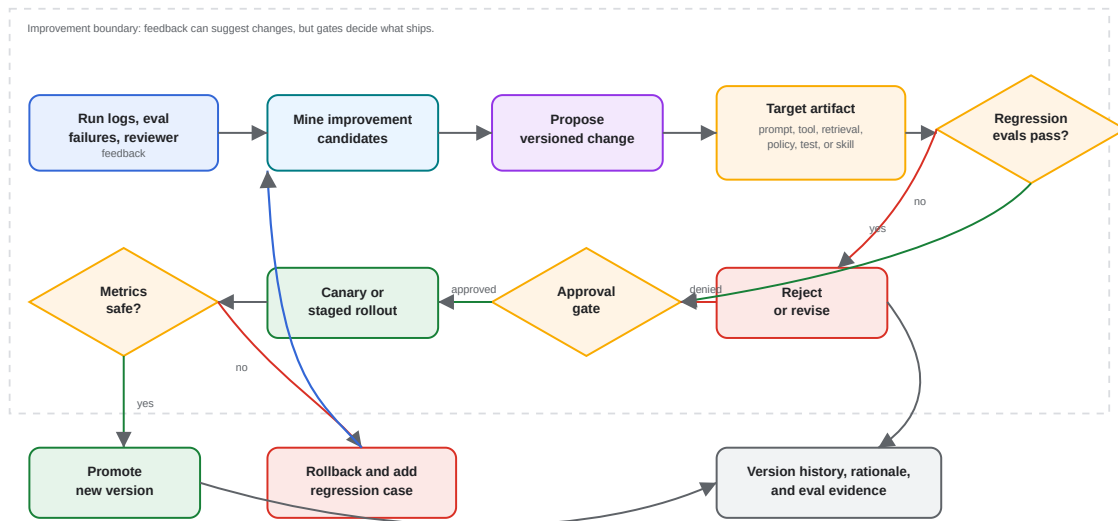
- The agent silently rewrites its own instructions in production.
- No eval suite exists to catch regressions.
- The feedback signal is noisy or easy to game.

Architecture

Use this diagram to read Self-Improvement as a system boundary, not only a code shape. The key ownership question is: the loop controller owns progress, budgets, stop conditions, and recovery state.

Self-Improvement Release Loop

The system improves through reviewed, tested, versioned changes, not silent production self-editing.



Read it as a release process: feedback can propose changes, but evals, approval, staged rollout, monitoring, and rollback decide what becomes a new version.

System Shape

- **Pattern boundary:** a controller repeatedly chooses the next step, executes it, observes the result, and decides whether to continue.
- **State owner:** the loop controller owns progress, budgets, stop conditions, and recovery state.
- **Primary artifact:** `reflection-and-self-improvement-pattern/` contains the runnable reference implementation and examples.
- **Operational promise:** Self-improvement uses feedback from prior runs to improve future runs through reviewed changes to prompts, tools, retrieval, policies, tests, or skills.

Core Protocol

1. Initialize goal state, constraints, budgets, and stop conditions.
2. Choose the next action from the current state instead of assuming the whole path upfront.
3. Execute the action through a validated tool, worker, or local function.
4. Observe the result and update state with evidence, errors, and remaining work.
5. Stop, retry, re-plan, or escalate according to explicit policy.

Implementation Notes

- Keep the pattern boundary explicit: inputs, state, side effects, and outputs should be visible.
- Validate model-produced decisions before they affect tools, users, or durable state.
- Emit enough trace data to debug failures after the run.

Failure Modes

- The pattern is applied where a simpler deterministic workflow would be better.
- State, tool calls, or model decisions are not observable enough to debug.
- The system lacks clear stop, retry, or escalation behavior.

Evaluation Strategy

- Test success cases, partial failure, repeated failure, budget exhaustion, and bad intermediate observations.
- Assert that the loop stops for the right reason and does not hide failed steps.
- Measure completion rate, number of iterations, recovery quality, cost, and latency.
- Include cases that prove each “Use When” condition is true for this pattern.
- Include negative cases from “Avoid When” so the system chooses a simpler or safer pattern when appropriate.

Production Checklist

- Set hard iteration, cost, and time limits.
- Persist state after meaningful steps if the run can be interrupted.
- Make retries idempotent or add compensation.
- Expose trace events for each decision, action, observation, and stop reason.
- Define human escalation for ambiguous, high-risk, or policy-blocked work.
- Keep the source bundle, generated chapter, tests, and deployment artifact in the same release.

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

`reflection-and-self-improvement-pattern/autogen_typescript_example/reflection_agent.ts`

[Open full source](#)

```
import dotenv from 'dotenv';
dotenv.config();
import axios from 'axios';
import readline from 'readline';

const MISTRAL_API_KEY = process.env.MISTRAL_API_KEY;
const MISTRAL_API_URL = 'https://api.mistral.ai/v1/chat/completions';
```

```

if (!MISTRAL_API_KEY) {
  console.error('Please set MISTRAL_API_KEY in your .env file');
  process.exit(1);
}

async function askMistral(messages: any[]) {
  const response = await axios.post(
    MISTRAL_API_URL,
    {
      model: 'mistral-tiny',
      messages,
    },
    {
      headers: {
        'Authorization': `Bearer ${MISTRAL_API_KEY}`,
        'Content-Type': 'application/json',
      },
    }
  );
  return response.data.choices[0].message.content.trim();
}

async function main() {
  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout,
  });

  rl.question('Ask the agent a question: ', async (userInput) => {
    let messages = [
      { role: 'system', content: 'You are a helpful assistant that reflects on your answers and tries to improve them if possible.' },
      { role: 'user', content: userInput },
    ];

    // First response
    let answer = await askMistral(messages);
    console.log('\nInitial Answer:\n', answer);

    // Reflection step
    messages.push({ role: 'assistant', content: answer });
    messages.push({ role: 'system', content: 'Reflect on your previous answer. Was it correct, clear, and complete? If not, revise and improve it.' });
  });
}

```

```

    let reflection = await askMistral(messages);
    console.log('\nReflected/Improved Answer:\n', reflection);

    rl.close();
  });
}

main();

```

[reflection-and-self-improvement-pattern/langgraph_python_example/reflection_agent.py](#)

[Open full source](#)

```

import os
import requests

def ask_mistral(messages):
    api_key = os.getenv('MISTRAL_API_KEY')
    if not api_key:
        raise ValueError('Please set MISTRAL_API_KEY in your .env file')
    url = 'https://api.mistral.ai/v1/chat/completions'
    headers = {
        'Authorization': f'Bearer {api_key}',
        'Content-Type': 'application/json',
    }
    data = {
        'model': 'mistral-tiny',
        'messages': messages,
    }
    response = requests.post(url, headers=headers, json=data)
    response.raise_for_status()
    return response.json()['choices'][0]['message']['content'].strip()

def main():
    user_input = input('Ask the agent a question: ')
    messages = [
        {'role': 'system', 'content': 'You are a helpful assistant that reflects on your answers and tries to improve them if possible.'},
        {'role': 'user', 'content': user_input},
    ]
    # First response
    answer = ask_mistral(messages)
    print('\nInitial Answer:\n', answer)
    # Reflection step

```



```
messages.append({'role': 'assistant', 'content': answer})
messages.append({'role': 'system', 'content': 'Reflect on your previous answer. Was it
correct, clear, and complete? If not, revise and improve it.'})
reflection = ask_mistral(messages)
print('\nReflected/Improved Answer:\n', reflection)

if __name__ == '__main__':
    main()
```

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `reflection-and-self-improvement-pattern/` folder from this repository.

Related Patterns

- [Planning and Execution](#)
- [ReAct](#)
- [Reflection](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

Self-Healing Workflows

Self-healing workflows detect failed steps and recover through retry, fallback, re-planning, or escalation.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Use this pattern to keep a long-running agentic workflow honest when a step fails. A self-healing workflow does not blindly try again. It records the failure, classifies it, chooses a recovery action, updates state, and stops when recovery would become unsafe or wasteful.

The goal is not perfect uptime. The goal is controlled recovery with evidence.

Scenario

An account-support agent gathers customer context, checks policy, drafts a resolution, and updates a CRM. The CRM write fails after the draft is created. A weak workflow would rerun the whole task and risk duplicate messages or duplicate records. A self-healing workflow classifies the failure as a partial side effect, keeps the successful draft, retries only the CRM write with an idempotency key, and escalates if the retry still fails.

The important design choice is recovery scope. Recover the failed step, not the entire workflow, unless the state proves the plan is stale.

Use When

- Failures are expected and can be classified before recovery starts.
- Steps have observable inputs, outputs, side effects, and stop reasons.
- Retries are idempotent, or compensation exists for partial side effects.
- Fallbacks reduce authority, cost, or risk instead of hiding the failure.
- Re-planning uses new evidence, changed state, or a confirmed stale plan.
- Human escalation has an owner, message, and handoff packet.

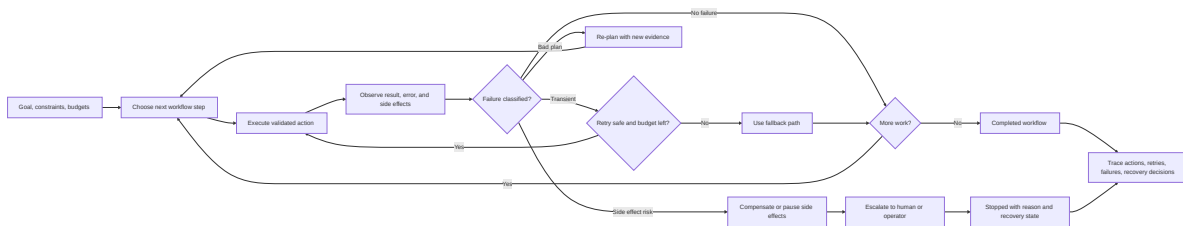
Avoid When

- The workflow cannot tell transient, fatal, policy, and partial-side-effect failures apart.
- Retrying a step can repeat an irreversible external action.
- The system would call the model again without new evidence.

- The recovery action is more dangerous than the original failure.
- The user or operator cannot inspect why the workflow recovered or stopped.

Architecture

Use this diagram to read Self-Healing Workflows as a system boundary, not only a code shape. The key ownership question is: the loop controller owns progress, budgets, stop conditions, and recovery state.

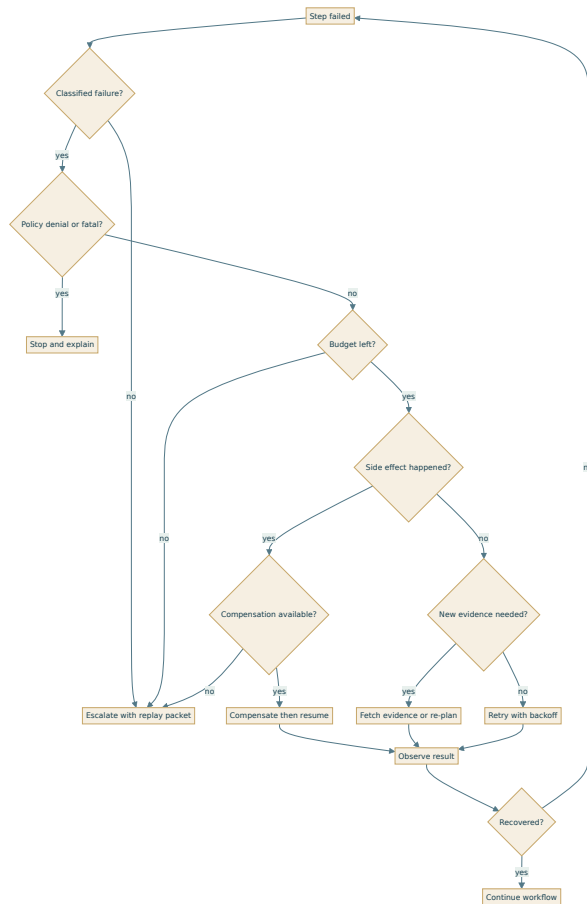


Read it as a recovery state machine: every retry, fallback, re-plan, compensation, escalation, and stop reason must be explicit and traceable.

Decision Rules

Classify before recovery. The controller should choose from a small recovery vocabulary and refuse ambiguous recovery.

Failure Class	Examples	Recovery Action	Stop Condition
Transient tool failure	timeout, temporary 5xx, connection reset	retry same step with backoff and idempotency key	retry budget exhausted
Rate limit or quota	429, token quota, per-account throttle	wait, use lower-cost fallback, or reschedule	deadline or budget exhausted
Missing evidence	source unavailable, weak retrieval, incomplete tool output	fetch more evidence or ask for clarification	no new source can change the decision
Stale plan	dependency changed, previous step invalidated	re-plan from current durable state	repeated stale-plan loop
Policy denial	unsafe tool request, forbidden data movement	block, explain, and escalate if needed	always stop autonomous recovery
Partial side effect	draft created but CRM write failed	compensate, resume failed step, or escalate	compensation unavailable
Fatal domain error	account closed, item unavailable, invalid recipient	stop and return a clear failure	always stop retry loop



This graph is the production contract: every edge needs a trace event, budget check, and stop reason.

System Shape

Component	Owns	Must Emit
Workflow controller	current step, progress, budgets, stop conditions	selected step, attempt number, remaining budget
Failure classifier	failure class, severity, retryability	class, evidence, confidence, ambiguity
Recovery policy	retry, fallback, re-plan, compensate, escalate, stop	decision, reason, allowed authority
Idempotency layer	duplicate suppression for side effects	key, target, previous result
Compensation handler	undo or repair of partial external actions	compensation action, result, residual risk
Replay builder	incident packet for debugging and evals	inputs, state, tool outputs, policy version
Escalation channel	human owner and user-facing status	owner, summary, next action, deadline

The controller owns the loop. Tools do not decide to retry themselves, and model calls do not silently re-plan after failure. Recovery is a policy decision over durable state.

Contract

The smallest useful contract separates step result, failure class, recovery decision, and trace evidence.

```
type FailureClass =  
  | "transient"  
  | "rate_limit"  
  | "missing_evidence"  
  | "stale_plan"  
  | "policy_denied"  
  | "partial_side_effect"  
  | "fatal";  
  
type RecoveryAction =  
  | "retry"  
  | "fallback"  
  | "replan"  
  | "compensate"  
  | "escalate"  
  | "stop";  
  
type RecoveryDecision = {  
  action: RecoveryAction;  
  reason: string;  
  retryAfterMs?: number;  
  requiresNewEvidence: boolean;  
  idempotencyKey?: string;  
};  
  
type RecoveryTraceEvent = {  
  workflowId: string;  
  stepId: string;  
  attempt: number;  
  failureClass: FailureClass;  
  decision: RecoveryDecision;  
  budgetRemaining: number;  
  stopReason?: string;  
};
```

This contract prevents a common production bug: treating every failure as a retryable exception.

Core Protocol

1. Persist workflow state before each step that can create an external side effect.

2. Execute the step through a bounded tool, worker, or model route.
3. If the step succeeds, store evidence and continue.
4. If the step fails, classify the failure with concrete evidence.
5. Check policy, retry budget, time budget, cost budget, and no-progress breaker.
6. Choose one recovery action: retry, fallback, re-plan, compensate, escalate, or stop.
7. Emit a trace event before executing the recovery action.
8. Resume from durable state, not from an optimistic in-memory plan.
9. Convert unresolved incidents into regression evals.

Workflow Transition Map

From	Event	To	Required Evidence
running	step succeeded	running or complete	output, validation result, updated state
running	transient failure	retry_wait	failure class, attempt count, backoff
retry_wait	timer elapsed	running	same idempotency key and unchanged target
running	fallback selected	running	fallback reason and reduced authority
running	stale plan detected	replanning	changed state or new evidence
replanning	valid plan produced	running	plan diff and validation result
running	partial side effect detected	compensating	side-effect ID and compensation rule
compensating	compensation succeeded	running or escalated	compensation result and residual risk
running	policy denial or fatal error	stopped	policy rule or fatal domain evidence
any active state	budget exhausted	escalated	budget values and replay packet

Implementation Notes

- Keep retry policy per step. A cheap read call and an outbound payment update should not share a retry rule.
- Use idempotency keys for every side-effecting call, including emails, CRM writes, ticket updates, calendar edits, and file mutations.
- Use backoff and jitter for transient infrastructure failures, not for policy failures.
- Re-plan only when state changed or new evidence arrived. Re-planning with the same facts usually burns tokens.

- Prefer lower-authority fallbacks: draft instead of send, read-only source instead of write source, cached summary instead of live mutation.
- Build replay packets automatically. An operator should not have to reconstruct state from scattered logs.
- Treat compensation as a first-class step with its own success, failure, and escalation path.

Failure Modes

- The loop retries a policy denial until the budget is exhausted.
- A partial side effect repeats because the retry lacks an idempotency key.
- Re-planning hides the original failure instead of preserving the trace.
- A fallback returns lower-quality data without marking the answer as degraded.
- The controller escalates without enough state for a human to continue.
- No-progress loops consume budget because every iteration looks slightly different.
- Compensation fails and the system keeps acting as if rollback succeeded.
- Recovery policy lives in prompt text only and cannot be audited.

Review Checklist

Use the [self-healing workflow review checklist](#) before moving a recovery loop past prototype stage.

- Every failure class has an owner and recovery action.
- Every retryable side effect has an idempotency key.
- Every compensation path has a residual-risk message.
- Every stop condition produces a user or operator-facing explanation.
- Every production incident can become a replayable regression eval.

Evaluation Strategy

Test recovery as behavior, not as exception handling.

Eval Case	Expected Result
transient read timeout	retries with backoff, same inputs, trace event recorded
repeated timeout	stops or escalates when retry budget is exhausted
policy-denied write	does not retry, records policy rule, returns blocked status
partial CRM write	compensates or resumes from idempotent state without duplicate write
stale plan	re-plans only after evidence changes
degraded fallback	marks output as fallback-derived and lower confidence
bad classifier	fails eval because recovery action does not match failure class
no-progress loop	breaker stops after threshold and emits replay packet

Measure completion rate, recovered-run rate, unsafe retry rate, duplicate side-effect rate, mean recovery latency, budget burn, escalation quality, and replay success rate.

Production Checklist

- Define failure classes in code, not only in prompt instructions.
- Set retry, cost, time, and no-progress budgets per workflow and per step.
- Persist state before externally visible side effects.
- Store idempotency keys with action targets and results.
- Require policy denial to stop autonomous recovery.
- Emit structured trace events for failure class, recovery decision, budget, and stop reason.
- Generate replay packets for escalations and failed recoveries.
- Add dashboards for retry rate, fallback rate, compensation rate, escalation rate, and duplicate-prevention hits.
- Turn resolved incidents into regression evals before widening automation.

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
self-healing-workflow-agent-pattern/autogen_typescript_example/self_healing_workflow.ts
```

[Open full source](#)

```
type FailureClass =  
  | "transient"  
  | "rate_limit"  
  | "missing_evidence"  
  | "stale_plan"  
  | "policy_denied"  
  | "partial_side_effect"  
  | "fatal";  
  
type RecoveryAction = "retry" | "fallback" | "replan" | "compensate" | "escalate" | "stop";  
  
type StepFailure = {  
  class: FailureClass;
```



```

    message: string;
    sideEffectId?: string;
};

type StepResult =
    | { ok: true; value: string }
    | { ok: false; failure: StepFailure };

function isStepFailure(result: StepResult): result is { ok: false; failure: StepFailure } {
    return result.ok === false;
}

type WorkflowState = {
    workflowId: string;
    stepId: string;
    attempt: number;
    maxAttempts: number;
    budgetRemaining: number;
    idempotencyKey: string;
    trace: string[];
};

type RecoveryDecision = {
    action: RecoveryAction;
    reason: string;
    retryAfterMs?: number;
};

function decideRecovery(state: WorkflowState, failure: StepFailure): RecoveryDecision {
    if (failure.class === "policy_denied") {
        return { action: "stop", reason: "Policy denials are never retried." };
    }

    if (failure.class === "fatal") {
        return { action: "stop", reason: "Fatal domain failure cannot be healed." };
    }

    if (failure.class === "partial_side_effect") {
        return failure.sideEffectId
            ? { action: "compensate", reason: `Compensate partial side effect
${failure.sideEffectId}.` }
            : { action: "escalate", reason: "Partial side effect has no compensation handle." };
    }
}

```

```

    if (state.attempt >= state.maxAttempts || state.budgetRemaining <= 0) {
      return { action: "escalate", reason: "Recovery budget exhausted." };
    }

    if (failure.class === "stale_plan" || failure.class === "missing_evidence") {
      return { action: "replan", reason: "Recovery needs changed state or new evidence." };
    }

    if (failure.class === "rate_limit") {
      return { action: "fallback", reason: "Use lower-cost fallback after quota failure." };
    }

    return {
      action: "retry",
      reason: "Transient failure can retry with same idempotency key.",
      retryAfterMs: Math.min(30_000, 2 ** state.attempt * 250)
    };
  }
}

async function runSelfHealingStep(
  state: WorkflowState,
  step: (idempotencyKey: string) => Promise<StepResult>
): Promise<StepResult> {
  while (state.budgetRemaining > 0) {
    const result = await step(state.idempotencyKey);
    if (result.ok) return result;
    if (!isStepFailure(result)) return result;

    const decision = decideRecovery(state, result.failure);
    state.trace.push(`${state.stepId} attempt ${state.attempt}: ${decision.action} - ${decision.reason}`);

    if (decision.action === "retry") {
      state.attempt += 1;
      state.budgetRemaining -= 1;
    }
  }
}

```

Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.

[self-healing-workflow-agent-pattern/langgraph_python_example/self_healing_workflow.py](#)

[Open full source](#)

```

from dataclasses import dataclass, field
from typing import Callable, Literal, Optional, Union

FailureClass = Literal[
    "transient",
    "rate_limit",
    "missing_evidence",
    "stale_plan",
    "policy_denied",
    "partial_side_effect",
    "fatal",
]

RecoveryAction = Literal["retry", "fallback", "replan", "compensate", "escalate", "stop"]

@dataclass
class StepFailure:
    failure_class: FailureClass
    message: str
    side_effect_id: Optional[str] = None

@dataclass
class StepSuccess:
    value: str

StepResult = Union[StepSuccess, StepFailure]

@dataclass
class WorkflowState:
    workflow_id: str
    step_id: str
    attempt: int
    max_attempts: int
    budget_remaining: int
    idempotency_key: str
    trace: list[str] = field(default_factory=list)

@dataclass
class RecoveryDecision:
    action: RecoveryAction
    reason: str
    retry_after_ms: Optional[int] = None

def decide_recovery(state: WorkflowState, failure: StepFailure) -> RecoveryDecision:
    if failure.failure_class == "policy_denied":

```

```

        return RecoveryDecision("stop", "Policy denials are never retried.")

    if failure.failure_class == "fatal":
        return RecoveryDecision("stop", "Fatal domain failure cannot be healed.")

    if failure.failure_class == "partial_side_effect":
        if failure.side_effect_id:
            return RecoveryDecision("compensate", f"Compensate partial side effect {failure.side_effect_id}.")
        return RecoveryDecision("escalate", "Partial side effect has no compensation handle.")

    if state.attempt >= state.max_attempts or state.budget_remaining <= 0:
        return RecoveryDecision("escalate", "Recovery budget exhausted.")

    if failure.failure_class in {"stale_plan", "missing_evidence"}:
        return RecoveryDecision("replan", "Recovery needs changed state or new evidence.")

    if failure.failure_class == "rate_limit":
        return RecoveryDecision("fallback", "Use lower-cost fallback after quota failure.")

    return RecoveryDecision(
        "retry",
        "Transient failure can retry with the same idempotency key.",
        retry_after_ms=min(30_000, 2**state.attempt * 250),
    )

def run_self_healing_step(
    state: WorkflowState,
    step: Callable[[str], StepResult],
) -> StepResult:
    while state.budget_remaining > 0:
        result = step(state.idempotency_key)
        if isinstance(result, StepSuccess):
            return result

        decision = decide_recovery(state, result)
        state.trace.append(f"{state.step_id} attempt {state.attempt}: {decision.action} - {decision.reason}")

        if decision.action == "retry":
            state.attempt += 1

```

Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `self-healing-workflow-agent-pattern/` folder from this repository.

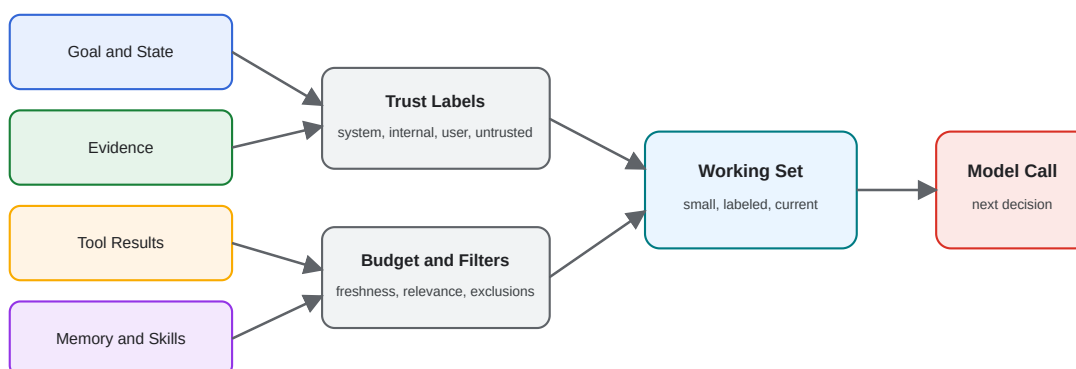
Related Patterns

- [Planning and Execution](#)
- [ReAct](#)
- [Reflection](#)
- [Durable Workflows](#)
- [Circuit Breakers, Fallbacks, and Replay](#)

Context Budgets And Working Sets

Context is not a storage system. It is the model's working set for the next decision, and that distinction is where a lot of agents quietly go wrong. Treat context like a box where anything potentially useful might as well be dropped in, and you get long prompts, duplicated state, stale memory, irrelevant tool output, hidden contradictions, and then missing evidence at the exact moment the model needs it. Context engineering is the discipline of deciding what the model should see right now, and nothing more.

Context Assembly Pipeline



What You Should Be Able To Do

After this chapter, you should be able to:

- define the working set for one model call;
- separate required, relevant, available, and excluded context;
- write a context manifest that reviewers can inspect;
- explain why each context item was included or omitted;
- test context selection directly instead of guessing from the final answer.

The Working Set

Borrow the idea from operating systems: the working set is the small set of information needed for the current step. For an agent, that usually means the active goal, the current step, the relevant constraints, compact state, the selected evidence, the tools available for this step, recent observations, any unresolved questions, and the budget and stop rules.

What it should not automatically include is everything else: the full chat history, every retrieved document, every tool result, all memories about the user, all available tools, every file in the workspace, and old plans that no longer match the current state. The model does not need everything. It needs the right things, and choosing them is the work.

Per-Agent Working Sets

In a multi-agent system, each agent should have its own working set. Giving every agent the same prompt, the same files, the same memory, the same tools, and the same conversation history defeats the point of decomposing the system in the first place.

Agent Type	Working Set Should Include	Should Usually Exclude
Router	task summary, route options, policy constraints.	full evidence bundle, long history, write tools.
Research agent	query, source rules, retrieval tools, citation requirements.	private memory unrelated to the task, write tools.
Tool agent	exact operation, validated inputs, tool contract, policy result.	broad conversation history, unrelated tools.
Reviewer	artifact to review, rubric, evidence, acceptance criteria.	implementation scratchpad, irrelevant prior drafts.
Supervisor	goal, worker contracts, progress, merge criteria.	every worker's raw context unless needed.
Human approver	proposed action, evidence, risk, policy result, diff.	hidden intermediate model chatter.

This is one of the strongest arguments for treating agents as services. Each agent receives the context its contract requires, not the context the whole system happens to be carrying around.

Context Budget

Every model call has a budget, even when the context window is large. The tokens you spend break down across instructions, state, evidence, tool descriptions, memory, conversation history, the output you reserve room for, and the cost of any compression you run. A large window does not remove the need for selection. It just makes bad selection more expensive and harder to notice.

Match the budget to the decision. A routing call should not carry the full evidence bundle. A tool-selection call should not carry unrelated user history. A final synthesis call should include evidence and constraints, not every failed intermediate thought the agent had on the way there.

Context Budget Ledger

Track the budget for each call before the prompt is assembled.

```
context_budget:
  call_type: refund_policy_synthesis
  model_window_tokens: 128000
```

```
reserved_output_tokens: 2000
max_input_tokens: 18000
allocation:
  instructions: 1200
  goal_and_state: 900
  policy: 3500
  evidence: 9000
  tool_results: 1800
  memory: 400
  conversation_history: 800
  safety_margin: 400
omitted:
  - id: old_chat_turns
    reason: "not relevant to current policy decision"
  - id: full_order_history
    reason: "replaced by validated order summary"
```

The ledger should make one thing obvious: context is allocated, not accumulated. If a source has no budget line or inclusion rule, it should not enter the model call by accident.

Context Manifests

For important agents, define a context manifest: an explicit statement of what is allowed to enter the model call.

Manifest Field	Example
Required state	goal, current_step, tenant_id, policy_version.
Allowed evidence	policy docs, order records, approved knowledge base snippets.
Allowed memory	explicit user preferences for this product area.
Allowed tool results	read-only lookup results for the current task.
Excluded data	secrets, unrelated private data, stale summaries, raw untrusted instructions.
Maximum size	token budget by source type.
Freshness rule	documents must be current or marked stale.
Trust labels	system, user, retrieved, tool, memory, untrusted.

A manifest turns context selection from prompt craft into system design. Once it exists, it can be reviewed, versioned, tested, and audited like any other part of the system.

In code, a working-set builder should make inclusion rules visible:


```

interface ContextItem {
  id: string;
  kind: 'state' | 'evidence' | 'tool_result' | 'memory' | 'instruction';
  trust: 'system' | 'internal' | 'user' | 'untrusted';
  sourceId?: string;
  relevance: number;
  freshness: 'current' | 'stale' | 'unknown';
  tokens: number;
  text: string;
}

type ContextSelection = {
  included: ContextItem[];
  omitted: { id: string; reason: string }[];
};

function buildWorkingSet(items: ContextItem[], maxTokens: number): ContextSelection {
  const required = items.filter(item => item.kind === 'state' || item.kind ===
'instruction');
  const optional = items
    .filter(item => item.kind !== 'state' && item.kind !== 'instruction')
    .filter(item => item.trust !== 'untrusted' || item.kind === 'evidence')
    .filter(item => item.freshness !== 'stale')
    .sort((a, b) => b.relevance - a.relevance);

  const selected: ContextItem[] = [];
  const omitted: ContextSelection['omitted'] = [];
  let used = 0;

  for (const item of [...required, ...optional]) {
    if (used + item.tokens > maxTokens) {
      omitted.push({ id: item.id, reason: 'token_budget' });
      continue;
    }

    selected.push(item);
    used += item.tokens;
  }

  for (const item of items) {
    if (selected.includes(item) || omitted.some(omittedItem => omittedItem.id === item.id))
continue;

    if (item.freshness === 'stale') omitted.push({ id: item.id, reason: 'stale' });
    else if (item.trust === 'untrusted' && item.kind !== 'evidence') {

```

```
    omitted.push({ id: item.id, reason: 'untrusted_non_evidence' });
  }
}

return { included: selected, omitted };
}
```

The point is not this exact scoring function. The point is that context selection becomes inspectable behavior, not hidden prompt assembly. The omitted list matters as much as the included list, because it tells operators whether the model missed evidence, excluded stale memory, or ran out of budget.

Sources Of Context

Context arrives from different sources, and each source carries a different level of trust.

Source	Use	Risk
Instructions	Defines behavior and constraints.	Too broad or contradictory.
Goal state	Keeps the task coherent.	Vague success criteria.
Working memory	Tracks progress and pending work.	Transcript dump instead of structured state.
Retrieved evidence	Grounds answers in sources.	Stale, irrelevant, or injected content.
Tool output	Reports external facts or action results.	Untrusted output treated as instruction.
Long-term memory	Carries prior knowledge or preferences.	Stale, private, or over-applied memory.
Skills	Loads procedural knowledge.	Irrelevant activation or outdated procedure.
Conversation history	Preserves user interaction.	Old turns override current task.

Do not mix these into one blob. Label them, and keep the boundaries sharp: instructions are not evidence, user claims are not verified facts, and tool results are not policy.

Context Tiers

Tiers are a simple way to decide what actually enters the model.

Tier	Content	Default Treatment
Required	Goal, policy, current state, stop rules.	Always include in compact form.

Tier	Content	Default Treatment
Relevant	Evidence, tool results, files, memories for the current step.	Include selectively with source labels.
Available	Workspace files, old messages, full documents, extra tools.	Reference by ID and retrieve only when needed.
Excluded	Secrets, unrelated private data, stale memory, untrusted instructions.	Keep out of context.

The split prevents context flooding, and it makes reviews easier, because the team can always ask why a given item was included.

Curated Context

Curated context means the system deliberately selects the smallest useful set of information before the model call. For a coding agent, that usually runs as a sequence:

1. identify the primary files;
2. search for symbols, imports, tests, and call sites;
3. rank secondary files;
4. load small relevant files directly;
5. load snippets or summaries for large files;
6. exclude unrelated files even when they sit right next to the relevant ones.

The same pattern applies well outside code. A support agent does not need every customer record. It needs the current request, the relevant account facts, the applicable policy, the recent related events, and the tools allowed for this step. Curation is not only a cost optimization. It is a reliability control, because the smaller the working set, the fewer ways the model has to be wrong.

Progressive Disclosure

Load context in stages rather than all at once. There is no reason to load every document, memory, file, tool description, and prior run artifact at the start of a task. Begin with the minimum needed to choose the next path, then retrieve more only when the state justifies it:

1. route the task from a compact user request;
2. load the policy and state needed for that route;
3. expose read-only tools for investigation;
4. load detailed evidence only after the missing facts are known;
5. expose write tools only after eligibility and approval checks pass;
6. include final evidence for synthesis and the user-facing explanation.

Disclosing context progressively lowers token cost, reduces prompt-injection exposure, and leaves traces that are far easier to inspect afterward.

Compression

Compression helps when context is large, but it is not free. Prefer structured state over chat summaries, source IDs over pasted documents, short cited snippets over full text, task-specific memory over global memory, and file references over full file contents. When you do summarize, keep the decisions, constraints, and unresolved questions intact.

Some things should never be compressed away: approval state, tool errors, policy constraints, source provenance, contradictions, user corrections, open questions, and stop reasons. Summaries are lossy by nature, so treat them as derived artifacts rather than as the truth of the run.

Compaction

Compaction is not the same as ordinary summarization. Summarization produces a shorter text; compaction preserves the operational state needed to continue a run. A useful compaction keeps the active goal, the current state, the completed and pending steps, the decisions made, the evidence used and its source IDs, the tool calls and their results, the policy and approval state, the open questions, the known errors, and the stop reasons.

It should not be your primary context strategy. If every run depends on emergency compaction, the system is loading too much context too early. Use compaction as a recovery and continuation mechanism, not as permission to keep flooding the model.

Compaction Boundaries

Context compaction should happen at stable boundaries, not in the middle of a fragile reasoning step. Good boundaries include:

- after requirements are restated;
- after discovery produces a source list;
- after a plan is accepted;
- after a commit or checkpoint;
- before handing work to another agent.

Each compaction should preserve decisions, open risks, file paths, verification evidence, and user constraints. It should remove transcript noise, not erase accountability.

Context Minimization

Remove context once it has served its purpose. This matters most for untrusted content. After a user message, web page, email, ticket, or document has been turned into a validated intermediate artifact, the later steps often do not need the raw source text at all:

1. a user message enters as untrusted text;
2. the system extracts intent into a typed request;
3. policy validates the request;

4. later tool calls use the typed request, not the raw text.

That reduces token load and, more importantly, lowers the chance that an old untrusted instruction keeps steering later decisions. Do not minimize blindly, though. Keep enough context to preserve user intent, auditability, and correction paths. The goal is not to forget. It is to stop treating every old token as an active instruction.

Tool Results

Tool results are context, but they are not instructions. A search result, web page, retrieved document, database row, browser observation, or command output can carry hostile or misleading text, and the model should see all of it as data from a source rather than as a new system message.

Good tool-result context arrives wrapped in its metadata: source, timestamp, trust level, scope, the raw result or excerpt, the parsed fields, any error state, and a correlation ID. Bad tool-result context just pastes the text into the prompt and hopes the model knows what to trust.

Memory Selection

Memory should not load by default. Before pulling a memory into context, ask whether it is relevant to the current goal, whether it is fresh enough, whether it was user-provided or inferred or imported, whether it is allowed for this user and task, whether it could bias the answer incorrectly, and whether it can be corrected or deleted. Memory is powerful because it persists, and that is exactly what makes it dangerous: a wrong memory can keep failing the system long after the run that created it is gone.

Working Set Drift

Working set drift is what happens when the context no longer matches the real task. It creeps in through familiar paths: an old plan that survives after the user changes the goal, a stale retrieval result that never leaves context, a compressed summary that drops a constraint, a tool error masked by a later success, a subagent summary that omits its own uncertainty, memory reused from a different domain, or a chat history that slowly drowns the current instructions.

Drift is hard to catch from the final answer alone, which is why you trace the context bundle for each model call rather than only the output.

Source Trust

Every context item should carry a trust label. Useful ones include system instruction, developer instruction, user request, verified internal data, retrieved internal document, external web content, tool result, long-term memory, generated summary, human approval, and untrusted content. Labels help the model and the runtime interpret context correctly, and they help operators reconstruct why a bad decision happened.

The rule that matters most: lower-trust context must never override higher-trust governance. A retrieved document can provide evidence, but it cannot change tool permissions. A tool result can report state, but it cannot rewrite approval policy. A memory can inform personalization, but it cannot override the current user request.

Evaluation

Evaluate context selection directly, not just final answers. Useful cases include checking that required evidence is present, that irrelevant evidence is excluded, that stale evidence is rejected, that prompt injection in retrieved text is ignored, that user corrections and tool errors are preserved, that memory is omitted when irrelevant, that summaries preserve constraints, and that the final answer cites the right sources. At the system level, also confirm that each agent's working set contains only what its contract requires, that minimization removes untrusted raw text after extraction, that compaction preserves approval state and open questions, and that the manifest actually rejects excluded data.

Worthwhile metrics include context precision and recall, source freshness, citation faithfulness, token cost, refusal rate when evidence is missing, error-preservation rate, drift rate, over-inclusion rate, missing-critical-context rate, and compaction loss rate. When an answer comes out wrong, inspect the context that produced it before blaming the model.

Design Checklist

Before shipping a context-heavy agent, work through these questions: What is always included? What is retrieved only when needed? What is never allowed into context? How are sources labeled, and how are tool results kept separate from instructions? How is stale memory detected? How are summaries generated and validated? What context bundle was used for each model call, and can an operator replay the run with the same context? Which evals test context selection rather than only final answers? Does each agent have its own working set? Which raw untrusted inputs are removed after extraction, what does compaction preserve, and which trust labels show up in your traces?

Design Rule

Do not maximize context. Curate the working set.

Related Chapters

- [Context Engineering](#)
- [What Is An Agent?](#)
- [Agent Harnesses](#)
- [Agents As Services](#)
- [Agent Threat Model](#)
- [Working Memory](#)
- [Resource-Aware Agent Design](#)
- [Semantic Recall and RAG](#)

Context Engineering

Context engineering controls what the model sees: instructions, state, retrieval results, tool documentation, memory, examples, and prior messages.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Context engineering decides what the model sees before it answers or acts. It assembles instructions, user input, working memory, retrieved evidence, tool results, policies, examples, budgets, and exclusions into a controlled working set.

RAG is not “put search results in the prompt.” It is an evidence pipeline. Context engineering is the larger discipline around that pipeline. The system must decide which sources are eligible, what counts as fresh enough, what the caller may see, how retrieved content is cited, what gets omitted, what gets pinned, and what the agent should do when evidence is missing or conflicting.

The rule is simple: context is assembled, not dumped. The model should see a deliberate packet, not a pile of chat history, retrieved text, tool output, and memory fragments.

This pattern owns the assembly boundary, not every source system inside it. Working memory owns run state. Durable memory owns what survives across runs. Retrieval owns evidence discovery. Context engineering decides which of those records enters the next model call.

Use When

- The answer depends on a large, changing, or private knowledge base.
- Relevant sources can be chunked, embedded, filtered, cited, and inspected.
- The retrieval layer can enforce tenant, role, source, and freshness constraints.
- The agent can refuse or escalate when evidence is missing.
- The system can evaluate retrieval quality separately from final-answer quality.
- The agent needs working memory, tool results, policy, and retrieved evidence in one coherent packet.

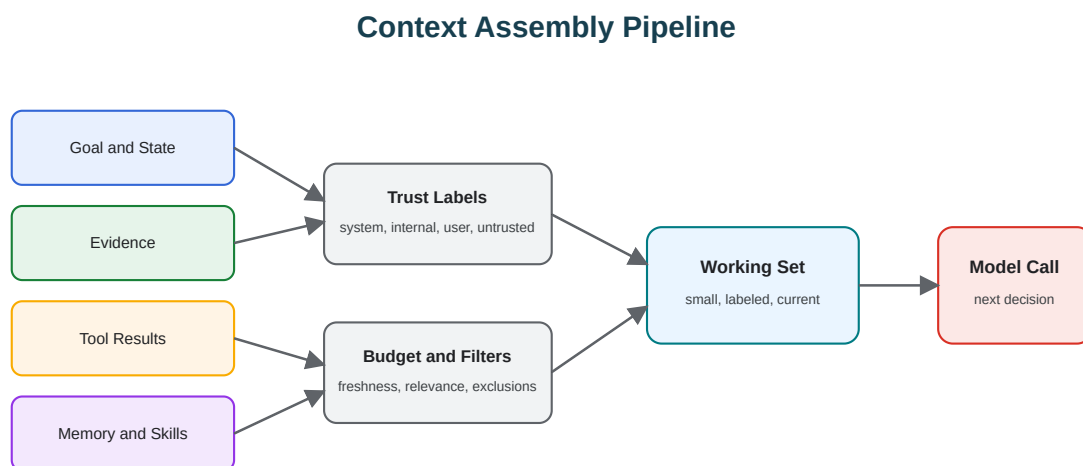
Avoid When

- The required knowledge is already in the task input or deterministic system state.
- The corpus is too noisy, stale, or untrusted to retrieve safely.

- The system cannot distinguish trusted metadata from untrusted document content.
- The answer needs exact database state that should come from a typed tool, not semantic search.
- Citations, source IDs, or retrieval traces cannot be stored for review.
- The system would rely on raw transcript stuffing instead of a controlled context builder.

Architecture

Use this diagram to read Context Engineering as a system boundary, not only a code shape. The key ownership question is: the caller or a small application service owns task state until a runtime pattern is introduced.



System Shape

- **Pattern boundary:** the context builder decides what may enter the model working set and how each part is labeled.
- **State owner:** the runtime owns working state, policies, budgets, retrieval traces, tool results, memory records, and context assembly records.
- **Model role:** the model uses the assembled packet to answer, act, ask for missing evidence, or explain why it cannot proceed.
- **Policy boundary:** source eligibility, tenant access, redaction, freshness, tool-result handling, memory writes, and instruction hierarchy run before generation.
- **Operational promise:** the model works from a small, labeled, policy-checked packet rather than vague memory or unfiltered context.

Core Protocol

1. Classify the context need: policy, task instructions, working state, tool result, user memory, event history, private record, public source, or example.
2. Load current goal, working memory, budget state, approval state, and stop conditions.
3. Apply caller, tenant, role, source, freshness, and data-handling filters before retrieval or inclusion.

4. Retrieve candidate chunks with metadata, not naked text.
5. Rank and trim candidates by relevance, freshness, diversity, trust, and token budget.
6. Build an evidence set with source IDs, citations, confidence, and known gaps.
7. Assemble context with instructions, policy, state, evidence, tool results, memory, examples, and exclusions separated.
8. Generate an answer, action, refusal, or escalation using only eligible context.
9. Record query, filters, source IDs, packet sections, omitted material, citations, and stop reason.

Implementation Notes

Treat retrieved material as evidence, not authority. A web page, email, ticket, PDF, or document can contain useful facts and malicious instructions at the same time.

Context Packet

A context packet is the runtime artifact that explains what the model was allowed to see.

```
type ContextSectionKind =  
  | "system_instruction"  
  | "policy"  
  | "user_request"  
  | "working_memory"  
  | "retrieved_evidence"  
  | "tool_result"  
  | "episodic_memory"  
  | "semantic_memory"  
  | "example"  
  | "exclusion";  
  
type ContextSection = {  
  kind: ContextSectionKind;  
  title: string;  
  content: string;  
  sourceRefs: string[];  
  trustLevel: "trusted" | "internal" | "user_supplied" | "public" | "untrusted";  
  freshness?: {  
    observedAt?: string;  
    expiresAt?: string;  
  };  
  tokenEstimate: number;  
};  
  
type ContextPacket = {  
  packetId: string;
```

```

runId: string;
goalId?: string;
actorId: string;
tenantId: string;
task: string;
sections: ContextSection[];
budget: {
  maxTokens: number;
  usedTokens: number;
  reservedOutputTokens: number;
};
omittedRefs: Array<{
  ref: string;
  reason: "not_relevant" | "stale" | "not_allowed" | "duplicate" | "over_budget";
}>;
policyVersion: string;
};

```

This packet should be traceable. If a run fails, you should be able to inspect what was included, what was omitted, and why.

Ordering And Trust Rules

Ordering is part of the control plane. A practical order is:

1. system instructions;
2. policy and safety constraints;
3. current user or workflow request;
4. goal and working memory;
5. approved tool results;
6. retrieved evidence;
7. retrieved memories;
8. examples;
9. explicit exclusions and known gaps.

The point is not only token order. The point is separation. Retrieved text, tool output, memory, and user documents should never be allowed to rewrite the system instructions, policy, tool permissions, approval rules, or memory write rules.

Evidence Contract

```

type EvidenceChunk = {
  sourceId: string;
  sourceType: 'policy' | 'docs' | 'ticket' | 'email' | 'memory' | 'web';
  tenantId?: string;
}

```

```

trustLevel: 'trusted' | 'internal' | 'user_supplied' | 'public' | 'unknown';
freshness: {
  retrievedAt: string;
  sourceUpdatedAt?: string;
  maxAgeDays?: number;
};
permissions: {
  allowedRoles: string[];
  redaction: 'none' | 'pii' | 'secret' | 'tenant_scoped';
};
score: number;
excerpt: string;
citation: string;
};

```

Answer Contract

The evidence contract should travel with the answer:

```

type RagAnswer = {
  status: 'answered' | 'missing_evidence' | 'conflicting_evidence' | 'refused';
  answer?: string;
  citations: string[];
  evidenceRefs: string[];
  missingEvidence?: string[];
};

```

Eligibility Check

A small eligibility check catches many production failures:

```

function isEligibleEvidence(chunk: EvidenceChunk, callerRole: string, now: Date) {
  if (!chunk.permissions.allowedRoles.includes(callerRole)) return false;
  if (chunk.permissions.redaction === 'secret') return false;

  if (chunk.freshness.sourceUpdatedAt && chunk.freshness.maxAgeDays) {
    const updatedAt = new Date(chunk.freshness.sourceUpdatedAt).getTime();
    const ageDays = (now.getTime() - updatedAt) / 86_400_000;
    if (ageDays > chunk.freshness.maxAgeDays) return false;
  }

  return chunk.trustLevel !== 'unknown';
}

```

Do not let the model decide whether a source is allowed. The model can summarize evidence quality. Software should enforce eligibility.

Context Budgeting

Context budgeting is not just truncation. It is deciding what the model must see, what can be summarized, what can be retrieved again, what can be omitted, and what must be pinned.

Pin small, high-authority items: system instructions, policy constraints, active goal, stop condition, approval state, and the user request. Summarize bulky but low-risk history. Retrieve source evidence instead of carrying old copied excerpts. Drop duplicate chunks. Omit stale, unauthorized, or low-confidence material. Reserve output tokens before filling the input context.

When the budget is tight, the runtime should degrade explicitly:

- answer from fewer cited sources;
- ask a clarifying question;
- retrieve again with narrower filters;
- summarize intermediate state;
- refuse when required evidence cannot fit safely;
- escalate when policy requires complete evidence.

Failure Modes

- Stale but plausible evidence is retrieved and treated as current.
- Retrieved text contains instructions that override the system goal.
- Tool output is treated as a new instruction rather than as data.
- Memory overrides the current user request.
- Working state is stale but still included as current.
- Chunks from the wrong tenant, role, or customer enter the context.
- The retriever returns semantically similar but operationally wrong sources.
- Citations point to broad documents instead of the exact supporting chunk.
- The agent answers when the evidence is missing or conflicting.
- Memory writes store an unverified summary as if it were durable fact.
- Retrieval scores are logged, but filters, source IDs, and freshness are not.
- The final answer is evaluated, but retrieval quality is not.
- The context packet cannot be reconstructed after an incident.
- Important policy or approval state is omitted under token pressure.

Evaluation Strategy

Context evals should test retrieval, assembly, and answer behavior separately.

- Test known-answer questions where the correct source is present.
- Test missing-evidence cases where the agent should refuse or ask for help.
- Test stale-source cases where an older source conflicts with a newer one.

- Test conflicting-source cases where the answer must explain uncertainty.
- Test prompt injection inside retrieved documents.
- Test prompt injection inside tool results and memory records.
- Test tenant and permission boundaries.
- Test context budget pressure and verify policy, goal, and citations are not dropped.
- Test omitted-source behavior and verify the trace explains why material was excluded.
- Test citation coverage: every factual claim should map to source evidence.
- Test retrieval precision and recall before testing final prose quality.

A compact eval fixture can make the evidence requirement explicit:

```
{
  "case_id": "stale_refund_policy",
  "question": "Can a damaged item be refunded after 45 days?",
  "retrieved_sources": [
    { "source_id": "refund_policy_2024", "freshness": "stale" },
    { "source_id": "refund_policy_2026", "freshness": "current" }
  ],
  "expected": {
    "must_cite": ["refund_policy_2026"],
    "must_not_cite": ["refund_policy_2024"],
    "status": "answered",
    "checks": ["freshness", "citation_coverage", "no_untrusted_instructions"]
  }
}
```

Measure retrieval recall, retrieval precision, source freshness, packet completeness, context-token efficiency, citation faithfulness, missing-evidence refusal rate, prompt-injection resistance, tenant-boundary violations, and answer quality grounded in cited evidence.

Production Checklist

- Define eligible sources by tenant, role, source type, freshness, and data class.
- Keep source metadata with every retrieved chunk.
- Build a traceable context packet for each run.
- Separate instructions from retrieved facts in the assembled context.
- Keep tool results, memory, retrieved evidence, examples, and user content labeled separately.
- Redact or exclude sources before generation, not after.
- Pin policy, active goal, stop condition, and approval state before filling optional context.
- Require citations for factual claims.
- Refuse or escalate when evidence is missing, stale, or conflicting.
- Trace query, filters, source IDs, scores, packet sections, omitted refs, and final citations.
- Evaluate retrieval quality separately from answer quality.
- Review memory writes before storing retrieved summaries as durable facts.

- Version chunking, embedding model, retrieval filters, rerankers, prompts, and citation rules.
- Add regression evals for context poisoning, stale context, omitted policy, and token pressure.

The architectural rule is simple: every item in context needs a reason, a source, a trust level, and a budget. Continue with [Context Budgets and Working Sets](#) for packet limits and [Agentic RAG Systems](#) for system-level composition.

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

`context-engineering-pattern/langgraph_python_example/rag_example.py`

[Open full source](#)

```
"""
Context Engineering Example: Retrieval-Augmented Generation (RAG) with Mistral
Requirements: pip install langchain-community sentence-transformers faiss-cpu requests
Note: This is a minimal example using HuggingFace embeddings by default.
"""
import os
import requests
from typing import List

# Optionally load environment variables from a local .env if present
try:
    from dotenv import load_dotenv, find_dotenv # type: ignore
    load_dotenv(find_dotenv(usecwd=True), override=True)
except Exception:
    pass

# Dummy documents
docs = [
    {"content": "Agentic systems are autonomous AI systems."},
    {"content": "Prompt engineering improves LLM outputs."}
]

# Build vector store (in-memory for demo)
```

```

texts = [d["content"] for d in docs]
try:
    from langchain_community.embeddings import HuggingFaceEmbeddings
    from langchain_community.vectorstores import FAISS

    embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
    vectorstore = FAISS.from_texts(texts, embeddings)
    retriever = vectorstore.as_retriever()

    def retrieve(query: str, k: int = 3):
        return retriever.get_relevant_documents(query)
except Exception:
    class _MiniDoc:
        def __init__(self, text: str):
            self.page_content = text

    def retrieve(query: str, k: int = 3):
        query_terms = {term.strip(".,?!").lower() for term in query.split()}

        def score(text: str) -> int:
            text_terms = {term.strip(".,?!").lower() for term in text.split()}
            return len(query_terms & text_terms)

        ranked = sorted(texts, key=score, reverse=True)
        return [_MiniDoc(text) for text in ranked[:k]]

MISTRAL_API_KEY = os.getenv("MISTRAL_API_KEY")

def chat_mistral(messages):
    if not MISTRAL_API_KEY:
        user_message = messages[-1]["content"]
        context = user_message.split("Context:\n", 1)[-1].split("\n\nQuestion:", 1)[0]
        return f"Local fallback answer from retrieved context: {context}"

    resp = requests.post(
        "https://api.mistral.ai/v1/chat/completions",
        headers={
            "Authorization": f"Bearer {MISTRAL_API_KEY}",
            "Content-Type": "application/json",
        },
        json={
            "model": "mistral-large-latest",
            "messages": messages,
            "temperature": 0.2,
        },
    )

```

```

        timeout=60,
    )
    resp.raise_for_status()
    data = resp.json()
    return (data.get("choices") or [{}])[0].get("message", {}).get("content", "")

if __name__ == "__main__":
    query = "What are agentic systems?"
    retrieved = retrieve(query)
    context = "\n\n".join(d.page_content for d in retrieved)
    answer = chat_mistral([
        {"role": "system", "content": "Use the provided context to answer the question succinctly."},
        {"role": "user", "content": f"Context:\n{context}\n\nQuestion: {query}"},
    ])
    print("Answer:", answer)

```

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `context-engineering-pattern/` folder from this repository.

Related Patterns

- [Context Budgets and Working Sets](#)
- [Working Memory](#)
- [Memory-Augmented Agent](#)
- [Knowledge-Bound Agents](#)
- [Agentic RAG Systems](#)
- [Pattern Evaluation Checklist](#)

Memory-Augmented Agent

Memory gives an agent continuity, but it also creates a durable trust boundary.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Memory-augmented agents store and retrieve information across turns, sessions, tasks, or users. Memory gives an agent continuity, but it also creates a durable trust boundary. A bad answer can be corrected in the next turn. A bad memory can keep influencing future runs.

The useful way to think about memory is simple: memory is not truth. It is a record with a source, scope, timestamp, confidence, privacy class, retention rule, and correction path. The model can propose a memory write. The runtime decides whether that write is allowed, how it is stored, how long it lives, and when it can be recalled.

Use this pattern for durable memory policy. For the state of a single run, use [Working Memory](#). For deciding whether a memory enters the prompt, use [Context Engineering](#) and [Context Budgets and Working Sets](#).

Use When

- The agent needs continuity beyond one interaction.
- Stored facts can be scoped, updated, corrected, and deleted.
- Retrieval results can be cited, inspected, and excluded when they are stale or unsafe.
- The product has a reason to remember preferences, task state, decisions, events, or user-provided facts.
- You can define consent, retention, privacy, and tenant boundaries.

Avoid When

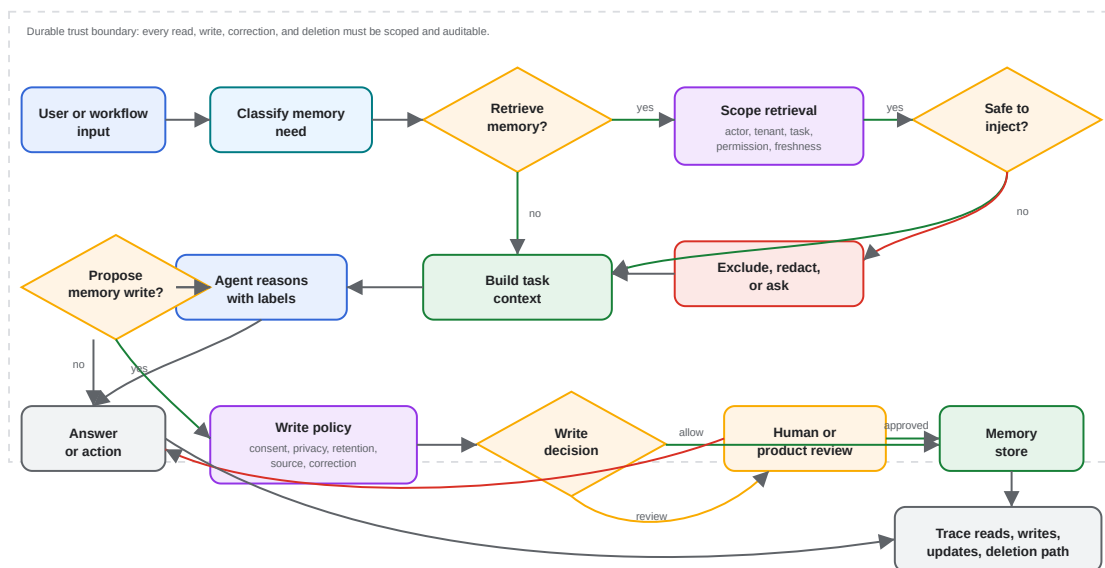
- The system would store sensitive data without consent or retention rules.
- Retrieved memories cannot be distinguished from current instructions.
- The memory store is used as an uncurated transcript dump.
- The agent would treat old summaries as authoritative facts.
- You cannot explain, edit, or delete what the agent remembers.

Architecture

Use this diagram to read Memory-Augmented Agent as a system boundary, not only a code shape. The key ownership question is: the memory or retrieval layer owns long-lived knowledge, while the agent owns task-local working state.

Memory-Augmented Agent Lifecycle

Memory is retrieved and stored as governed data, not as hidden model authority.



Read it as a governed lifecycle: retrieve only scoped memory, inject it as labeled data, and store only approved memory with retention and correction rules.

System Shape

- **Pattern boundary:** the memory boundary owns retrieval scope, write policy, record schema, retention, correction, deletion, and audit.
- **State owner:** the memory service or application store owns durable memory; the agent owns only proposed reads and writes.
- **Model role:** the model can summarize, classify, and propose memory updates, but it does not silently decide what the system will remember.
- **Policy boundary:** memory writes pass through consent, source, privacy, tenant, retention, and safety checks before storage.
- **Operational promise:** memory improves continuity without becoming an unreviewed pile of durable claims.

Core Protocol

1. Classify the information need: working state, episodic event, semantic fact, user preference, policy source, or tool-result cache.
2. Retrieve only memory records scoped to the current actor, tenant, task, permissions, and freshness window.
3. Inject memory as data with source labels, timestamps, confidence, and trust level.
4. Keep retrieved memory separate from system instructions and policy.

- 5. Ask the model to propose memory writes only when the task produced a durable fact, preference, event, correction, or decision.
- 6. Run memory write policy before storage.
- 7. Store approved memory with provenance, retention, privacy class, and deletion path.
- 8. Record memory reads, writes, updates, denials, and deletions in the trace.

Implementation Notes

Do not store raw chat history as memory by default. Chat history is evidence. Memory is a curated operational record. The difference matters because memory is reused by future runs.

Memory Types

Memory Type	What It Stores	Typical Risk
Working memory	Current task state, open questions, active constraints.	stale or inconsistent state inside a run.
Episodic memory	Events that happened, with time and participants.	over-retention, privacy leakage, wrong attribution.
Semantic memory	Durable facts about a domain, user, project, or system.	treating unverified claims as truth.
Preference memory	User choices and habits.	over-personalization or storing sensitive preferences.
Policy memory	Approved rules, sources, and constraints.	stale policy or unauthorized edits.
Tool-result cache	Prior tool outputs reused for speed or cost.	stale data and cross-tenant leakage.

Memory Write Contract

A memory write should be a typed request. The record should say what is being stored, why it is allowed, where it came from, who it belongs to, and how it can be corrected.

```
type MemoryKind =
  | "working_state"
  | "episodic_event"
  | "semantic_fact"
  | "user_preference"
  | "policy_reference"
  | "tool_result_cache";

type MemoryWriteRequest = {
```

```

runId: string;
actorId: string;
tenantId: string;
proposedBy: "model" | "workflow" | "user" | "operator";
kind: MemoryKind;
content: string;
sourceRefs: string[];
sourceTrust: "user_provided" | "tool_result" | "approved_source" | "untrusted_content";
confidence: "low" | "medium" | "high";
privacyClass: "public" | "internal" | "private" | "sensitive";
retention: {
  expiresAt?: string;
  deleteOnRequest: boolean;
};
consent: {
  required: boolean;
  granted: boolean;
  consentRef?: string;
};
correctionPath: string;
};

type MemoryPolicyDecision =
  | { decision: "allow"; reason: string }
  | { decision: "deny"; reason: string }
  | { decision: "review"; reason: string; approverRole: string };

function decideMemoryWrite(request: MemoryWriteRequest): MemoryPolicyDecision {
  if (request.sourceTrust === "untrusted_content") {
    return { decision: "review", reason: "untrusted_source", approverRole:
"memory_reviewer" };
  }

  if (request.privacyClass === "sensitive" && !request.consent.granted) {
    return { decision: "deny", reason: "missing_consent_for_sensitive_memory" };
  }

  if (request.kind === "policy_reference" && request.proposedBy === "model") {
    return { decision: "review", reason: "policy_memory_requires_review", approverRole:
"policy_owner" };
  }

  return { decision: "allow", reason: "memory_policy_passed" };
}

```

Memory Records

The stored record should not be just text.

```
type MemoryRecord = {  
  memoryId: string;  
  kind: MemoryKind;  
  content: string;  
  actorId: string;  
  tenantId: string;  
  sourceRefs: string[];  
  sourceTrust: string;  
  confidence: string;  
  privacyClass: string;  
  createdAt: string;  
  updatedAt: string;  
  expiresAt?: string;  
  correctionPath: string;  
  policyVersion: string;  
};
```

This schema is intentionally boring. Boring memory is easier to inspect, correct, delete, evaluate, and audit.

Retrieval Rules

Memory retrieval should be scoped before relevance ranking. First filter by tenant, actor, permissions, memory kind, retention, and freshness. Then rank by relevance. If the system ranks first and filters later, private or stale memory can leak into context.

Retrieved memory should enter context with labels such as `source`, `created_at`, `confidence`, `privacy_class`, and `trust_level`. Do not let retrieved memory override system instructions, tool policy, approval rules, or security controls.

Failure Modes

- Raw transcripts are stored as durable memory.
- The model writes memory silently without consent, policy, or trace.
- Untrusted web pages, emails, tickets, or documents poison future memory.
- Stale memory overrides fresher evidence.
- User preferences are treated as facts.
- Facts from one tenant, user, or project appear in another context.
- Sensitive data is stored without retention, deletion, or correction rules.
- Summaries lose the evidence needed to verify or repair the memory.
- The system has no way to show users what is remembered.

- Memory grows until retrieval becomes noisy and expensive.

Evaluation Strategy

Memory evals should test both recall and restraint. A memory system that remembers everything is not good. It is risky.

- Test allowed memory writes from explicit user preferences.
- Test denied writes for sensitive data without consent.
- Test review-required writes from untrusted content.
- Test correction of an existing memory record.
- Test deletion and verify the record cannot be retrieved later.
- Test stale memory against fresher evidence.
- Test tenant and actor isolation.
- Test memory retrieval with conflicting records.
- Test whether retrieved memory is cited as memory, not treated as instruction.
- Test that memory writes appear in traces with policy decisions.

Measure write precision, write recall, unsafe-write rate, stale-recall rate, correction success, deletion success, cross-tenant leakage, retrieval relevance, citation coverage, and policy-decision accuracy.

Production Checklist

- Define the memory types the system is allowed to store.
- Keep memory records typed, scoped, source-backed, and timestamped.
- Require consent for sensitive or personal memory.
- Filter by tenant, actor, permission, freshness, and retention before ranking by relevance.
- Separate retrieved memory from instructions and policy.
- Add write policy for untrusted sources, sensitive data, policy memory, and tool-result summaries.
- Provide correction and deletion paths.
- Trace memory reads, writes, updates, denials, approvals, and deletions.
- Convert unsafe memory writes and stale recalls into regression evals.
- Give operators a way to disable memory writes without disabling retrieval.

Run the Example

```
npm run memory-augmented-agent
```

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

`memory-augmented-agent-pattern/autogen_typescript_example/memory_agent.ts`

[Open full source](#)

```
import dotenv from 'dotenv';
dotenv.config();
import axios from 'axios';
import readline from 'readline';
import fs from 'fs';

type MemoryMessage = { role: string; content: string };

const MISTRAL_API_KEY = process.env.MISTRAL_API_KEY;
const MISTRAL_API_URL = 'https://api.mistral.ai/v1/chat/completions';
const MEMORY_FILE = './memory-augmented-agent-pattern/autogen_typescript_example/memory.json';

if (!MISTRAL_API_KEY) {
  console.error('Please set MISTRAL_API_KEY in your .env file');
  process.exit(1);
}

function loadMemory(): MemoryMessage[] {
  if (fs.existsSync(MEMORY_FILE)) {
    return JSON.parse(fs.readFileSync(MEMORY_FILE, 'utf-8')) as MemoryMessage[];
  }
  return [];
}

function saveMemory(memory: MemoryMessage[]) {
  fs.writeFileSync(MEMORY_FILE, JSON.stringify(memory, null, 2));
}

async function askMistral(messages: any[]) {
  const response = await axios.post(
    MISTRAL_API_URL,
    {
      model: 'mistral-tiny',
      messages,
```

```

    },
    {
      headers: {
        'Authorization': `Bearer ${MISTRAL_API_KEY}`,
        'Content-Type': 'application/json',
      },
    }
  );
  return response.data.choices[0].message.content.trim();
}

async function main() {
  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout,
  });

  let memory: MemoryMessage[] = loadMemory();

  rl.question('Ask the agent a question: ', async (userInput) => {
    // Retrieve relevant memory (for demo, just concatenate all previous user/assistant
    messages)
    let context = memory.map((m: MemoryMessage) => `${m.role}: ${m.content}`).join('\n');
    let messages = [
      { role: 'system', content: 'You are a helpful assistant with memory. Use the
following context from previous interactions if relevant:\n' + context },
      { role: 'user', content: userInput },
    ];

    let answer = await askMistral(messages);
    console.log('\nAgent Answer (with memory):\n', answer);

    // Update memory
    memory.push({ role: 'user', content: userInput });
    memory.push({ role: 'assistant', content: answer });
    saveMemory(memory);

    rl.close();
  });
}

main();

```


Open full source

```
import os
import json
import requests

MEMORY_FILE = os.path.join(os.path.dirname(__file__), 'memory.json')

def load_memory():
    if os.path.exists(MEMORY_FILE):
        with open(MEMORY_FILE, 'r') as f:
            return json.load(f)
    return []

def save_memory(memory):
    with open(MEMORY_FILE, 'w') as f:
        json.dump(memory, f, indent=2)

def ask_mistral(messages):
    api_key = os.getenv('MISTRAL_API_KEY')
    if not api_key:
        raise ValueError('Please set MISTRAL_API_KEY in your .env file')
    url = 'https://api.mistral.ai/v1/chat/completions'
    headers = {
        'Authorization': f'Bearer {api_key}',
        'Content-Type': 'application/json',
    }
    data = {
        'model': 'mistral-tiny',
        'messages': messages,
    }
    response = requests.post(url, headers=headers, json=data)
    response.raise_for_status()
    return response.json()['choices'][0]['message']['content'].strip()

def main():
    memory = load_memory()
    user_input = input('Ask the agent a question: ')
    # Retrieve relevant memory (for demo, just concatenate all previous user/assistant
    messages)
    context = '\n'.join([f"{m['role']}: {m['content']}" for m in memory])
    messages = [
        {'role': 'system', 'content': 'You are a helpful assistant with memory. Use the
following context from previous interactions if relevant:\n' + context},
```

```

        {'role': 'user', 'content': user_input},
    ]
    answer = ask_mistral(messages)
    print('\nAgent Answer (with memory):\n', answer)
    # Update memory
    memory.append({'role': 'user', 'content': user_input})
    memory.append({'role': 'assistant', 'content': answer})
    save_memory(memory)

if __name__ == '__main__':
    main()

```

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `memory-augmented-agent-pattern/` folder from this repository.

Related Patterns

- [Long-Term Episodic Memory](#)
- [Goals and State](#)
- [Context Engineering](#)
- [Policy Enforcement](#)
- [Human Approval Gates](#)
- [Pattern Evaluation Checklist](#)
- [Evaluation-Driven Agent Development](#)

Long-Term Episodic Memory

Long-term episodic memory stores events: what happened, when, who was involved, and why it mattered.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Long-term episodic memory stores events: what happened, when it happened, who or what was involved, what evidence supports it, and why it may matter later.

Episodic memory is not the same thing as semantic memory. Semantic memory stores facts or knowledge. Episodic memory stores remembered events. “The customer prefers email” is a preference or semantic claim. “On 2026-06-17 the customer asked to receive renewal notices by email” is an episode. Keeping that distinction clear makes memory easier to audit, correct, expire, and explain.

The goal is not to remember everything. The goal is to preserve useful events with enough provenance that future runs can reason from them without treating old, partial, or private information as universal truth.

Use When

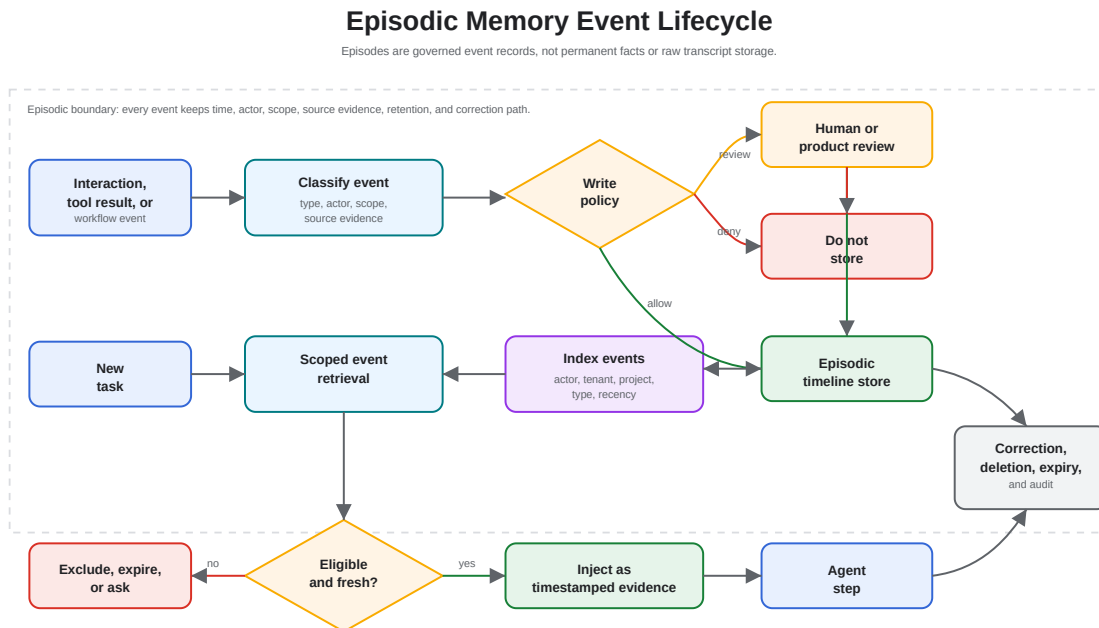
- The assistant needs continuity across sessions.
- Past events affect current decisions, personalization, support, operations, or project history.
- Events can be retrieved by relevance, recency, actor, tenant, project, and event type.
- The system can enforce retention, privacy, correction, deletion, and tenant isolation.
- You need an inspectable timeline rather than a vague user profile.

Avoid When

- The task only needs semantic facts, not event history.
- The system cannot explain or delete remembered events.
- The memory would store sensitive events without consent or retention limits.
- The agent would convert every interaction into a durable episode.
- The product cannot tolerate wrong attribution, stale recall, or cross-user leakage.

Architecture

Use this diagram to read Long-Term Episodic Memory as a system boundary, not only a code shape. The key ownership question is: the memory or retrieval layer owns long-lived knowledge, while the agent owns task-local working state.



Read it as an event lifecycle: store only policy-approved episodes, retrieve them by scope and freshness, and keep correction, deletion, expiry, and audit paths visible.

System Shape

- **Pattern boundary:** the episodic memory boundary owns event classification, write policy, timeline storage, retrieval scope, retention, correction, deletion, and audit.
- **State owner:** the memory service owns durable event records; the agent owns only proposed event writes and scoped event reads.
- **Model role:** the model can summarize and classify candidate events, but the runtime decides whether the event is worth storing.
- **Policy boundary:** event writes are checked for consent, privacy class, source trust, tenant scope, retention, and review requirements.
- **Operational promise:** future runs can use past events without confusing them with current instructions or permanent facts.

Core Protocol

1. Observe an interaction, tool result, workflow transition, human decision, correction, or external event.
2. Decide whether the event is worth remembering.
3. Classify the event type, actor, tenant, project, resources, source evidence, privacy class, confidence, and retention rule.
4. Run episodic memory write policy.

- 5. Store approved events in a timeline and, when useful, an index for semantic retrieval.
- 6. Retrieve events only after filtering by tenant, actor, permissions, event type, recency, and retention.
- 7. Inject retrieved events as evidence with timestamps and source references.
- 8. Allow correction, deletion, expiry, and audit review.

Implementation Notes

Episodic memory should be event-shaped. If the record does not include time, actor, source, and scope, it is probably not an episode. It is a note.

Event Types

Event Type	Example	Notes
User preference stated	User asked for PDF reports instead of slides.	May require consent depending on sensitivity.
Correction received	User corrected a project name or contact role.	Should update or supersede older events.
Task milestone	Agent completed a deployment plan review.	Useful for project continuity.
Human approval	Finance approved refund request <code>R-1024</code> .	Keep approval ID and policy version.
Tool-observed event	CRM returned account status changed to active.	Store source and freshness.
Incident or failure	Agent failed because a tool timed out.	Useful for reliability and evals.

Event Record

A useful episodic record keeps event content separate from evidence and retrieval metadata.

```
type EpisodicEventType =
  | "user_preference_stated"
  | "correction_received"
  | "task_milestone"
  | "human_approval"
  | "tool_observed_event"
  | "incident_or_failure";

type EpisodicMemoryEvent = {
  eventId: string;
  eventType: EpisodicEventType;
  occurredAt: string;
```

```

recordedAt: string;
actorId: string;
tenantId: string;
projectId?: string;
resourceRefs: string[];
summary: string;
sourceRefs: string[];
sourceTrust: "user_provided" | "tool_result" | "approved_source" | "operator_entered" |
"untrusted_content";
confidence: "low" | "medium" | "high";
privacyClass: "public" | "internal" | "private" | "sensitive";
retention: {
  expiresAt?: string;
  deleteOnRequest: boolean;
};
supersedesEventIds: string[];
correctionPath: string;
policyVersion: string;
};

```

Write Policy

The model should not silently write an event because it sounds useful. The runtime should decide.

```

type EpisodicWriteDecision =
  | { decision: "allow"; reason: string }
  | { decision: "deny"; reason: string }
  | { decision: "review"; reason: string; approverRole: string };

function decideEpisodicWrite(event: EpisodicMemoryEvent): EpisodicWriteDecision {
  if (event.sourceTrust === "untrusted_content") {
    return { decision: "review", reason: "untrusted_event_source", approverRole:
"memory_reviewer" };
  }

  if (event.privacyClass === "sensitive" && !event.retention.expiresAt) {
    return { decision: "deny", reason: "sensitive_event_requires_expiry" };
  }

  if (event.eventType === "human_approval" && !event.resourceRefs.length) {
    return { decision: "deny", reason: "approval_event_missing_resource" };
  }
}

```

```
return { decision: "allow", reason: "episodic_write_policy_passed" };  
}
```

Retrieval Rules

Episodic retrieval should start with scope, not similarity. Filter by tenant, actor, project, permissions, event type, retention, and recency before embedding search or ranking. Then rank events by relevance, recency, source trust, and confidence.

Retrieved events should be introduced as events, not facts:

```
Past event, not current instruction:  
- occurred_at: 2026-06-17T09:20:00Z  
- event_type: correction_received  
- summary: User corrected the deployment target from staging to production.  
- confidence: high  
- source: support-ticket-1842
```

That framing matters. The event may explain context, but it should not override current user intent, current policy, fresh tool data, or approval rules.

Failure Modes

- The system stores every interaction as an event and retrieval becomes noisy.
- An old event is treated as a current fact.
- A preference event becomes a permanent user trait.
- A false event is written from a hallucinated summary.
- An event is attributed to the wrong user, tenant, project, or resource.
- Sensitive events are retained indefinitely.
- Corrections create new conflicting events instead of superseding older ones.
- Events from untrusted documents poison future behavior.
- Timeline order is wrong because `recordedAt` is confused with `occurredAt`.
- The system cannot show, correct, delete, or expire event records.

Evaluation Strategy

Episodic memory evals should test event write quality, retrieval quality, and restraint.

- Test that meaningful events are stored with actor, tenant, time, source, confidence, and retention.
- Test that routine chatter is not stored.
- Test stale event suppression when fresher evidence exists.
- Test correction and supersession of an older event.
- Test deletion and expiry.
- Test cross-tenant and cross-user isolation.
- Test wrong-attribution prevention.

- Test retrieval by recency, relevance, and event type.
- Test untrusted-source review before storage.
- Test that retrieved events do not override current instructions or policy.

Measure event-write precision, event-write recall, attribution accuracy, stale-event recall, correction success, deletion success, tenant-isolation failures, retrieval relevance, and unsafe-write rate.

Production Checklist

- Define which event types the system is allowed to remember.
- Store `occurredAt` and `recordedAt` separately.
- Keep actor, tenant, project, resource, source, confidence, privacy, and retention metadata on every event.
- Filter by scope before semantic ranking.
- Treat retrieved events as evidence, not instructions.
- Require review for sensitive, untrusted, or policy-significant events.
- Support correction, supersession, deletion, and expiry.
- Trace event reads, writes, denials, reviews, corrections, and deletions.
- Convert false memories and stale recalls into regression evals.
- Give users or operators an inspectable timeline when the product requires trust.

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

This pattern currently has no dedicated code excerpt. Use the source and download links below for the full pattern folder.

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `long-term-episodic-memory-agent-pattern/` folder from this repository.

Related Patterns

- [Memory-Augmented Agent](#)
- [Goals and State](#)
- [Context Engineering](#)

- Human Approval Gates
- Policy Enforcement

Semantic Recall and RAG

Semantic recall retrieves relevant material by meaning rather than exact keywords. RAG injects retrieved material into context before generation.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Context engineering decides what the model sees before it answers or acts. It assembles instructions, user input, working memory, retrieved evidence, tool results, policies, examples, budgets, and exclusions into a controlled working set.

RAG is not “put search results in the prompt.” It is an evidence pipeline. Context engineering is the larger discipline around that pipeline. The system must decide which sources are eligible, what counts as fresh enough, what the caller may see, how retrieved content is cited, what gets omitted, what gets pinned, and what the agent should do when evidence is missing or conflicting.

The rule is simple: context is assembled, not dumped. The model should see a deliberate packet, not a pile of chat history, retrieved text, tool output, and memory fragments.

This pattern owns the assembly boundary, not every source system inside it. Working memory owns run state. Durable memory owns what survives across runs. Retrieval owns evidence discovery. Context engineering decides which of those records enters the next model call.

Use When

- The answer depends on a large, changing, or private knowledge base.
- Relevant sources can be chunked, embedded, filtered, cited, and inspected.
- The retrieval layer can enforce tenant, role, source, and freshness constraints.
- The agent can refuse or escalate when evidence is missing.
- The system can evaluate retrieval quality separately from final-answer quality.
- The agent needs working memory, tool results, policy, and retrieved evidence in one coherent packet.

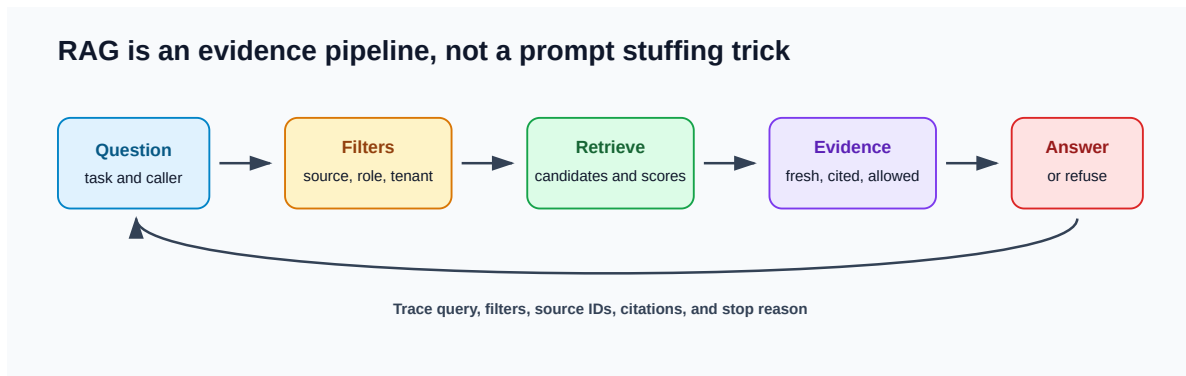
Avoid When

- The required knowledge is already in the task input or deterministic system state.
- The corpus is too noisy, stale, or untrusted to retrieve safely.

- The system cannot distinguish trusted metadata from untrusted document content.
- The answer needs exact database state that should come from a typed tool, not semantic search.
- Citations, source IDs, or retrieval traces cannot be stored for review.
- The system would rely on raw transcript stuffing instead of a controlled context builder.

Architecture

Use this diagram to read Semantic Recall and RAG as a system boundary, not only a code shape. The key ownership question is: the memory or retrieval layer owns long-lived knowledge, while the agent owns task-local working state.



System Shape

- **Pattern boundary:** the context builder decides what may enter the model working set and how each part is labeled.
- **State owner:** the runtime owns working state, policies, budgets, retrieval traces, tool results, memory records, and context assembly records.
- **Model role:** the model uses the assembled packet to answer, act, ask for missing evidence, or explain why it cannot proceed.
- **Policy boundary:** source eligibility, tenant access, redaction, freshness, tool-result handling, memory writes, and instruction hierarchy run before generation.
- **Operational promise:** the model works from a small, labeled, policy-checked packet rather than vague memory or unfiltered context.

Core Protocol

1. Classify the context need: policy, task instructions, working state, tool result, user memory, event history, private record, public source, or example.
2. Load current goal, working memory, budget state, approval state, and stop conditions.
3. Apply caller, tenant, role, source, freshness, and data-handling filters before retrieval or inclusion.
4. Retrieve candidate chunks with metadata, not naked text.
5. Rank and trim candidates by relevance, freshness, diversity, trust, and token budget.
6. Build an evidence set with source IDs, citations, confidence, and known gaps.
7. Assemble context with instructions, policy, state, evidence, tool results, memory, examples, and exclusions separated.

8. Generate an answer, action, refusal, or escalation using only eligible context.
9. Record query, filters, source IDs, packet sections, omitted material, citations, and stop reason.

Implementation Notes

Treat retrieved material as evidence, not authority. A web page, email, ticket, PDF, or document can contain useful facts and malicious instructions at the same time.

Context Packet

A context packet is the runtime artifact that explains what the model was allowed to see.

```
type ContextSectionKind =  
  | "system_instruction"  
  | "policy"  
  | "user_request"  
  | "working_memory"  
  | "retrieved_evidence"  
  | "tool_result"  
  | "episodic_memory"  
  | "semantic_memory"  
  | "example"  
  | "exclusion";  
  
type ContextSection = {  
  kind: ContextSectionKind;  
  title: string;  
  content: string;  
  sourceRefs: string[];  
  trustLevel: "trusted" | "internal" | "user_supplied" | "public" | "untrusted";  
  freshness?: {  
    observedAt?: string;  
    expiresAt?: string;  
  };  
  tokenEstimate: number;  
};  
  
type ContextPacket = {  
  packetId: string;  
  runId: string;  
  goalId?: string;  
  actorId: string;  
  tenantId: string;  
  task: string;
```

```

sections: ContextSection[];
budget: {
  maxTokens: number;
  usedTokens: number;
  reservedOutputTokens: number;
};
omittedRefs: Array<{
  ref: string;
  reason: "not_relevant" | "stale" | "not_allowed" | "duplicate" | "over_budget";
}>;
policyVersion: string;
};

```

This packet should be traceable. If a run fails, you should be able to inspect what was included, what was omitted, and why.

Ordering And Trust Rules

Ordering is part of the control plane. A practical order is:

1. system instructions;
2. policy and safety constraints;
3. current user or workflow request;
4. goal and working memory;
5. approved tool results;
6. retrieved evidence;
7. retrieved memories;
8. examples;
9. explicit exclusions and known gaps.

The point is not only token order. The point is separation. Retrieved text, tool output, memory, and user documents should never be allowed to rewrite the system instructions, policy, tool permissions, approval rules, or memory write rules.

Evidence Contract

```

type EvidenceChunk = {
  sourceId: string;
  sourceType: 'policy' | 'docs' | 'ticket' | 'email' | 'memory' | 'web';
  tenantId?: string;
  trustLevel: 'trusted' | 'internal' | 'user_supplied' | 'public' | 'unknown';
  freshness: {
    retrievedAt: string;
    sourceUpdatedAt?: string;
    maxAgeDays?: number;
  };
};

```

```

};
permissions: {
  allowedRoles: string[];
  redaction: 'none' | 'pii' | 'secret' | 'tenant_scoped';
};
score: number;
excerpt: string;
citation: string;
};

```

Answer Contract

The evidence contract should travel with the answer:

```

type RagAnswer = {
  status: 'answered' | 'missing_evidence' | 'conflicting_evidence' | 'refused';
  answer?: string;
  citations: string[];
  evidenceRefs: string[];
  missingEvidence?: string[];
};

```

Eligibility Check

A small eligibility check catches many production failures:

```

function isEligibleEvidence(chunk: EvidenceChunk, callerRole: string, now: Date) {
  if (!chunk.permissions.allowedRoles.includes(callerRole)) return false;
  if (chunk.permissions.redaction === 'secret') return false;

  if (chunk.freshness.sourceUpdatedAt && chunk.freshness.maxAgeDays) {
    const updatedAt = new Date(chunk.freshness.sourceUpdatedAt).getTime();
    const ageDays = (now.getTime() - updatedAt) / 86_400_000;
    if (ageDays > chunk.freshness.maxAgeDays) return false;
  }

  return chunk.trustLevel !== 'unknown';
}

```

Do not let the model decide whether a source is allowed. The model can summarize evidence quality. Software should enforce eligibility.

Context Budgeting

Context budgeting is not just truncation. It is deciding what the model must see, what can be summarized, what can be retrieved again, what can be omitted, and what must be pinned.

Pin small, high-authority items: system instructions, policy constraints, active goal, stop condition, approval state, and the user request. Summarize bulky but low-risk history. Retrieve source evidence instead of carrying old copied excerpts. Drop duplicate chunks. Omit stale, unauthorized, or low-confidence material. Reserve output tokens before filling the input context.

When the budget is tight, the runtime should degrade explicitly:

- answer from fewer cited sources;
- ask a clarifying question;
- retrieve again with narrower filters;
- summarize intermediate state;
- refuse when required evidence cannot fit safely;
- escalate when policy requires complete evidence.

Failure Modes

- Stale but plausible evidence is retrieved and treated as current.
- Retrieved text contains instructions that override the system goal.
- Tool output is treated as a new instruction rather than as data.
- Memory overrides the current user request.
- Working state is stale but still included as current.
- Chunks from the wrong tenant, role, or customer enter the context.
- The retriever returns semantically similar but operationally wrong sources.
- Citations point to broad documents instead of the exact supporting chunk.
- The agent answers when the evidence is missing or conflicting.
- Memory writes store an unverified summary as if it were durable fact.
- Retrieval scores are logged, but filters, source IDs, and freshness are not.
- The final answer is evaluated, but retrieval quality is not.
- The context packet cannot be reconstructed after an incident.
- Important policy or approval state is omitted under token pressure.

Evaluation Strategy

Context evals should test retrieval, assembly, and answer behavior separately.

- Test known-answer questions where the correct source is present.
- Test missing-evidence cases where the agent should refuse or ask for help.
- Test stale-source cases where an older source conflicts with a newer one.
- Test conflicting-source cases where the answer must explain uncertainty.
- Test prompt injection inside retrieved documents.
- Test prompt injection inside tool results and memory records.
- Test tenant and permission boundaries.

- Test context budget pressure and verify policy, goal, and citations are not dropped.
- Test omitted-source behavior and verify the trace explains why material was excluded.
- Test citation coverage: every factual claim should map to source evidence.
- Test retrieval precision and recall before testing final prose quality.

A compact eval fixture can make the evidence requirement explicit:

```
{
  "case_id": "stale_refund_policy",
  "question": "Can a damaged item be refunded after 45 days?",
  "retrieved_sources": [
    { "source_id": "refund_policy_2024", "freshness": "stale" },
    { "source_id": "refund_policy_2026", "freshness": "current" }
  ],
  "expected": {
    "must_cite": ["refund_policy_2026"],
    "must_not_cite": ["refund_policy_2024"],
    "status": "answered",
    "checks": ["freshness", "citation_coverage", "no_untrusted_instructions"]
  }
}
```

Measure retrieval recall, retrieval precision, source freshness, packet completeness, context-token efficiency, citation faithfulness, missing-evidence refusal rate, prompt-injection resistance, tenant-boundary violations, and answer quality grounded in cited evidence.

Production Checklist

- Define eligible sources by tenant, role, source type, freshness, and data class.
- Keep source metadata with every retrieved chunk.
- Build a traceable context packet for each run.
- Separate instructions from retrieved facts in the assembled context.
- Keep tool results, memory, retrieved evidence, examples, and user content labeled separately.
- Redact or exclude sources before generation, not after.
- Pin policy, active goal, stop condition, and approval state before filling optional context.
- Require citations for factual claims.
- Refuse or escalate when evidence is missing, stale, or conflicting.
- Trace query, filters, source IDs, scores, packet sections, omitted refs, and final citations.
- Evaluate retrieval quality separately from answer quality.
- Review memory writes before storing retrieved summaries as durable facts.
- Version chunking, embedding model, retrieval filters, rerankers, prompts, and citation rules.
- Add regression evals for context poisoning, stale context, omitted policy, and token pressure.

The architectural rule is simple: every item in context needs a reason, a source, a trust level, and a budget. Continue with [Context Budgets and Working Sets](#) for packet limits and [Agentic RAG Systems](#) for system-level composition.

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

`context-engineering-pattern/langgraph_python_example/rag_example.py`

[Open full source](#)

```
"""
Context Engineering Example: Retrieval-Augmented Generation (RAG) with Mistral
Requirements: pip install langchain-community sentence-transformers faiss-cpu requests
Note: This is a minimal example using HuggingFace embeddings by default.
"""
import os
import requests
from typing import List

# Optionally load environment variables from a local .env if present
try:
    from dotenv import load_dotenv, find_dotenv # type: ignore
    load_dotenv(find_dotenv(usecwd=True), override=True)
except Exception:
    pass

# Dummy documents
docs = [
    {"content": "Agentic systems are autonomous AI systems."},
    {"content": "Prompt engineering improves LLM outputs."}
]

# Build vector store (in-memory for demo)
texts = [d["content"] for d in docs]
try:
```

```

from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS

embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
vectorstore = FAISS.from_texts(texts, embeddings)
retriever = vectorstore.as_retriever()

def retrieve(query: str, k: int = 3):
    return retriever.get_relevant_documents(query)
except Exception:
    class _MiniDoc:
        def __init__(self, text: str):
            self.page_content = text

    def retrieve(query: str, k: int = 3):
        query_terms = {term.strip(".,?!").lower() for term in query.split()}

        def score(text: str) -> int:
            text_terms = {term.strip(".,?!").lower() for term in text.split()}
            return len(query_terms & text_terms)

        ranked = sorted(texts, key=score, reverse=True)
        return [_MiniDoc(text) for text in ranked[:k]]

MISTRAL_API_KEY = os.getenv("MISTRAL_API_KEY")

def chat_mistral(messages):
    if not MISTRAL_API_KEY:
        user_message = messages[-1]["content"]
        context = user_message.split("Context:\n", 1)[-1].split("\n\nQuestion:", 1)[0]
        return f"Local fallback answer from retrieved context: {context}"

    resp = requests.post(
        "https://api.mistral.ai/v1/chat/completions",
        headers={
            "Authorization": f"Bearer {MISTRAL_API_KEY}",
            "Content-Type": "application/json",
        },
        json={
            "model": "mistral-large-latest",
            "messages": messages,
            "temperature": 0.2,
        },
        timeout=60,
    )

```

```

resp.raise_for_status()
data = resp.json()
return (data.get("choices") or [{}])[0].get("message", {}).get("content", "")

if __name__ == "__main__":
    query = "What are agentic systems?"
    retrieved = retrieve(query)
    context = "\n\n".join(d.page_content for d in retrieved)
    answer = chat_mistral([
        {"role": "system", "content": "Use the provided context to answer the question succinctly."},
        {"role": "user", "content": f"Context:\n{context}\n\nQuestion: {query}"},
    ])
    print("Answer:", answer)

```

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `context-engineering-pattern/` folder from this repository.

Related Patterns

- [Context Budgets and Working Sets](#)
- [Working Memory](#)
- [Memory-Augmented Agent](#)
- [Knowledge-Bound Agents](#)
- [Agentic RAG Systems](#)
- [Pattern Evaluation Checklist](#)

Working Memory

Working memory is compact, typed task state the agent can update and consult during a run.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

The Goals and State Pattern separates what the agent is trying to achieve from the mutable state it accumulates while working. Goals define success. State records progress, constraints, evidence, pending work, approvals, budget, and stop reasons.

This is the practical core of working memory. Working memory is not long-term memory. It is run-scoped operational state. It should be compact, typed, inspectable, and disposable unless another policy explicitly promotes part of it into durable memory.

Use this pattern for task-local state. Durable memory is handled by [Memory-Augmented Agent](#). Context assembly is handled by [Context Engineering](#).

The model can propose a next step, a state update, or a stop reason. The runtime owns the state transition. That boundary keeps the agent from silently rewriting what has happened, what remains, or why it is allowed to continue.

Use When

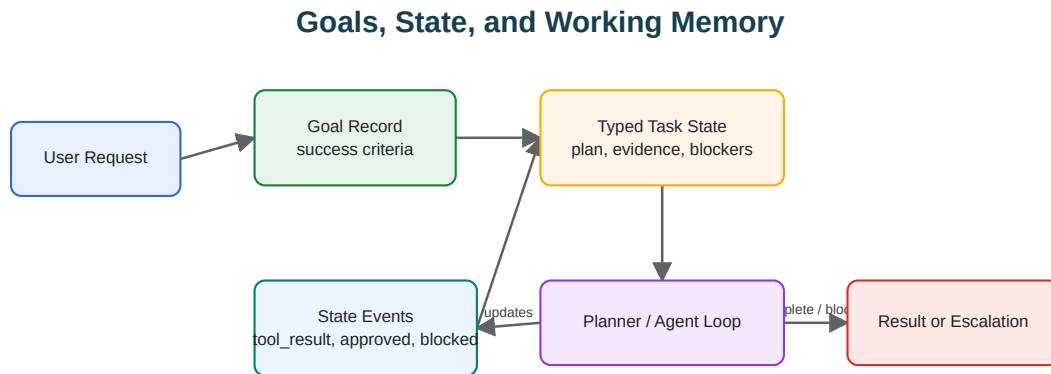
- A task spans multiple turns, tools, agents, retries, approvals, or workflow steps.
- You need resumable execution after failure, interruption, timeout, or human approval.
- The agent must explain progress against an explicit objective.
- You need a compact working set instead of raw chat history.
- Evaluation depends on trajectory, not only the final answer.

Avoid When

- The task is stateless and can be answered in one call.
- The goal cannot be expressed as observable success criteria.
- State would contain sensitive data you cannot store safely.
- The system would use state as an unstructured scratchpad with no owner.
- The runtime cannot replay or inspect state transitions.

Architecture

Use this diagram to read Working Memory as a system boundary, not only a code shape. The key ownership question is: the memory or retrieval layer owns long-lived knowledge, while the agent owns task-local working state.



System Shape

- **Pattern boundary:** the state boundary owns the goal, working state, event log, transition rules, stop reason, and replay record.
- **State owner:** the runtime, workflow engine, or application service owns state; the model owns only proposed updates.
- **Model role:** the model may summarize progress, identify gaps, and propose updates, but it should not be the source of truth for state.
- **Policy boundary:** state changes that affect permissions, approvals, memory promotion, or side effects pass through runtime checks.
- **Operational promise:** the system can explain what it is doing, what changed, what evidence caused the change, and why it stopped or continued.

Core Protocol

1. Create a goal record with success criteria, constraints, owner, risk class, and stop conditions.
2. Initialize working memory with the minimum state needed to start.
3. Execute one bounded step through a planner, model call, tool, worker, evaluator, or approval gate.
4. Convert observations into typed state events.
5. Apply state transitions idempotently.
6. Recompute open questions, pending work, budget state, approval state, and stop reason.
7. Continue, retry, ask for approval, escalate, or stop according to explicit rules.
8. Persist enough state and event history to replay or audit the run.

Implementation Notes

State should be smaller than the transcript and more structured than a summary. It is the operating model for the current run.

Goal Record

```
type AgentGoal = {
  goalId: string;
  runId: string;
  owner: string;
  description: string;
  successCriteria: string[];
  constraints: string[];
  riskClass: "low" | "medium" | "high" | "critical";
  status: "active" | "blocked" | "waiting_for_approval" | "completed" | "cancelled" |
  "failed";
  createdAt: string;
  updatedAt: string;
};
```

Working Memory State

```
type WorkingMemoryState = {
  runId: string;
  goalId: string;
  currentStep?: string;
  completedSteps: string[];
  pendingSteps: string[];
  openQuestions: string[];
  constraints: string[];
  evidenceRefs: string[];
  toolResultRefs: string[];
  approvalRefs: string[];
  budgetState: {
    iterationCount: number;
    tokenEstimate: number;
    costCents: number;
    deadlineAt?: string;
  };
  stopReason?: "success" | "blocked" | "approval_required" | "budget_exhausted" |
  "cancelled" | "failed";
  version: number;
};
```

State Events

State should change through events, not hidden mutation.

```
type StateEvent =  
  | { type: "goal_created"; goal: AgentGoal }  
  | { type: "step_started"; stepId: string; description: string }  
  | { type: "tool_result_recorded"; toolCallId: string; evidenceRef: string }  
  | { type: "approval_recorded"; approvalId: string; decision: "approved" | "denied" |  
    "expired" }  
  | { type: "question_opened"; question: string }  
  | { type: "question_answered"; question: string; evidenceRef: string }  
  | { type: "blocked"; reason: string }  
  | { type: "completed"; resultRef: string }  
  | { type: "cancelled"; reason: string };  
  
function applyStateEvent(state: WorkingMemoryState, event: StateEvent): WorkingMemoryState  
{  
  switch (event.type) {  
    case "step_started":  
      return { ...state, currentStep: event.stepId, version: state.version + 1 };  
    case "tool_result_recorded":  
      return {  
        ...state,  
        evidenceRefs: [...state.evidenceRefs, event.evidenceRef],  
        toolResultRefs: [...state.toolResultRefs, event.toolCallId],  
        version: state.version + 1,  
      };  
    case "approval_recorded":  
      return {  
        ...state,  
        approvalRefs: [...state.approvalRefs, event.approvalId],  
        stopReason: event.decision === "approved" ? undefined : "blocked",  
        version: state.version + 1,  
      };  
    case "completed":  
      return { ...state, stopReason: "success", version: state.version + 1 };  
    case "cancelled":  
      return { ...state, stopReason: "cancelled", version: state.version + 1 };  
    default:  
      return { ...state, version: state.version + 1 };  
  }  
}
```

This is intentionally simple. The important part is the rule: state transitions are explicit, typed, replayable, and tied to observations.

Running Case: Refund Goal And State

A refund assistant needs more than a chat transcript because operators must know what evidence the system used before money moved. The goal record should name the outcome and the risk; the working state should show the evidence, open questions, approval state, and stop reason.

```
goal:
  description: "Resolve refund eligibility for order 0-104."
  success_criteria:
    - "current refund policy is referenced"
    - "order and delivery evidence are attached"
    - "recommendation is validated by policy gate"
    - "high-value refund waits for approval"
  risk_class: "high"
state:
  completed_steps:
    - "order_loaded"
    - "delivery_status_loaded"
  open_questions:
    - "policy exception evidence is missing"
  evidence_refs:
    - "order:0-104"
    - "delivery:D-88"
    - "policy:refunds:v2026-06"
  approval_refs: []
  stop_reason: "approval_required"
```

This state tells the runtime what happened and what remains. It also gives evals something concrete to inspect: missing policy should block the recommendation, denied approval should block payment, and a resumed run should preserve the same evidence references.

Promotion Rules

Working memory should not automatically become durable memory. A completed step, tool result, or user correction may be a candidate for long-term memory, but promotion needs a separate policy decision.

Before promoting working memory, ask:

- Is this useful beyond the current run?
- Is the source trustworthy and referenced?
- Does the user or tenant allow it to be stored?
- Is there sensitive data that needs redaction or expiry?

- Is this an event, a fact, a preference, or a policy reference?
- Is there a correction and deletion path?

If the answer is unclear, keep it in the run trace and do not promote it.

Failure Modes

- The state becomes a transcript dump.
- The model silently rewrites state without an event.
- The goal describes activity, not success.
- Subgoals drift away from the parent goal.
- Tool results are summarized without evidence references.
- Approval, cancellation, or budget state is lost after retry.
- The system cannot resume because state only lived in model context.
- Stale state overrides fresh tool results.
- Sensitive state is persisted without retention or redaction rules.
- Working memory is automatically promoted to long-term memory.

Evaluation Strategy

Working-memory evals should inspect the trajectory, not only the final answer.

- Test that a multi-step task creates a goal with observable success criteria.
- Test that each tool result becomes a state event with an evidence reference.
- Test retry and verify idempotent state transitions.
- Test resume after interruption.
- Test cancellation and verify the run does not continue.
- Test approval waits and verify approval state survives resume.
- Test budget exhaustion and verify the stop reason is recorded.
- Test stale tool results and verify fresh evidence wins.
- Test blocked runs and verify open questions are preserved.
- Test that working memory is not promoted to durable memory without policy.

Measure state completeness, transition validity, replay success, resume success, stale-state rate, lost-approval rate, stop-reason accuracy, and promotion-policy accuracy.

Production Checklist

- Define goal schema, state schema, event schema, and stop reasons.
- Keep working memory compact and typed.
- Store state separately from chat history.
- Make state updates event-based, idempotent, and replayable.
- Attach evidence references to tool results, decisions, approvals, and final answers.
- Persist approval, cancellation, budget, retry, and blocked state.
- Treat state promotion to durable memory as a separate policy decision.

- Redact or avoid sensitive data in working memory when possible.
- Trace state reads, writes, transitions, retries, resumes, and stop reasons.
- Convert lost-state and wrong-stop incidents into regression evals.

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

This pattern currently has no dedicated code excerpt. Use the source and download links below for the full pattern folder.

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `goals-and-state-pattern/` folder from this repository.

Related Patterns

- [Agent Loop](#)
- [Planning Pattern](#)
- [Durable Workflow](#)
- [Memory-Augmented Agent](#)
- [Long-Term Episodic Memory](#)
- [Human Approval Gates](#)

Knowledge-Bound Agents

Knowledge-bound agents ground answers and actions in approved sources, policies, and citation rules.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Policy enforcement is the software-owned boundary that decides whether a model-proposed answer, tool call, data access, memory write, or side effect is allowed. The model can propose an action. The runtime decides whether to allow, deny, require approval, escalate, or audit it.

Policy should not live only in prompt text. Prompts can explain policy to the model, but enforcement belongs in code, workflow, tool manifests, access-control systems, and auditable decision records.

Knowledge-bound agents use the same idea for answers: the model should answer only from approved sources, cite the evidence, and refuse or escalate when the required source is missing, stale, forbidden, or conflicting.

The practical rule is: policy runs before authority. Before retrieval, before memory writes, before tool execution, before external communication, and before final answers in regulated or evidence-bound domains, the runtime should know whether the action is allowed.

Use When

- Actions must be checked before execution.
- The agent handles private, regulated, security-sensitive, or business-critical data.
- The system needs approved sources, citations, or compliance constraints.
- Policy decisions must be auditable and replayable.
- The runtime can identify actor, resource, action, capability, risk, and context.
- Tool calls, memory writes, retrieval, final answers, or workflow transitions require different rules by task class.
- A human approval path exists for actions that are valid but too risky to execute autonomously.

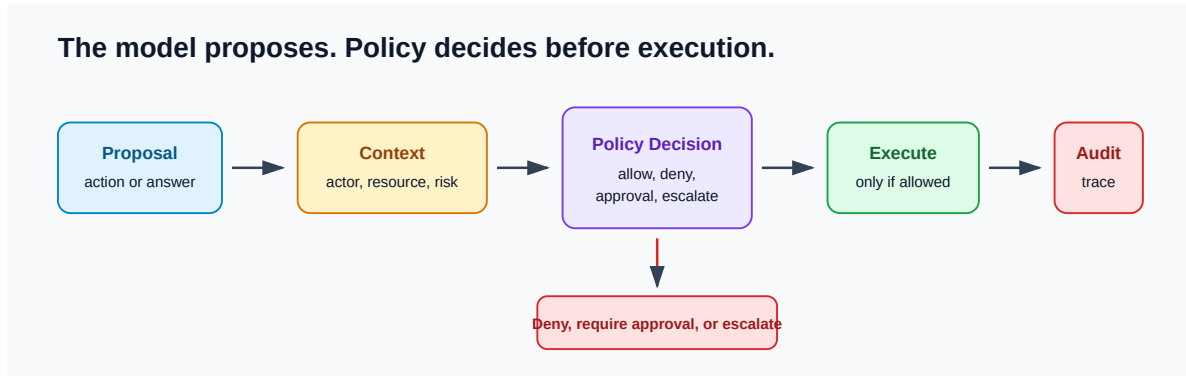
Avoid When

- Policy is only written as prompt guidance with no runtime check.
- The system cannot identify the actor, resource, action, and context.
- Policy checks happen after irreversible actions.

- Exceptions are silent, unreviewed, or missing from traces.
- Approved knowledge sources cannot be identified, updated, or cited.
- The runtime cannot stop, pause, or change execution after a policy decision.

Architecture

Use this diagram to read Knowledge-Bound Agents as a system boundary, not only a code shape. The key ownership question is: the memory or retrieval layer owns long-lived knowledge, while the agent owns task-local working state.



System Shape

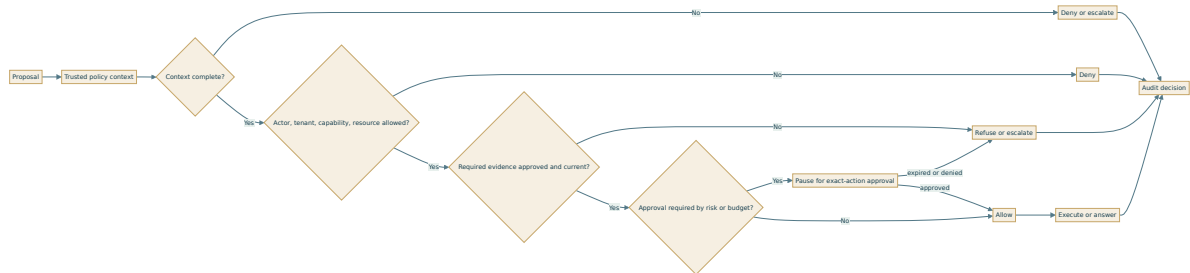
- **Pattern boundary:** the policy boundary evaluates proposed actions, data access, answers, and memory writes before they take effect.
- **State owner:** the runtime owns policy context, decision records, policy version, trace ID, and enforcement outcome.
- **Model role:** the model proposes an action or answer and may explain risk, but it does not grant itself permission.
- **Knowledge boundary:** approved sources, freshness, citations, and refusal rules define what the agent may claim.
- **Budget boundary:** policy can require approval, downgrade capability, or stop when a run has exceeded the spend or autonomy allowed for its risk class.
- **Operational promise:** policy decisions happen before execution and are visible after the run.

Core Protocol

1. Receive a proposed action, answer, tool call, retrieval result, or memory write.
2. Build policy context: actor, caller, tenant, resource, capability, risk, evidence, and trace ID.
3. Evaluate policy before the action executes or the answer is returned.
4. Return a decision: allow, deny, require approval, escalate, or audit-only.
5. If approval is required, pause through the approval gate.
6. Execute only decisions that are allowed or approved.
7. Record decision, reason, policy version, actor, resource, action, and trace ID.
8. Feed serious denials, misses, and overrides into regression evals.

Implementation Notes

Use this decision flow when reviewing where policy runs. Every branch should produce a trace event with the policy version, reason, actor, resource, capability, and execution effect.



A policy decision should be a typed runtime object.

```
type PolicyOutcome = 'allow' | 'deny' | 'require_approval' | 'escalate' | 'audit';

type PolicyDecision = {
  actionId: string;
  actor: {
    id: string;
    role: string;
    tenantId?: string;
  };
  resource: {
    type: 'customer_record' | 'refund' | 'email' | 'memory' | 'document';
    id: string;
    tenantId?: string;
  };
  capability: 'read' | 'write' | 'send' | 'refund' | 'remember' | 'answer';
  riskLevel: 'low' | 'medium' | 'high' | 'critical';
  decision: PolicyOutcome;
  reason: string;
  requiredApproval?: {
    approverRole: string;
    approvalPolicy: string;
  };
  policyVersion: string;
  traceId: string;
};
```

The policy context should come from trusted runtime state, not only from model text:

```
type PolicyContext = {
  actionId: string;
```

```

traceId: string;
actorRole: string;
actorTenant?: string;
resourceTenant?: string;
capability: PolicyDecision['capability'];
riskLevel: PolicyDecision['riskLevel'];
toolName?: string;
evidenceStatus?: 'present' | 'missing' | 'stale' | 'forbidden';
budgetState: 'within_budget' | 'approval_threshold' | 'exhausted';
hasHumanApproval: boolean;
policyVersion: string;
};

```

The enforcement function should run before retrieval, memory write, tool call, side effect, or final answer:

```

function enforcePolicy(input: PolicyContext): Pick<PolicyDecision, 'decision' | 'reason'>
{
  if (input.actorTenant && input.resourceTenant && input.actorTenant !==
input.resourceTenant) {
    return { decision: 'deny', reason: 'tenant_boundary' };
  }

  if (input.budgetState === 'exhausted') {
    return { decision: 'escalate', reason: 'budget_exhausted' };
  }

  if (input.budgetState === 'approval_threshold' && !input.hasHumanApproval) {
    return { decision: 'require_approval', reason: 'budget_approval_required' };
  }

  if (input.capability === 'refund' && input.riskLevel === 'high') {
    return { decision: 'require_approval', reason: 'high_risk_refund' };
  }

  if (input.capability === 'send' && input.actorRole !== 'support_agent') {
    return { decision: 'deny', reason: 'role_not_allowed' };
  }

  if (input.capability === 'answer' && input.evidenceStatus !== 'present') {
    return { decision: 'escalate', reason: 'required_evidence_not_available' };
  }

  if (input.capability === 'remember' && input.riskLevel !== 'low') {
    return { decision: 'require_approval', reason: 'memory_write_requires_review' };
  }
}

```

```

    }

    return { decision: 'allow', reason: 'policy_passed' };
  }
}

```

For knowledge-bound answers, policy also decides whether the evidence is allowed:

```

type SourcePolicy = {
  sourceId: string;
  approved: boolean;
  freshness: 'current' | 'stale' | 'unknown';
  citationRequired: boolean;
  allowedTenant?: string;
};

```

The model can explain why an action looks safe. The runtime still makes the decision.

Where Policy Runs

Boundary	Policy Question
Retrieval	Is this actor allowed to read these sources for this task?
Tool call	Is this tool allowed for the actor, tenant, resource, risk, and budget?
Memory write	Is this memory safe, scoped, useful, and allowed to persist?
Human approval	Is this action allowed only after review, and who can approve it?
Final answer	Is the answer supported by approved evidence and safe to return?
Workflow transition	Is the next step valid after the current state and policy decision?

Treat each policy decision as a runtime event. It should have a trace ID, policy version, input summary, decision, reason, and execution effect.

Failure Modes

- Policy exists only in the system prompt.
- Policy runs after a tool has already executed.
- The decision lacks actor, resource, tenant, or capability context.
- A retry bypasses policy because the first attempt was checked.
- Policy versions change but traces do not record which version applied.
- Denials are not logged, so operators cannot see attempted unsafe actions.
- Approval-required actions are treated as allowed.
- Knowledge answers cite unapproved, stale, or inaccessible sources.
- Exceptions become permanent undocumented policy holes.

- Policy checks ignore budget state, so an agent can keep spending after approval should be required.
- Memory writes bypass policy because they are treated as harmless context management.
- Final answers bypass policy even when the domain requires approved evidence.

Evaluation Strategy

Policy evals should test allowed, denied, approval-required, and escalation paths.

- Test allowed low-risk actions.
- Test denied actions across role, tenant, resource, and capability boundaries.
- Test approval-required actions before side effects.
- Test stale or unapproved sources in knowledge-bound answers.
- Test retries and resumed workflows to ensure policy is applied every time.
- Test missing actor, resource, or tenant context.
- Test policy version changes and audit completeness.
- Test production incidents as replayable policy fixtures.
- Test budget-threshold cases that require approval before more work.
- Test memory-write denials, approvals, and tenant scoping.
- Test final-answer refusal or escalation when required evidence is missing.

A compact policy eval can look like this:

```
{
  "case_id": "cross_tenant_customer_record_read",
  "proposed_action": {
    "actor_tenant": "tenant_a",
    "resource_tenant": "tenant_b",
    "capability": "read",
    "resource_type": "customer_record",
    "budget_state": "within_budget",
    "evidence_status": "present"
  },
  "expected": {
    "decision": "deny",
    "reason": "tenant_boundary",
    "must_not_execute": true,
    "required_trace_fields": ["actor", "resource", "policy_version", "trace_id"]
  }
}
```

Measure policy decision accuracy, false allow rate, false denial rate, approval-routing accuracy, tenant-boundary violations, stale-policy use, denial logging completeness, and recurrence of known policy failures.

For production systems, false allow is usually the most dangerous metric. A false denial may frustrate a user. A false allow may leak data, move money, send the wrong message, or create an incident.

Production Checklist

- Enforce policy before execution, answer return, memory write, or external communication.
- Build policy context from trusted runtime data, not only model text.
- Return explicit allow, deny, require-approval, escalate, or audit decisions.
- Record actor, resource, capability, reason, policy version, and trace ID.
- Apply policy on retries and resumed workflows.
- Keep policy versions, tool manifests, source rules, and approval rules versioned.
- Treat missing policy context as deny or escalate.
- Apply policy to memory writes and final answers, not only tools.
- Connect policy decisions to runtime budget state.
- Add dashboards for denials, approvals, overrides, and false allows.
- Convert policy misses into regression evals.
- Review exceptions and expire them intentionally.

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

`compliance-policy-enforcer-agent/policy_contract.ts`

[Open full source](#)

```
export type PolicyOutcome = 'allow' | 'deny' | 'require_approval' | 'escalate';

export type PolicyContext = {
  traceId: string;
  actor: {
    id: string;
    role: 'support_agent' | 'finance_reviewer' | 'viewer';
    tenantId: string;
  };
  resource: {
    type: 'customer_record' | 'refund' | 'email' | 'memory' | 'document';
```

```

    id: string;
    tenantId: string;
};

capability: 'read' | 'write' | 'send' | 'refund' | 'remember' | 'answer';
riskLevel: 'low' | 'medium' | 'high' | 'critical';
evidenceStatus: 'present' | 'missing' | 'stale' | 'forbidden';
budgetState: 'within_budget' | 'approval_threshold' | 'exhausted';
hasHumanApproval: boolean;
policyVersion: string;
};

export type PolicyDecision = {
    traceId: string;
    policyVersion: string;
    decision: PolicyOutcome;
    reason: string;
    executionAllowed: boolean;
    requiredApprovalRole?: 'finance_reviewer' | 'security_reviewer' | 'manager';
};

export function enforcePolicy(context: PolicyContext): PolicyDecision {
    const base = {
        traceId: context.traceId,
        policyVersion: context.policyVersion
    };

    if (context.actor.tenantId !== context.resource.tenantId) {
        return {
            ...base,
            decision: 'deny',
            reason: 'tenant_boundary',
            executionAllowed: false
        };
    }

    if (context.evidenceStatus === 'forbidden') {
        return {
            ...base,
            decision: 'deny',
            reason: 'evidence_forbidden',
            executionAllowed: false
        };
    }
}

```

```

if (context.evidenceStatus === 'missing' || context.evidenceStatus === 'stale') {
  return {
    ...base,
    decision: 'escalate',
    reason: 'required_evidence_not_current',
    executionAllowed: false
  };
}

if (context.budgetState === 'exhausted') {
  return {
    ...base,
    decision: 'escalate',
    reason: 'budget_exhausted',
    executionAllowed: false
  };
}

if (context.capability === 'refund' && context.riskLevel !== 'low' &&
!context.hasHumanApproval) {
  return {
    ...base,
    decision: 'require_approval',
    reason: 'refund_requires_review',
    executionAllowed: false,
    requiredApprovalRole: 'finance_reviewer'
  };
}

if (context.capability === 'send' && context.actor.role !== 'support_agent') {
  return {
    ...base,
    decision: 'deny',
    reason: 'role_cannot_send',
    executionAllowed: false
  };
}

```

Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `compliance-policy-enforcer-agent/` folder from this repository.

Related Patterns

- [Human Approval Gates](#)
- [Tool Capability Design](#)
- [Agent Threat Model](#)
- [Semantic Recall and RAG](#)
- [Production Runtime Overview](#)
- [Cost Controls and Runtime Budgets](#)
- [Production Evaluation Feedback Loops](#)
- [Pattern Evaluation Checklist](#)

Skills

Skills package procedural knowledge as discoverable, versioned folders of instructions, references, scripts, templates, assets, and tests.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

The Skills Pattern packages procedural knowledge as discoverable folders: concise instructions, references, scripts, templates, and tests that an agent loads only when relevant.

This folder includes a small release-notes skill package. It shows the shape a production skill should have:

`SKILL.md` for activation and procedure, references for policy, templates for stable output, fixtures for repeatable examples, and a TypeScript runner/test that a human or agent can execute.

Scenario

A platform team wants coding agents to prepare release notes, update a changelog, and collect verification evidence. A tool schema can describe “write release notes”, but it cannot carry the team’s release rubric, examples, required commands, source-link policy, and final checklist.

A skill is the better boundary. `SKILL.md` gives the activation rule and the short procedure. Reference files hold release policy and examples. Scripts collect version, diff, and test evidence. Templates keep the output shape stable. The agent loads deeper files only when the release task appears, so normal coding work does not pay the context cost.

The important question is whether the folder contains a repeatable procedure another engineer can review, run, version, and test.

Use When

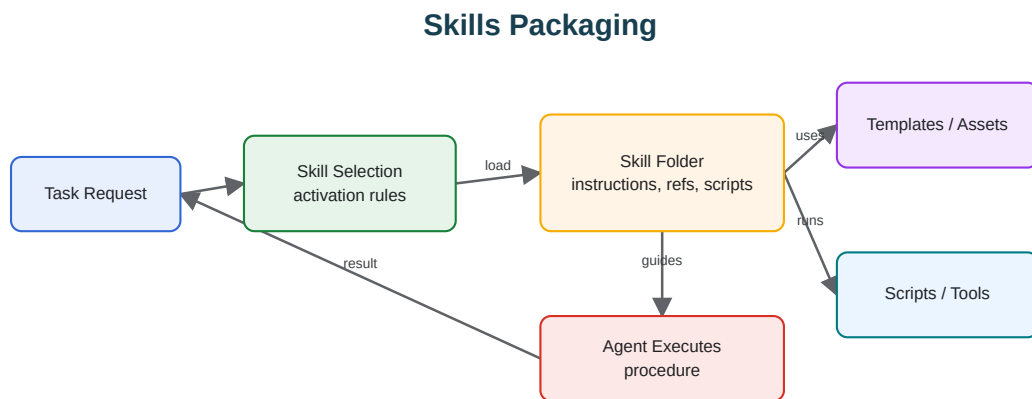
- A capability requires repeatable domain procedure rather than only a tool API.
- You want reusable know-how across agents, teams, or projects.
- The agent benefits from progressive disclosure: short instructions first, deeper references only when needed.

Avoid When

- A simple tool schema fully describes the capability.
- The skill would embed secrets, credentials, or unsafe scripts.
- The instructions are too vague to test with real tasks.

Architecture

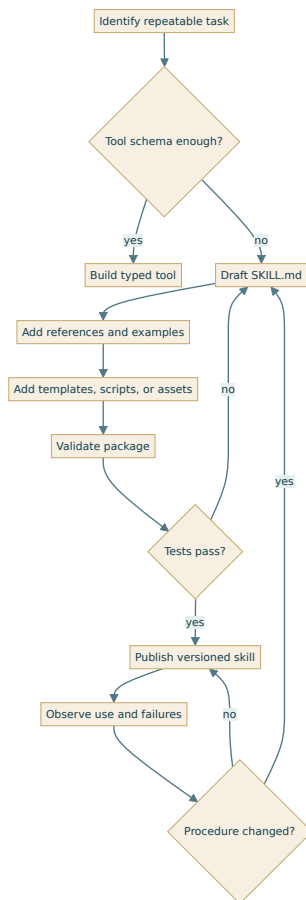
Use this diagram to read Skills as a system boundary, not only a code shape. The key ownership question is: the protocol or capability boundary owns schemas, permissions, invocation records, and response validation.



Decision Rules

Use a skill when the model needs procedural judgment around a capability. Use a tool when the model only needs to call a typed operation.

Need	Prefer	Reason
Call a stable API with typed input and output.	Tool	The schema and authorization rule carry the whole contract.
Follow a multi-step team procedure.	Skill	The agent needs instructions, examples, templates, and checks.
Produce a repeated artifact such as a PR note, ADR, report, or release packet.	Skill	Templates and examples reduce drift.
Execute a dangerous action.	Tool behind policy, not skill alone.	Skills may explain the procedure, but permissions must live outside prose.
Share capability across agents or teams.	Skill plus tested scripts.	Humans and agents need the same runnable contract.



System Shape

- **Pattern boundary:** the agent discovers or selects a capability, submits a typed request, and receives a typed result across a policy boundary.
- **State owner:** the protocol or capability boundary owns schemas, permissions, invocation records, and response validation.
- **Primary artifact:** `skills-pattern/` contains a reviewable release-notes skill package with activation instructions, policy reference, template, fixture, TypeScript runner, and tests.
- **Operational promise:** Skills package procedural knowledge as discoverable, versioned folders of instructions, references, scripts, templates, assets, and tests.
- **Runnable path:** start with `npm run skills:demo` before adapting the pattern to a larger system.

Contract

A production skill should have a small, reviewable anatomy.

Part	Owns	A++ Test
<code>SKILL.md</code>	Activation rule, short procedure, and routing to deeper material.	A new agent can decide when to use the skill without loading every reference.
<code>references/</code>	Domain policy, examples, rubrics, and edge cases.	References are scoped, current, and cited by the skill.

Part	Owns	A++ Test
<code>scripts/</code>	Repeatable collection, validation, generation, or formatting.	A human can run the scripts outside the agent loop.
<code>templates/</code>	Stable output shape for common artifacts.	Outputs stay consistent across runs and agents.
<code>tests/</code> or fixtures	Positive and negative cases.	Bad activation, missing inputs, unsafe outputs, and malformed artifacts fail.
Manifest or metadata	Version, owner, dependencies, permissions, and environment.	Reviewers can audit supply-chain and permission risk.

Bad Skill vs Production Skill

Weak Skill	Production Skill
"Use this for writing."	"Use this for release notes from verified engineering evidence."
One long instruction file.	Short <code>SKILL.md</code> plus routed references.
Vague advice and examples copied into context.	Templates, fixtures, and scripts with deterministic output.
Hidden shell assumptions.	Explicit commands, dependencies, inputs, outputs, and failure modes.
No negative cases.	Wrong-task activation, missing evidence, unsafe request, and malformed output tests.
Skill text implies permission.	Policy and execution authority live outside prose.

Core Protocol

1. Discover the capability, schema, permissions, and operating constraints.
2. Prepare a typed request from the current goal and state.
3. Authorize the request before invocation.
4. Invoke the tool, skill, or remote agent and validate the result.
5. Return structured output, refusal, progress, or error without losing correlation IDs.

Implementation Notes

- Keep `SKILL.md` short and route to deeper files only when needed.
- Bundle scripts and templates instead of asking the model to recreate fragile artifacts.
- Treat skills as supply-chain inputs: review, version, test, and restrict execution.
- Include examples of successful and unsuccessful use.
- Record owner, version, dependencies, and rollback path before distributing a skill.

- Prefer runnable scripts for fragile formatting, evidence collection, or validation.
- Keep credentials in platform stores or environment bindings, never inside skill files.

CLI-First Skills

A useful skill should be callable by both a human and an agent. A command-line interface is often the simplest shared contract:

- one command per capability;
- predictable subcommands such as `list`, `get`, `create`, and `run`;
- structured output for agents and readable output for humans;
- non-interactive defaults with explicit `--yes` or `--force` flags;
- credentials from environment or platform stores, not prompts hidden inside the command.

This keeps the skill testable outside the agent loop. If a human cannot run the skill directly and inspect the output, the agent will be harder to debug when the skill fails.

Failure Modes

- Skill descriptions that are too broad, causing irrelevant activation.
- Long instruction files that consume context before the task is understood.
- Hidden dependencies that only work on one machine.
- Malicious or outdated bundled scripts.
- Prompt text that silently expands tool or filesystem authority.
- References that conflict with `SKILL.md` or with each other.
- Templates that drift because no fixture proves the final artifact.
- Missing rollback path after a bad skill version ships.

Review Checklist

Before adding a skill to an agent environment, check:

- The description is narrow enough that the skill activates only for the right tasks.
- The first screen of instructions tells the agent what to do, what not to do, and which files to load next.
- Any script is deterministic, non-interactive by default, and safe to run with least privilege.
- Secrets come from platform stores or environment bindings, never from copied prose.
- The skill has at least one success example and one refusal or misuse example.
- The skill records enough evidence that a reviewer can reproduce the result.
- The skill can be disabled, versioned, or rolled back independently of the agent prompt.

Evaluation Strategy

- Test happy-path use, wrong-task activation, missing inputs, malformed artifacts, unsafe requests, and missing dependencies.
- Assert that the skill loads only the references required for the task.

- Verify generated artifacts against templates instead of only checking prose quality.
- Run scripts outside the agent loop so humans can reproduce failures.
- Track activation accuracy, evidence completeness, script failure rate, artifact validity, and rollback success.

Production Checklist

- Name the owner, version, dependency set, and supported runtime.
- Keep `SKILL.md` short enough for first-load use.
- Gate dangerous scripts behind external policy and approval.
- Validate inputs, outputs, template placeholders, and required evidence.
- Log skill name, version, loaded references, scripts run, artifacts written, and final status.
- Pin or roll back the skill independently of the agent prompt.

Use the online book's downloadable skill review checklist when reviewing a production skill package.

Run the Example

```
npm run skills:demo
npm run skills:test
```

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
skills-pattern/release-notes-skill/SKILL.md
```

[Open full source](#)

```
# Release Notes Skill
```

```
Use this skill when the task is to prepare release notes from verified engineering evidence.
```

```
## Activation
```

```
Use this skill for release notes, changelog entries, launch summaries, or publish-ready
```

update notes. Do not use it for product marketing copy, speculative roadmap notes, or incident reports.

Procedure

1. Read `references/release-policy.md` before drafting.
2. Load the evidence record from the caller or from `fixtures/release-evidence.json` for the demo.
3. Render the notes with `templates/release-notes.md`.
4. Include only shipped changes, verification evidence, known limits, and artifact links.
5. Refuse or return `needs\_evidence` when verification, owner, version, or artifact data is missing.

Do Not

- Invent test results, artifact links, owners, dates, or version numbers.
- Convert unresolved risks into shipped features.
- Use promotional language.
- Hide failed or skipped verification.

Expected Output

Return concise release notes with these sections: summary, shipped changes, verification, artifacts, known limits, and owner.

skills-pattern/typescript/src/skill_package.ts

[Open full source](#)

```
import fs from "node:fs/promises";
import path from "node:path";
import { fileURLToPath } from "node:url";

export type ReleaseEvidence = {
  version?: string;
  owner?: string;
  summary?: string;
  changes?: string[];
  verification?: string[];
  artifacts?: string[];
  knownLimits?: string[];
};

export type SkillPackage = {
```

```

    root: string;
    skillMarkdown: string;
    policyMarkdown: string;
    templateMarkdown: string;
};

export type SkillValidation = {
    status: "pass" | "fail";
    reasons: string[];
};

export type SkillRunResult = {
    status: "rendered" | "needs_evidence" | "blocked";
    validation: SkillValidation;
    output: string;
};

const requiredTemplateFields = [
    "version",
    "owner",
    "summary",
    "changes",
    "verification",
    "artifacts",
    "knownLimits",
];

const requiredEvidenceFields: Array<keyof ReleaseEvidence> = [
    "version",
    "owner",
    "summary",
    "changes",
    "verification",
    "artifacts",
];

export function defaultSkillRoot() {
    const current = path.dirname(fileURLToPath(import.meta.url));
    return path.resolve(current, "..", "..", "release-notes-skill");
}

export async function readSkillPackage(root = defaultSkillRoot()): Promise<SkillPackage> {
    const [skillMarkdown, policyMarkdown, templateMarkdown] = await Promise.all([
        fs.readFile(path.join(root, "SKILL.md"), "utf8"),
    ]

```

```

    fs.readFile(path.join(root, "references", "release-policy.md"), "utf8"),
    fs.readFile(path.join(root, "templates", "release-notes.md"), "utf8"),
  ]);

  return {
    root,
    skillMarkdown,
    policyMarkdown,
    templateMarkdown,
  };
}

export async function readEvidence(evidencePath: string): Promise<ReleaseEvidence> {
  return JSON.parse(await fs.readFile(evidencePath, "utf8")) as ReleaseEvidence;
}

export function validateSkillPackage(skillPackage: SkillPackage): SkillValidation {
  const reasons: string[] = [];
  const skillWordCount = skillPackage.skillMarkdown.split(/\s+/).filter(Boolean).length;

  if (!/^# Release Notes Skill/m.test(skillPackage.skillMarkdown)) reasons.push("missing skill title");
  if (!/## Activation/.test(skillPackage.skillMarkdown)) reasons.push("missing activation section");
  if (!/## Procedure/.test(skillPackage.skillMarkdown)) reasons.push("missing procedure section");
  if (!/## Do Not/.test(skillPackage.skillMarkdown)) reasons.push("missing negative constraints");
  if (skillWordCount > 260) reasons.push("SKILL.md is too long for first-load instructions");
  if (!/Required Evidence/.test(skillPackage.policyMarkdown)) reasons.push("policy missing required evidence");

  for (const field of requiredTemplateFields) {
    if (!skillPackage.templateMarkdown.includes(`${field}`)) {
      reasons.push(`template missing placeholder: ${field}`);
    }
  }
}

```

Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.

[skills-pattern/typescript/test/skill_package.spec.ts](#)

[Open full source](#)

```

import path from "node:path";
import {
  defaultSkillRoot,
  readSkillPackage,
  renderReleaseNotes,
  runReleaseNotesSkill,
  validateEvidence,
  validateSkillPackage,
  type ReleaseEvidence,
} from "../src/skill_package.ts";

function assert(condition: unknown, message: string): asserts condition {
  if (!condition) throw new Error(message);
}

const skillRoot = defaultSkillRoot();
const skillPackage = await readSkillPackage(skillRoot);
const validation = validateSkillPackage(skillPackage);

assert(validation.status === "pass", `Expected valid skill package:
${validation.reasons.join(", ")}`);
assert(skillPackage.skillMarkdown.includes("## Activation"), "Expected activation
section");
assert(skillPackage.policyMarkdown.includes("Required Evidence"), "Expected policy required
evidence");

const evidencePath = path.join(skillRoot, "fixtures", "release-evidence.json");
const result = await runReleaseNotesSkill({ skillRoot, evidencePath });
assert(result.status === "rendered", "Expected rendered release notes");
assert(result.output.includes("# Release Notes: 2026-06-21-skills-reference"), "Expected
release title");
assert(result.output.includes("npm run skills:test"), "Expected verification evidence");
assert(!result.output.includes("{"), "Expected all placeholders replaced");

const missingEvidence: ReleaseEvidence = {
  version: "missing-fields",
  changes: [],
};

const evidenceIssues = validateEvidence(missingEvidence);
assert(evidenceIssues.includes("missing evidence: owner"), "Expected owner evidence
issue");
assert(evidenceIssues.includes("missing evidence: verification"), "Expected verification
evidence issue");

```

```
const unsafeTemplate = "# Release Notes: {{version}}\n\n{{summary}}";
const rendered = renderReleaseNotes(unsafeTemplate, {
  version: "v1",
  owner: "maintainer",
  summary: "Shipped source-backed skill package.",
  changes: ["Added skill package."],
  verification: ["npm run skills:test"],
  artifacts: ["skills-pattern/release-notes-skill/SKILL.md"],
});
assert(rendered.includes("v1"), "Expected template replacement");
assert(!rendered.includes("{{summary}}"), "Expected summary replacement");

console.log("Skills package tests OK");
```

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `skills-pattern/` folder from this repository.

Related Patterns

- [MCP-first Tool Use](#)
- [Context Engineering](#)
- [Human-in-the-Loop Approval](#)

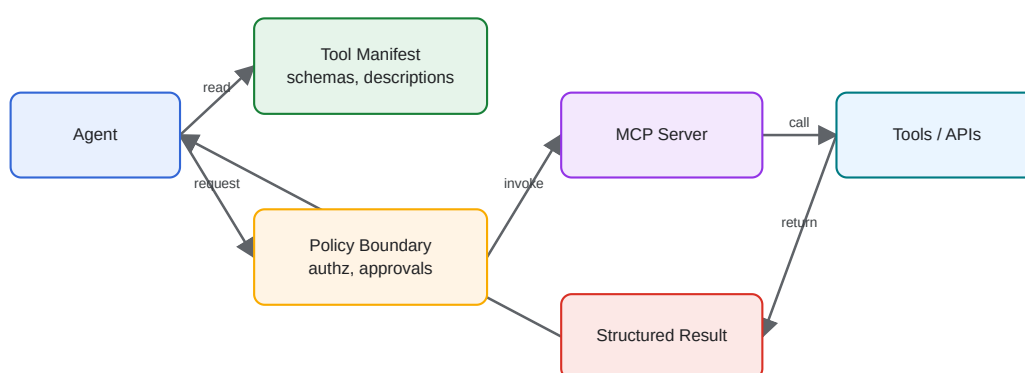
Tool Capability Design

Tools are where agentic systems become real. Before tools, the model can only produce text. After tools, it can read systems, write systems, spend money, send messages, change state, and trigger workflows. That is why tool design is not an implementation detail. It is architecture.

A strong agent does not need a large tool list. It needs the right tools, with clear contracts, narrow authority, useful errors, and observable results.

Download the [tool capability design review checklist](#) when reviewing a new tool surface, MCP server, or agent capability set.

MCP-first Tool Use



The Tool Surface Is A Control Plane

The tool surface decides what the agent can do, so do not treat tools as helper functions you happen to expose to a model. Treat the surface as a control plane with security, reliability, and product semantics.

Every tool should be able to answer what capability it exposes, who is allowed to call it, what inputs are valid, what state it can read, what state it can change, what side effects it can produce, what policy checks apply, what result shape it returns, and how it is traced, replayed, or audited. When those answers are not explicit, the system is leaning on prompt discipline to stay safe, which is the weakest place to put a boundary.

Narrow Tools Beat Broad Tools

Broad tools look convenient and are usually dangerous. The ones to avoid are the open-ended primitives:

- `run_sql(query);`
- `execute_shell(command);`
- `send_http_request(method, url, body);`
- `update_customer_record(fields);`

- `manage_ticket(action, payload)`.

These force the model to invent the safety boundary at runtime, which is exactly the wrong place for it. Prefer tools that encode the workflow instead:

- `lookup_customer_by_id(customer_id)`;
- `get_refund_eligibility(order_id)`;
- `draft_refund_request(order_id, reason)`;
- `submit_refund_for_approval(request_id)`;
- `create_ticket_reply_draft(ticket_id, body)`;
- `post_approved_ticket_reply(ticket_id, draft_id)`.

Narrow tools reduce prompt burden, improve evaluation, and make policy enforcement far easier, because the workflow lives in the tool surface rather than in the model's discretion.

Capability Classes

Classify every tool by capability.

Capability Class	Example	Risk
Read public data	search docs, fetch public page.	stale or untrusted information.
Read private data	customer lookup, internal database query.	privacy leak, tenant boundary failure.
Read untrusted content	email, webpage, user document, ticket comment.	prompt injection and hostile instructions.
Write internal state	update CRM, create ticket, save memory.	corrupted records or durable bad state.
External communication	send email, post message, submit form, webhook.	data exfiltration or unwanted communication.
Code or shell execution	run tests, execute script, modify files.	arbitrary side effects.
Money or entitlement change	refund, purchase, credit, permission grant.	financial or access-control impact.

This classification belongs in the tool manifest or registry, not only in prose.

Tool Risk Matrix

Capability classes are useful, but production systems also need an operational risk view. A tool may look harmless in isolation and become dangerous when combined with private data, untrusted content, external communication, or durable memory.

Tool Type	Primary Risk	Required Control
Read public data	stale, low-quality, or hostile content.	source metadata, freshness signal, and untrusted-content handling.
Read private data	privacy leak or tenant boundary failure.	scoped authorization, audit log, and output redaction.
Read untrusted content	prompt injection.	content isolation, instruction stripping, and policy checks before action.
Write business state	corrupted records or unintended workflow progress.	strict schema, idempotency key, policy decision, and trace.
External communication	data exfiltration or unwanted message delivery.	approval gate, recipient validation, and message audit.
Browser or code execution	arbitrary side effects and hidden network access.	sandbox, egress policy, timeout, and filesystem/network limits.
Memory write	durable poisoning or privacy retention failure.	memory policy, source attribution, retention class, and deletion path.
Credential use	confused deputy or privilege escalation.	short-lived credentials, scope binding, audience checks, and secret redaction.

The point is not to make every tool heavy. The point is to make the runtime know which tools are heavy. A public documentation search tool and a payment tool should not live behind the same policy, trace, retry, and approval rules.

Tool Manifest

A useful tool manifest is more than a name and a description.

Field	Purpose
Name	Stable identifier used in traces, evals, and policy.
Description	What the tool does, written for correct selection.
Input schema	Required fields, types, limits, enums, and validation rules.
Output schema	Structured result, refusal, partial result, or error.
Capability class	Read, write, communication, code execution, private data, untrusted content.
Side effects	What can change if the tool succeeds.
Idempotency	Whether repeated calls are safe and how duplicates are handled.
Permissions	Which roles, agents, routes, or tenants may call it.

Field	Purpose
Approval rule	When the call must be reviewed by a human or workflow.
Trace fields	Correlation ID, actor, resource, policy decision, and result.
Data handling	Redaction, retention, privacy class, and logging constraints.

If a tool is missing these fields, the runtime should treat it as high risk by default.

A compact manifest can make those boundaries explicit:

```

name: draft_refund_request
description: Drafts a refund request for human approval. It does not issue money.
capabilities:
  - read_private_data
  - write_internal_state
side_effects:
  - creates_refund_draft
permissions:
  roles:
    - support_agent
approval:
  required_for:
    - submit_refund
input_schema:
  type: object
  required:
    - order_id
    - reason
  properties:
    order_id:
      type: string
    reason:
      type: string
      maxLength: 500
output_schema:
  type: object
  required:
    - draft_id
    - status
  properties:
    draft_id:
      type: string
    status:

```

```

enum:
  - created
  - refused
  - needs_more_context
trace:
  fields:
    - run_id
    - actor_id
    - policy_version
    - approval_id

```

The tool name says what the tool does and what it does not do. A separate tool should submit the refund after approval.

For strongly typed systems, the same contract can be represented as code and registered with the tool runtime:

```

type CapabilityClass =
  | "read_public"
  | "read_private"
  | "read_untrusted"
  | "write_side_effect"
  | "external_communication"
  | "browser_or_code"
  | "memory_write"
  | "credential_use";

type ToolCapabilityManifest = {
  name: string;
  owner: string;
  version: string;
  description: string;
  inputSchemaRef: string;
  outputSchemaRef: string;
  risk: "low" | "medium" | "high" | "critical";
  capabilityClasses: CapabilityClass[];
  requiredScopes: string[];
  sideEffects: "none" | "draft" | "external_write" | "money_movement" | "message_send";
  idempotencyRequired: boolean;
  approvalRequired: boolean;
  timeoutMs: number;
  egress: {
    allowedDomains: string[];
    allowPrivateNetwork: boolean;
  };
}

```

```

};
observability: {
  requiredTraceFields: string[];
  redactFields: string[];
};
};

const refundDraftTool: ToolCapabilityManifest = {
  name: "draft_refund_request",
  owner: "support-platform",
  version: "2026-06-17",
  description: "Creates a refund draft for human approval. It does not issue money.",
  inputSchemaRef: "schemas/refund-draft-input.json",
  outputSchemaRef: "schemas/refund-draft-output.json",
  risk: "high",
  capabilityClasses: ["read_private", "write_side_effect"],
  requiredScopes: ["orders:read", "refunds:draft"],
  sideEffects: "draft",
  idempotencyRequired: true,
  approvalRequired: false,
  timeoutMs: 5000,
  egress: {
    allowedDomains: [],
    allowPrivateNetwork: false,
  },
  observability: {
    requiredTraceFields: ["run_id", "actor_id", "order_id", "policy_version"],
    redactFields: ["customer_email", "payment_token"],
  },
};

```

This looks bureaucratic only until the first incident. After that, it becomes the map of what the agent was allowed to do, which policy version was active, which data crossed the boundary, and which team owns the fix.

Agent-Friendly Interfaces

A tool designed for humans is not always a tool designed for agents. Agent-friendly tools have explicit schemas, short and specific descriptions, stable names, examples of valid and invalid calls, clear errors, structured outputs, bounded result sizes, correlation IDs, and machine-readable status values, with no hidden global state and no surprise side effects.

Error messages carry more weight than people expect. A vague `failed` makes the model guess. A useful error says what failed, whether retry is safe, and which field needs correction, which is the difference between a clean recovery and a confused retry loop.

Agent-First Tool Interface Checklist

Design every tool call so the agent can decide, execute, and recover without guessing:

- name the capability, not the implementation detail;
- declare required inputs, optional inputs, limits, and side effects;
- return machine-readable status, result data, and recoverable error codes;
- log the request id, actor, policy decision, and affected resource;
- provide dry-run or preview mode for destructive operations;
- document fallback behavior when the tool is denied, unavailable, or rate-limited.

The interface should make the safe path the shortest path.

Tool Results Are Data

Tool results should not become new instructions. A search result, web page, email, ticket, document, or log line may contain malicious or irrelevant text. The model can inspect it as evidence, but the runtime must not let it override system goals, tool permissions, approval rules, or memory policy.

Good tool results keep their pieces separate: trusted metadata, the untrusted content itself, the source, timestamp, confidence, permissions, and redaction status. That separation makes context construction safer and evaluation easier.

Credentials And Egress Boundaries

An agent should not hold broad credentials. The runtime should exchange the agent's identity, task, tenant, and approved capability for a narrow credential at the moment of tool invocation. That credential should have a short lifetime, a scoped audience, and only the permissions needed for the specific call.

The same idea applies to network egress. A tool that reads an internal order service should not also be able to send arbitrary HTTP requests to the internet. A browser tool should not reach private networks unless the run explicitly needs that access. A code execution tool should not inherit the developer machine's environment by default.

At minimum, high-risk tools need:

- scoped OAuth or OIDC claims checked by the tool server;
- TLS for service-to-service transport;
- per-tool egress allowlists;
- no ambient secrets in prompts, logs, or tool results;
- tenant and actor binding on every request;
- redaction before traces, eval fixtures, and memory writes;

- revocation and kill-switch support.

This is where tool design connects to service architecture. The model selects an action. The runtime authorizes the action. The tool performs the action. Those responsibilities should not collapse into one prompt.

Observability Requirements

Tool calls need traces that explain both what happened and why the runtime allowed it to happen. A useful tool trace includes the run ID, agent ID, actor or service principal, tool name and version, input shape, redacted input summary, policy decision, approval ID when present, idempotency key, timeout, retry count, result status, error class, latency, token context reference, and redaction status.

Do not log raw sensitive payloads just because debugging is easier that way. The production trace should be good enough to reconstruct the decision path without turning observability into a second data leak.

For incident review, the trace should answer:

- Which tool was called?
- Which capability classes were active?
- Which policy allowed or blocked the call?
- Was untrusted content in context before the call?
- Was private data returned?
- Did private data leave through an external channel?
- Was approval required, and was it attached to the exact action?
- Was the call replayed, retried, or deduplicated?

Progressive Tool Disclosure

Do not show every tool to every agent on every step. Expose them progressively:

1. Start with the route or task type.
2. Load the smallest useful tool set.
3. Add tools only after the state justifies them.
4. Remove tools when the phase changes.
5. Require approval or policy checks for cross-capability chains.

A support agent, for example, might start with read-only policy and order-lookup tools. Refund creation appears only after eligibility is established, refund submission only after a draft exists, and external customer messaging only after approval. Disclosing tools this way reduces selection errors and limits the blast radius of prompt injection.

Tool Composition

The danger often lives in the chain, not in any single tool. A read-only private-data tool is fine on its own. An external messaging tool is fine on its own. A browser tool is fine on its own. The chain turns unsafe when one run combines private data, untrusted content, and external communication, so evaluate tool chains, not only individual calls.

For each run, ask what capabilities are present, whether untrusted content entered the context, whether private data can leave through any output channel, whether the write action is justified by trusted evidence, whether the approval record is attached to the exact action, and whether a safer deterministic workflow could do the same thing. This is where tool design connects directly to threat modeling.

Evaluation Guidance

Tool evals should test both selection and restraint. Build cases where the correct behavior is to call the right tool, but also cases where the correct behavior is to call no tool, ask for missing input, refuse an unsupported action, route to approval, use a read-only tool instead of a write tool, avoid external communication, stop after a tool returns untrusted instructions, recover from a malformed response, or retry only when retry is safe.

Then measure tool-selection accuracy, invalid-argument rate, unauthorized-call rate, approval-routing accuracy, unsafe-chain prevention, tool-result grounding, latency and cost per tool path, and recovery from tool errors. A final answer can look correct while the tool trajectory was unsafe, so evaluate the trajectory.

Good tool evals are vertical slices. Do not only test `call_tool(input)`. Test the full path from user request to context construction, tool disclosure, policy decision, tool invocation, result handling, memory behavior, final answer, trace, and replay. That is where the real bugs hide.

Useful eval scenarios include:

- a normal successful path;
- missing required input;
- malformed but recoverable input;
- untrusted content that tries to override instructions;
- private data followed by an external communication request;
- duplicate submit or retry after timeout;
- approval required but missing;
- stale tool result;
- tool returns partial data;
- tool is disabled by policy;
- memory write requested from untrusted evidence.

The evaluation target is not just whether the final answer reads well. It is whether the system used the minimum necessary authority, respected policy, preserved evidence, avoided unsafe chains, and left a useful audit trail.

Design Checklist

Before exposing a tool to an agent, check:

- Is the tool narrow enough?
- Is the input schema strict?
- Is the output structured?
- Are side effects explicit?
- Is the capability class declared?
- Is the risk class declared?
- Are permissions enforced outside the prompt?
- Are credentials short lived and scoped?
- Is egress restricted to what the tool needs?
- Does the tool support idempotency where needed?
- Are retries safe and bounded?
- Are approvals bound to the exact action?
- Are errors actionable?
- Are traces complete and redacted?
- Are private data and untrusted content marked separately?
- Can the tool be mocked in evals?
- Can the tool be replayed without repeating unsafe side effects?
- Can the tool be disabled quickly?
- Does the tool return untrusted content as data, not instructions?

The test is simple: if the model calls this tool incorrectly, can the architecture catch it before harm happens?

Related Chapters

- [Tool Use](#)
- [MCP-first Tool Use](#)
- [Skills](#)
- [Human Approval Gates](#)
- [Agent Threat Model](#)
- [Agent Security and Sandboxing](#)
- [Secure Agent Communication](#)
- [Policy Enforcement](#)
- [Observability and Evals](#)
- [Production Runtime Overview](#)
- [Evaluation-Driven Agent Development](#)

MCP-first Tool Use

MCP-first tool use separates tool capability from agent logic through manifests, validation, invocation, and structured results.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Use this pattern when the set of tools may evolve independently from the agent. MCP gives tools a manifest boundary so the agent can inspect capabilities instead of relying on hard-coded assumptions.

Use When

- Tools are shared across agents or applications.
- Tool schemas, context, and permissions need to be discoverable.
- You want to test tool invocation separately from model reasoning.

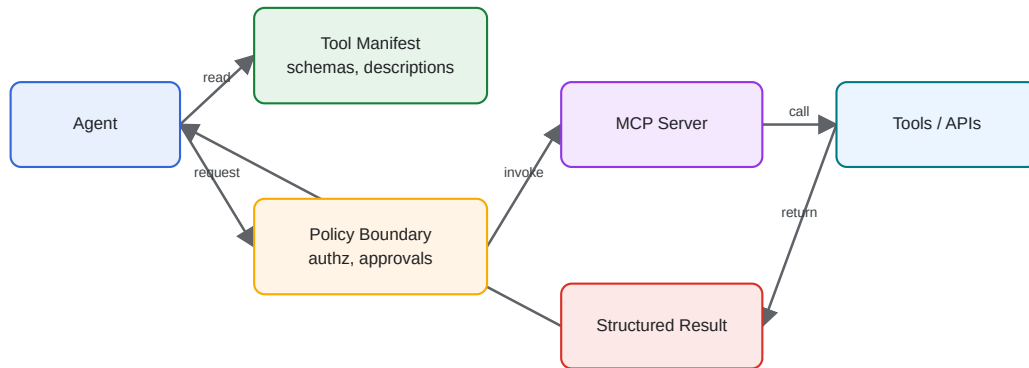
Avoid When

- A local function call is enough and no discovery boundary is needed.
- Tool inputs cannot be validated.
- The agent has permission to call too many tools without policy checks.

Architecture

Use this diagram to read MCP-first Tool Use as a system boundary, not only a code shape. The key ownership question is: the protocol or capability boundary owns schemas, permissions, invocation records, and response validation.

MCP-first Tool Use



System Shape

- **Pattern boundary:** the agent discovers or selects a capability, submits a typed request, and receives a typed result across a policy boundary.
- **State owner:** the protocol or capability boundary owns schemas, permissions, invocation records, and response validation.
- **Primary artifact:** `modern-tool-use-pattern/` contains the runnable reference implementation and examples.
- **Operational promise:** MCP-first tool use separates tool capability from agent logic through manifests, validation, invocation, and structured results.
- **Runnable path:** start with `npm run mcp:search` before adapting the pattern to a larger system.

Core Protocol

1. Discover the capability, schema, permissions, and operating constraints.
2. Prepare a typed request from the current goal and state.
3. Authorize the request before invocation.
4. Invoke the tool, skill, or remote agent and validate the result.
5. Return structured output, refusal, progress, or error without losing correlation IDs.

Implementation Notes

- Validate tool input before every invocation.
- Keep tool results structured and observable.
- Put policy checks between model intent and tool execution.
- Prefer small tools with clear contracts over broad tools with vague descriptions.

Failure Modes

- Tool manifests that describe capabilities too vaguely.
- Model-generated tool inputs used without schema validation.

- Hidden side effects that are not visible in the tool contract.
- No trace linking model decision, tool input, and tool result.

Evaluation Strategy

- Test valid calls, invalid arguments, unauthorized calls, timeouts, refusals, and malformed responses.
- Assert that dangerous actions require approval or are blocked before execution.
- Measure tool-selection accuracy, schema validity, authorization failures, and recovery behavior.
- Include cases that prove each “Use When” condition is true for this pattern.
- Include negative cases from “Avoid When” so the system chooses a simpler or safer pattern when appropriate.

Production Checklist

- Use typed schemas for inputs and outputs.
- Separate model intent from actual execution permissions.
- Add timeouts, retries, idempotency keys, and audit records.
- Treat refusal and cancellation as first-class outcomes.
- Define human escalation for ambiguous, high-risk, or policy-blocked work.
- Keep the source bundle, generated chapter, tests, and deployment artifact in the same release.

Run the Example

```
npm run mcp:search
npm run mcp:cloud
npm run mcp:agent
npm run mcp:test
```

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
modern-tool-use-pattern/typescript/src/agent.ts
```

[Open full source](#)

```

import axios from 'axios';
import Ajv from 'ajv';
import { pathToFileURL } from 'node:url';

const ajv = new Ajv({ allErrors: true, strict: true });

async function getManifest(base: string) {
  const { data } = await axios.get(`${base}/manifest`);
  return data;
}

async function invoke(base: string, tool: string, input: any) {
  const { data } = await axios.post(`${base}/invoke`, { tool, input });
  if (!data?.ok) throw new Error(data?.error || 'invoke_failed');
  return data.output;
}

export async function runScenario() {
  // Discover tools via MCP manifests
  const search = await getManifest('http://localhost:3031');
  const cloud = await getManifest('http://localhost:3032');
  // Validate inputs before invoking
  const searchSchema = search.tools.find((t: any) => t.name ===
'search.query')?.input_schema;
  const cloudStoreSchema = cloud.tools.find((t: any) => t.name ===
'cloud.store')?.input_schema;
  const vSearch = ajv.compile(searchSchema);
  const vCloudStore = ajv.compile(cloudStoreSchema);
  // Plan (deterministic): search for a topic, then store results in cloud
  const q = 'agents';
  const inputSearch = { q, limit: 2 };
  if (!vSearch(inputSearch)) throw new Error('bad search input');
  const { items } = await invoke('http://localhost:3031', 'search.query', inputSearch);
  const doc = { topic: q, titles: items.map((i: any) => i.title) };
  const inputStore = { key: `topic:${q}`, value: doc };
  if (!vCloudStore(inputStore)) throw new Error('bad cloud.store input');
  await invoke('http://localhost:3032', 'cloud.store', inputStore);
  return doc;
}

async function main() {
  const out = await runScenario();
  console.log('Stored doc:', out);
}

```

```
if (process.argv[1] && import.meta.url === pathToFileURL(process.argv[1]).href) {  
  main().catch(err => { console.error(err); process.exit(1); });  
}
```

`modern-tool-use-pattern/typescript/src/mcp_search_server.ts`

[Open full source](#)

```
import express from 'express';  
import type { Request, Response } from 'express-serve-static-core';  
  
const app = express();  
app.use(express.json());  
  
const manifest = {  
  name: 'search-tools',  
  version: '1.0.0',  
  tools: [  
    {  
      name: 'search.query',  
      description: 'Mock web search returning titles for a query',  
      input_schema: {  
        type: 'object',  
        required: ['q'],  
        properties: {  
          q: { type: 'string' },  
          limit: { type: 'number', minimum: 1, maximum: 5 }  
        },  
      },  
      additionalProperties: false  
    }  
  ]  
};  
  
app.get('/manifest', (_req: Request, res: Response) => res.json(manifest));  
  
app.post('/invoke', (req: Request, res: Response) => {  
  const { tool, input } = req.body || {};  
  if (tool === 'search.query') {  
    const q = String(input?.q || '').trim();  
    const limit = Math.max(1, Math.min(5, Number(input?.limit ?? 3)));  
    const items = Array.from({ length: limit }, (_, i) => ({ title: `${q} result ${i + 1}`  
    }));  
  }  
});
```

```

    return res.json({ ok: true, output: { items } });
  }
  return res.status(400).json({ ok: false, error: 'unknown_tool' });
});

app.listen(3031, () => console.log('MCP search server on 3031'));

```

`modern-tool-use-pattern/typescript/src/mcp_cloud_server.ts`

[Open full source](#)

```

import express from 'express';
import type { Request, Response } from 'express-serve-static-core';

const app = express();
app.use(express.json());

const store = new Map<string, any>();

const manifest = {
  name: 'cloud-store',
  version: '1.0.0',
  tools: [
    {
      name: 'cloud.store',
      description: 'Store a JSON document under a key',
      input_schema: {
        type: 'object',
        required: ['key', 'value'],
        properties: { key: { type: 'string' }, value: { type: 'object' } },
        additionalProperties: false
      }
    },
    {
      name: 'cloud.fetch',
      description: 'Fetch a document by key',
      input_schema: {
        type: 'object',
        required: ['key'],
        properties: { key: { type: 'string' } },
        additionalProperties: false
      }
    }
  ]
}

```

```

};

app.get('/manifest', (_req: Request, res: Response) => res.json(manifest));

app.post('/invoke', (req: Request, res: Response) => {
  const { tool, input } = req.body || {};
  if (tool === 'cloud.store') {
    store.set(String(input.key), input.value);
    return res.json({ ok: true });
  }
  if (tool === 'cloud.fetch') {
    return res.json({ ok: true, output: { value: store.get(String(input.key)) ?? null } });
  }
  return res.status(400).json({ ok: false, error: 'unknown_tool' });
});

app.listen(3032, () => console.log('MCP cloud server on 3032'));

```

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `modern-tool-use-pattern/` folder from this repository.

Related Patterns

- [Skills](#)
- [A2A Agent Interoperability](#)
- [Secure Agent Communication](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

A2A Agent Interoperability

A2A makes agents discoverable and callable across process, team, runtime, and vendor boundaries.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

A2A makes agents discoverable and callable across process, team, runtime, and vendor boundaries. Use it when one agent needs to call another agent as a remote collaborator through an explicit protocol contract.

This is the agent version of service-to-service communication. The point is not to let agents chat freely. The point is to let one bounded agent request work from another bounded agent with identity, schema validation, authorization, idempotency, progress, refusal, cancellation, traceability, and versioning.

Microservice communication patterns still apply. REST, gRPC, MCP, and A2A are different protocol shapes, but the engineering concerns are familiar: contracts, identity, authorization, retries, timeouts, idempotency, observability, ownership, and backward compatibility.

Use When

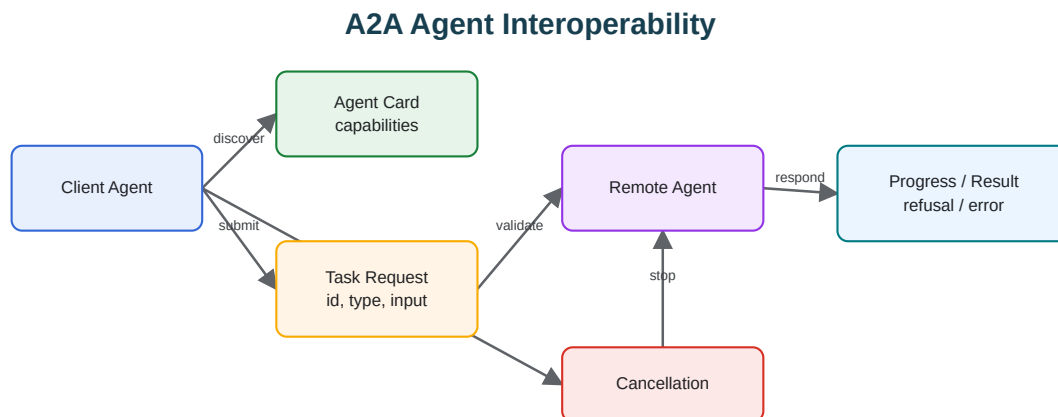
- Agents are owned by different services, teams, runtimes, or vendors.
- A caller must discover what a remote agent can do before sending work.
- Task state must survive asynchronous progress, refusal, error, timeout, retry, or cancellation.
- The caller and callee need a shared contract for ownership, task lifecycle, and result shape.
- Cross-agent communication needs TLS, OAuth or OIDC scopes, trace IDs, and audit records.

Avoid When

- Both agents are simple functions inside one process.
- The interaction is only a local tool call with a typed input and output.
- You cannot authenticate callers, validate messages, or trace handoffs.
- The remote agent has no stable owner, capability contract, or versioning policy.
- A deterministic workflow would be clearer than adding another autonomous collaborator.

Architecture

Use this diagram to read A2A Agent Interoperability as a system boundary, not only a code shape. The key ownership question is: the protocol or capability boundary owns schemas, permissions, invocation records, and response validation.



System Shape

- **Capability boundary:** callers discover capabilities from an agent card instead of relying on stale assumptions.
- **Message boundary:** every task request has a schema, trace ID, message ID, idempotency key, source agent, target agent, tenant, audience, scopes, and timeout.
- **Identity boundary:** callers are authenticated through service identity and delegated credentials.
- **Authorization boundary:** scopes, tenant, audience, capability, policy, and risk are checked before the remote agent starts work.
- **Ownership boundary:** every task has a current owner, status, stop reason, and cancellation path.
- **Observability boundary:** handoffs, retries, refusals, timeouts, approvals, and results are trace events, not hidden chat messages.

Core Protocol

1. Fetch the remote agent card: capability, schema, version, scopes, owner, timeout, and lifecycle states.
2. Build a task request envelope from current goal, state, tenant, trace ID, idempotency key, and allowed delegation budget.
3. Validate the message schema before transport.
4. Authenticate caller and verify audience, scopes, tenant, capability, and policy.
5. Deliver the request to the remote agent only after authorization succeeds.
6. Emit progress, refusal, error, timeout, cancellation, or result events with the same trace ID.
7. Validate the response schema and ownership state before the caller consumes the result.
8. Record the handoff, decision, latency, cost, status, owner, and stop reason.
9. Convert repeated failures, unsafe delegation, or protocol mismatches into eval cases.

Implementation Notes

- Messages should be validated against schemas before delivery.
- A2A messages should carry correlation fields, not rely on logs to reconstruct a handoff later.
- Use TLS for remote transport and mTLS where service identity matters.
- Use OAuth or OIDC scopes for delegated authority.
- Treat refusals, timeouts, and cancellation as normal protocol outcomes, not exceptions.
- Include idempotency keys or task IDs so retries do not duplicate work.
- Add authorization before crossing a trust boundary.
- Keep agent card versions and message schema versions backward compatible.
- Avoid delegation loops by tracking owner, parent task, delegation depth, and stop reason.

Message Envelope

```
type AgentMessageEnvelope = {  
  traceId: string;  
  messageId: string;  
  idempotencyKey: string;  
  fromAgent: string;  
  toAgent: string;  
  tenantId: string;  
  capability: string;  
  auth: {  
    audience: string;  
    scopes: string[];  
  };  
  timeoutMs: number;  
  payload: Record<string, unknown>;  
};
```

Authorization Check

```
function validateA2AEnvelope(input: {  
  envelope: AgentMessageEnvelope;  
  requiredAudience: string;  
  requiredScope: string;  
  targetAgent: string;  
  seenIdempotencyKeys: Set<string>;  
}) {  
  const { envelope, requiredAudience, requiredScope, targetAgent, seenIdempotencyKeys } =  
    input;  
  
  if (!envelope.traceId || !envelope.messageId || !envelope.idempotencyKey) {  
    return { decision: 'deny', reason: 'missing_correlation_fields' };  
  }
```

```

    }

    if (envelope.toAgent !== targetAgent || envelope.auth.audience !== targetAgent) {
        return { decision: 'deny', reason: 'wrong_audience' };
    }

    if (envelope.auth.audience !== requiredAudience) {
        return { decision: 'deny', reason: 'wrong_required_audience' };
    }

    if (!envelope.auth.scopes.includes(requiredScope)) {
        return { decision: 'deny', reason: 'missing_scope' };
    }

    if (seenIdempotencyKeys.has(envelope.idempotencyKey)) {
        return { decision: 'deny', reason: 'duplicate_message' };
    }

    return { decision: 'allow', reason: 'accepted' };
}

```

The exact envelope can change by protocol, but the boundary should remain: identify the caller, identify the target, validate authority, preserve correlation, and stop duplicate work.

Failure Modes

- Treating a remote agent like a local function and ignoring latency, refusal, timeout, or cancellation.
- Sending unvalidated natural language blobs instead of typed task messages.
- Missing capability discovery, causing callers to rely on stale assumptions.
- No trace ID across progress, result, and error messages.
- No idempotency key, so retries duplicate work.
- Wrong audience or missing scope is accepted because the remote agent trusts the caller by name.
- Agents delegate the same task back and forth with no owner or delegation budget.
- The remote agent changes its schema without versioning.
- A fallback path bypasses authorization or observability.

Evaluation Strategy

Test the protocol, not only the happy-path collaboration.

- Test valid calls with correct audience, scope, tenant, schema, trace ID, and idempotency key.
- Test wrong audience, missing scope, schema mismatch, missing trace ID, and duplicate message.
- Test refusal as a valid result.
- Test timeout and cancellation behavior.

- Test delegation loop detection.
- Test unsafe capability requests.
- Test backward-compatible schema evolution.
- Test that every handoff emits trace and ownership events.

Measure schema-validity rate, authorization false allows, refusal handling, timeout recovery, duplicate-message detection, delegation-loop rate, trace completeness, and owner-at-failure coverage.

Production Checklist

- Publish an agent card with capabilities, schemas, scopes, owner, version, and lifecycle states.
- Validate every request and response against versioned schemas.
- Require TLS for remote transport and mTLS for service identity where appropriate.
- Validate OAuth or OIDC audience, scopes, subject, tenant, and expiry before execution.
- Require trace ID, message ID, task ID, idempotency key, source agent, and target agent.
- Treat refusal, cancellation, timeout, and approval wait as first-class outcomes.
- Track current owner and delegation depth to prevent task bouncing.
- Add timeouts, retries, and cancellation semantics before production.
- Record handoffs, authorization decisions, progress, results, and stop reasons in traces.
- Convert protocol failures and unsafe delegation into eval fixtures.

Run the Example

```
npm run a2a:test
npm run a2a:run
```

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

[agent-to-agent-communication-pattern/src/run_demo.ts](#)

[Open full source](#)

```
import { BusMemory } from './bus_memory.ts';
import { AgentA } from './agent_a.ts';
import { AgentB } from './agent_b.ts';
```

```

async function run() {
  const bus = new BusMemory();
  const a = new AgentA(bus);
  const b = new AgentB(bus);
  a.start();
  b.start();
  a.handshake();
  a.requestTask('t1', 'sum', { a: 2, b: 5 });
}

run();

```

agent-to-agent-communication-pattern/src/agent_a.ts

[Open full source](#)

```

import { BusMemory, A2A_SCHEMA } from './bus_memory.ts';
import type { Msg } from './bus_memory.ts';
import Ajv from 'ajv';
import crypto from 'node:crypto';
const ajv = new Ajv({ allErrors: true, strict: true });
const validateResponse = ajv.compile((A2A_SCHEMA as any).properties.TaskResponse);

export class AgentA {
  private bus: BusMemory;
  private traceId = crypto.randomUUID();
  constructor(bus: BusMemory) { this.bus = bus; }
  start() {
    // listen for responses
    this.bus.subscribe('TaskResponse', (m: Msg) => {
      if (!validateResponse(m.payload)) {
        console.error('Invalid TaskResponse', validateResponse.errors);
        return;
      }
      console.log('AgentA received response:', m.payload);
    });
  }
  handshake() {
    this.bus.publish({ type: 'Handshake', payload: { version: '1.0', capabilities:
['tasks', 'cancel'] } });
  }
  requestTask(id: string, task_type: string, input: any) {
    this.bus.publish({

```

```

        type: 'TaskRequest',
        payload: {
            id,
            task_type,
            input,
            meta: {
                trace_id: this.traceId,
                message_id: crypto.randomUUID(),
                idempotency_key: `task:${id}`,
                from_agent: 'agent-a',
                to_agent: 'agent-b',
                tenant_id: 'tenant_a',
                auth: {
                    audience: 'agent-b',
                    scopes: ['task:sum']
                },
                timeout_ms: 30000,
                ts: Date.now()
            }
        }
    });
}

cancel(id: string, reason: string) {
    this.bus.publish({ type: 'Cancel', payload: { id, reason } });
}
}

```

agent-to-agent-communication-pattern/src/agent_b.ts

[Open full source](#)

```

import { BusMemory, A2A_SCHEMA } from './bus_memory.ts';
import type { Msg } from './bus_memory.ts';
import Ajv from 'ajv';
const ajv = new Ajv({ allErrors: true, strict: true });
const validateReq = ajv.compile((A2A_SCHEMA as any).properties.TaskRequest);

export class AgentB {
    private bus: BusMemory;
    private handshakeAckSent = false;
    private seenIdempotencyKeys = new Set<string>();
    constructor(bus: BusMemory) { this.bus = bus; }
    start() {
        this.bus.subscribe('Handshake', () => {

```

```

    if (this.handshakeAckSent) return;
    this.handshakeAckSent = true;
    this.bus.publish({ type: 'Handshake', payload: { version: '1.0', capabilities:
['tasks'] } });
  });
  this.bus.subscribe('TaskRequest', (m: Msg) => {
    if (!validateReq(m.payload)) return;
    const payload = m.payload as any;
    const { id, task_type, input } = payload;
    const authorization = this.authorize(payload);
    if (!authorization.allowed) {
      this.bus.publish({
        type: 'TaskResponse',
        payload: { id, status: 'refused', error: authorization.reason }
      });
      return;
    }
    this.seenIdempotencyKeys.add(payload.meta.idempotency_key);
    if (task_type !== 'sum') {
      this.bus.publish({ type: 'TaskResponse', payload: { id, status: 'refused', error:
'unsupported_task' } });
      return;
    }
    // progress
    this.bus.publish({ type: 'Progress', payload: { id, stage: 'start', pct: 10, message:
'starting' } });
    const a = input?.a;
    const b = input?.b;
    if (typeof a !== 'number' || typeof b !== 'number') {
      this.bus.publish({ type: 'TaskResponse', payload: { id, status: 'error', error:
'invalid_input' } });
      return;
    }
    // compute safely
    const sum = a + b;
    this.bus.publish({ type: 'Progress', payload: { id, stage: 'compute', pct: 60 } });
    this.bus.publish({ type: 'TaskResponse', payload: { id, status: 'success', output: {
sum } } });
  });
  this.bus.subscribe('Cancel', (m: Msg) => {
    console.log('AgentB cancel received:', m.payload);
  });
}

```



```

private authorize(payload: any) {
  const meta = payload.meta;
  if (meta.to_agent !== 'agent-b' || meta.auth.audience !== 'agent-b') {
    return { allowed: false, reason: 'wrong_audience' };
  }
  if (!meta.auth.scopes.includes('task:sum')) {
    return { allowed: false, reason: 'missing_scope' };
  }
  if (meta.tenant_id !== 'tenant_a') {
    return { allowed: false, reason: 'tenant_boundary' };
  }
  if (this.seenIdempotencyKeys.has(meta.idempotency_key)) {
    return { allowed: false, reason: 'duplicate_message' };
  }
  return { allowed: true, reason: 'authorized' };
}
}

```

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `agent-to-agent-communication-pattern/` folder from this repository.

Related Patterns

- [Agents As Services](#)
- [Choosing Multi-Agent Topology](#)
- [Supervisor / Worker](#)
- [Secure Agent Communication](#)
- [Policy Enforcement](#)
- [Observability and Evals](#)
- [MCP-first Tool Use](#)

Secure Agent Communication

Secure communication protects messages between agents with authentication, integrity, confidentiality, and policy checks.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Secure agent communication protects messages between agents, tools, runtimes, and services with identity, authorization, confidentiality, integrity, replay protection, and observability.

This pattern matters because agent-to-agent communication is not just chat between models. Messages can carry private data, tool instructions, delegated goals, approval requests, workflow commands, and side-effect proposals. If the communication layer is weak, the model can be correct and the system can still be unsafe.

The practical rule is simple: every cross-boundary agent message needs a transport boundary, an identity boundary, an authorization boundary, and an audit boundary.

Use When

- Agents, tools, skills, or runtimes communicate across processes, services, teams, tenants, or networks.
- Messages contain private data, business state, credentials, delegated tasks, or side-effect instructions.
- Remote agents can invoke tools, workflows, APIs, browsers, shells, or data stores.
- The system needs TLS, mTLS, OAuth or OIDC, scoped credentials, replay protection, and trace correlation.
- Operators must explain who requested what, which policy allowed it, what executed, and which trace proves it.

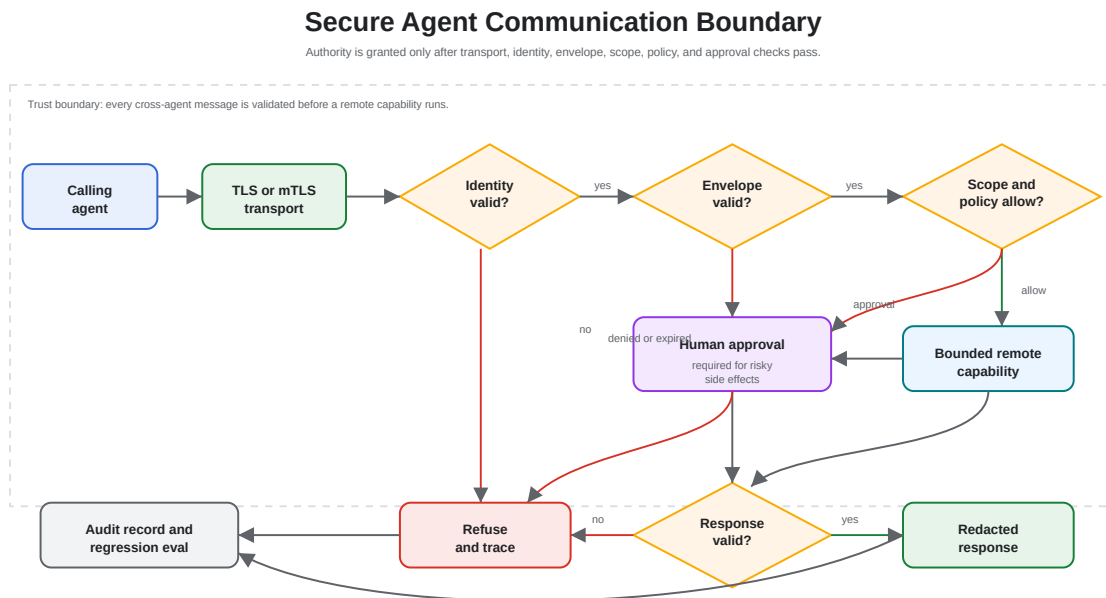
Avoid When

- All communication is local, in-process, and already covered by the same trusted execution boundary.
- The system cannot identify caller, tenant, audience, capability, and trace ID.
- Authorization happens only in the prompt or only after the remote action executes.
- Messages cannot be redacted, retained, traced, or replayed safely.

- Token and certificate lifecycle cannot be operated by the team.

Architecture

Use this diagram to read Secure Agent Communication as a system boundary, not only a code shape. The key ownership question is: the protocol or capability boundary owns schemas, permissions, invocation records, and response validation.



Read it as a sequence of authority gates. The remote agent or tool receives capability only after transport, identity, envelope, scope, policy, and approval checks pass.

System Shape

- **Transport boundary:** use TLS for service communication and mTLS where both sides need certificate-backed service identity.
- **Identity boundary:** identify caller, subject, tenant, audience, issuer, and service identity before processing the request.
- **Authorization boundary:** check OAuth or OIDC token claims, scopes, audience, tenant, tool capability, risk class, and approval requirements.
- **Message boundary:** validate schemas, idempotency keys, timestamps, nonces, and content size before the message enters the agent loop.
- **Data boundary:** encrypt sensitive payloads when needed, redact traces, and avoid placing secrets in prompts, memory, logs, or eval fixtures.
- **Observability boundary:** emit trace events for identity verification, authorization decision, policy result, tool invocation, refusal, approval, and response validation.

Core Protocol

1. Receive a message over TLS or mTLS.
2. Verify service identity and token issuer, audience, expiry, subject, tenant, and scopes.
3. Validate message schema, idempotency key, timestamp, nonce, trace ID, and correlation IDs.
4. Build policy context from trusted runtime data, not only message text.
5. Check tool capability, data scope, risk class, budget state, and approval requirement.
6. Decrypt or unwrap sensitive payloads only after authorization succeeds.
7. Execute the bounded remote action, or deny, escalate, or wait for approval.
8. Validate the response schema and redact sensitive fields before returning.
9. Record trace events, policy decision, token subject, service identity, scopes, latency, cost, and stop reason.

Implementation Notes

TLS protects the transport. OAuth and OIDC identify and authorize the caller. Policy decides whether the requested capability is allowed. Observability proves what happened.

Do not treat these as interchangeable:

Control	What It Answers
TLS	Is the connection encrypted and protected in transit?
mTLS	Which service identity is on the other side of this connection?
OAuth or OIDC	Who or what is the caller, and what scopes or claims does it have?
Policy engine	Is this caller allowed to perform this action on this resource now?
Message signing	Has this message been tampered with or replayed outside the transport session?
Trace correlation	Can operators reconstruct the request path across agents and tools?

Use scoped, short-lived credentials. A remote agent should receive the minimum scope required for the delegated task, not the user's full authority and not a platform-wide service token.

Observability

Secure communication without observability is difficult to operate. Record enough to audit behavior without leaking secrets.

Trace these events:

- inbound message accepted or rejected;
- TLS or mTLS service identity;
- token issuer, audience, subject, tenant, and scopes after redaction;
- policy decision and reason;
- idempotency key and replay detection result;
- approval request, approval grant, or approval denial;

- remote agent or tool invoked;
- response validation result;
- latency, cost, timeout, retry, and stop reason.

Do not log raw access tokens, refresh tokens, private keys, decrypted secrets, or unredacted private data. Observability should explain authority and behavior, not become a second data leak.

Secure Message Envelope

```
type SecureAgentEnvelope = {
  traceId: string;
  messageId: string;
  idempotencyKey: string;
  issuedAt: string;
  expiresAt: string;
  caller: {
    subject: string;
    tenantId: string;
    serviceId: string;
  };
  auth: {
    issuer: string;
    audience: string;
    scopes: string[];
  };
  capability: 'read' | 'delegate' | 'tool_call' | 'workflow_command';
  payloadRef?: string;
  encryptedPayload?: string;
};
```

Authorization Check

```
function authorizeEnvelope(input: {
  envelope: SecureAgentEnvelope;
  requiredAudience: string;
  requiredScope: string;
  now: Date;
}) {
  const { envelope, requiredAudience, requiredScope, now } = input;

  if (envelope.auth.audience !== requiredAudience) {
    return { decision: 'deny', reason: 'wrong_audience' };
  }
}
```

```

if (!envelope.auth.scopes.includes(requiredScope)) {
  return { decision: 'deny', reason: 'missing_scope' };
}

if (new Date(envelope.expiresAt).getTime() <= now.getTime()) {
  return { decision: 'deny', reason: 'expired_message' };
}

if (!envelope.traceId || !envelope.idempotencyKey) {
  return { decision: 'deny', reason: 'missing_trace_or_idempotency' };
}

return { decision: 'allow', reason: 'authorized' };
}

```

The real implementation should validate signed tokens and certificates with trusted libraries and platform identity providers. The important architectural point is where the check lives: before the remote agent or tool receives authority.

Failure Modes

- TLS is present, but any authenticated service can call every agent capability.
- OAuth scopes are broad, long-lived, or not checked against the specific capability.
- The token audience is ignored, so a token minted for one service works against another.
- Tenant, subject, or service identity is taken from model text instead of trusted runtime claims.
- Messages have no idempotency key, nonce, timestamp, or replay detection.
- Remote agents receive ambient credentials instead of scoped delegated authority.
- Sensitive payloads, tokens, or decrypted data appear in prompts, memory, logs, or traces.
- Denied calls are not traced, so attempted abuse is invisible.
- Observability records final outputs but not identity, scope, policy, or transport decisions.

Evaluation Strategy

Test communication security as behavior, not as configuration.

- Test valid messages with correct issuer, audience, scopes, tenant, and schema.
- Test wrong audience, missing scope, expired token, wrong tenant, missing trace ID, and repeated idempotency key.
- Test attempts to call tools outside the delegated capability.
- Test replay of an old message.
- Test redaction of tokens and private data from traces.
- Test approval-required capabilities before side effects.
- Test that denied calls are visible in audit and observability.
- Test that fallback paths do not bypass identity or authorization.

Measure authorization false allows, false denials, replay detection, missing trace fields, policy-decision coverage, redaction failures, and mean time to investigate a cross-agent incident.

Production Checklist

- Enforce TLS for all remote agent communication.
- Use mTLS for service-to-service identity where the infrastructure supports it.
- Validate OAuth or OIDC issuer, audience, expiry, subject, tenant, and scopes.
- Keep tokens scoped, short-lived, rotated, and never visible to the model unless absolutely required.
- Validate message schema, idempotency key, timestamp, nonce, and trace ID.
- Authorize before decrypting sensitive payloads or invoking remote capabilities.
- Apply policy checks to delegated tool calls, workflow commands, and final responses.
- Redact tokens, secrets, private data, and sensitive payloads from traces.
- Emit observability events for identity, authorization, policy, approval, invocation, and response validation.
- Convert security incidents and denied dangerous calls into regression evals.

Run the Example

```
npm run secure-agent
```

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
secure-agent-communication-pattern/autogen_typescript_example/secure_agent.ts
```

[Open full source](#)

```
import crypto from 'crypto';

type SecureAgentEnvelope = {
  traceId: string;
  messageId: string;
  idempotencyKey: string;
  issuedAt: string;
  expiresAt: string;
```

```

    caller: {
      subject: string;
      tenantId: string;
      serviceId: string;
    };
    auth: {
      issuer: string;
      audience: string;
      scopes: string[];
    };
    capability: 'read' | 'delegate' | 'tool_call' | 'workflow_command';
    encryptedPayload: string;
  };

  type TraceEvent = {
    traceId: string;
    event: string;
    decision?: 'allow' | 'deny';
    reason?: string;
    subject?: string;
    scopes?: string[];
  };

  const key = crypto.randomBytes(32);

  function encryptPayload(payload: object): string {
    const iv = crypto.randomBytes(12);
    const cipher = crypto.createCipheriv('aes-256-gcm', key, iv);
    const ciphertext = Buffer.concat([
      cipher.update(JSON.stringify(payload), 'utf8'),
      cipher.final()
    ]);
    const tag = cipher.getAuthTag();

    return Buffer.concat([iv, tag, ciphertext]).toString('base64url');
  }

  function decryptPayload(token: string): unknown {
    const data = Buffer.from(token, 'base64url');
    const iv = data.subarray(0, 12);
    const tag = data.subarray(12, 28);
    const ciphertext = data.subarray(28);
    const decipher = crypto.createDecipheriv('aes-256-gcm', key, iv);
    decipher.setAuthTag(tag);

```



```

const plaintext = Buffer.concat([
  decipher.update(ciphertext),
  decipher.final()
]).toString('utf8');

return JSON.parse(plaintext);
}

function authorizeEnvelope(input: {
  envelope: SecureAgentEnvelope;
  requiredAudience: string;
  requiredScope: string;
  now: Date;
}) {
  const { envelope, requiredAudience, requiredScope, now } = input;

  if (envelope.auth.audience !== requiredAudience) {
    return { decision: 'deny' as const, reason: 'wrong_audience' };
  }

  if (!envelope.auth.scopes.includes(requiredScope)) {
    return { decision: 'deny' as const, reason: 'missing_scope' };
  }

  if (new Date(envelope.expiresAt).getTime() <= now.getTime()) {
    return { decision: 'deny' as const, reason: 'expired_message' };
  }

  return { decision: 'allow' as const, reason: 'authorized' };
}

function trace(event: TraceEvent) {
  console.log(JSON.stringify(event));
}

function main() {
  const traceId = crypto.randomUUID();

```

Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.

[secure-agent-communication-pattern/langgraph_python_example/secure_agent.py](#)

[Open full source](#)

```
import json
import time
import uuid
from cryptography.fernet import Fernet

key = Fernet.generate_key()
fernet = Fernet(key)

def encrypt_payload(payload):
    return fernet.encrypt(json.dumps(payload).encode()).decode()

def decrypt_payload(token):
    return json.loads(fernet.decrypt(token.encode()).decode())

def authorize_envelope(envelope, required_audience, required_scope, now):
    if envelope['auth']['audience'] != required_audience:
        return {'decision': 'deny', 'reason': 'wrong_audience'}

    if required_scope not in envelope['auth']['scopes']:
        return {'decision': 'deny', 'reason': 'missing_scope'}

    if envelope['expires_at'] <= now:
        return {'decision': 'deny', 'reason': 'expired_message'}

    return {'decision': 'allow', 'reason': 'authorized'}

def trace(event):
    print(json.dumps(event))

def main():
    trace_id = str(uuid.uuid4())
    envelope = {
        'trace_id': trace_id,
        'message_id': str(uuid.uuid4()),
        'idempotency_key': f'agent-message:{uuid.uuid4()}',
        'issued_at': int(time.time()),
        'expires_at': int(time.time()) + 60,
        'caller': {
            'subject': 'agent:customer-support',
            'tenant_id': 'tenant_a',
            'service_id': 'support-agent-runtime',
        },
    },
```

```

    'auth': {
        'issuer': 'https://identity.example.test',
        'audience': 'billing-agent',
        'scopes': ['agent:delegate', 'refund:draft'],
    },
    'capability': 'tool_call',
    'encrypted_payload': encrypt_payload({
        'tool': 'refunds.draft_refund',
        'order_id': 'ord_redacted',
        'amount_cents': 2500,
    }),
}

trace({
    'trace_id': trace_id,
    'event': 'secure_message_received',
    'subject': envelope['caller']['subject'],
    'scopes': envelope['auth']['scopes'],
})

authorization = authorize_envelope(
    envelope,
    required_audience='billing-agent',
    required_scope='refund:draft',
    now=int(time.time()),
)

trace({
    'trace_id': trace_id,
    'event': 'authorization_decision',
    'decision': authorization['decision'],
    'reason': authorization['reason'],
})

if authorization['decision'] == 'deny':
    return

payload = decrypt_payload(envelope['encrypted_payload'])
trace({'trace_id': trace_id, 'event': 'payload_decrypted_after_authorization'})
print(json.dumps(payload))

```

Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `secure-agent-communication-pattern/` folder from this repository.

Related Patterns

- [Production Runtime Overview](#)
- [Policy Enforcement](#)
- [Observability and Evals](#)
- [A2A Agent Interoperability](#)
- [MCP-first Tool Use](#)
- [Tool Capability Design](#)
- [Agent Threat Model](#)
- [Agent Security and Sandboxing](#)

Human Approval Gates

Human approval gates pause execution before sensitive, expensive, destructive, or externally visible actions.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Human approval gates pause an agentic workflow before a sensitive, expensive, destructive, or externally visible action is executed. The gate is not “ask a human in chat.” It is a controlled workflow pause with the proposed action, risk, evidence, policy, approver identity, timeout, decision, and audit record.

Approval is an architecture boundary. The model can propose an action. Software decides whether that action needs approval, packages the approval request, waits durably, records the decision, and resumes or stops the workflow.

The important detail is that approval does not make an agent safe by itself. A vague approval step can become theater. A useful approval gate binds a human decision to one exact action, under one policy version, with enough evidence for the human to make a real decision.

Use When

- The agent may trigger financial, legal, security, data-access, or customer-visible side effects.
- A human must review evidence before the workflow continues.
- The workflow can pause, persist state, and resume safely.
- Approval decisions need an audit trail.
- Policy can define which actions require approval and which can proceed automatically.

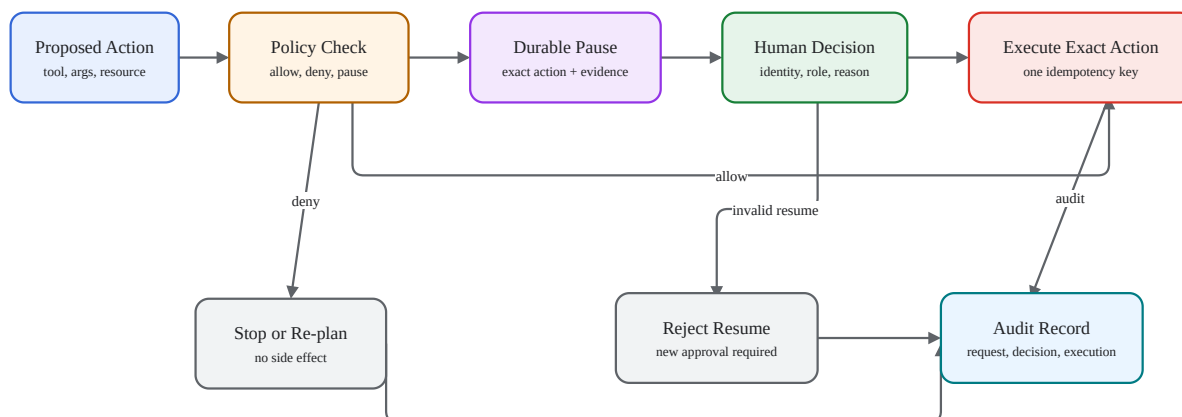
Avoid When

- Every step requires approval and the agent no longer reduces work.
- The approver cannot see enough evidence to make a decision.
- Approval happens in an untracked chat message or side channel.
- The workflow cannot safely resume after waiting.
- The system cannot prevent retries from reusing stale approvals.

Architecture

Use this diagram to read Human Approval Gates as a system boundary, not only a code shape. The key ownership question is: the protocol or capability boundary owns schemas, permissions, invocation records, and response validation.

Human Approval Gate



System Shape

- **Pattern boundary:** the approval gate owns the pause, request, decision, timeout, resume token, and audit record.
- **State owner:** the workflow engine or runtime owns durable state while waiting for approval.
- **Model role:** the model explains the proposed action and supporting evidence, but it does not approve its own action.
- **Policy boundary:** software decides whether approval is required before the action runs.
- **Operational promise:** high-risk actions do not execute only because the model asked for them.

Core Protocol

1. Receive a model-proposed action with caller, trace ID, risk, evidence, and side effects.
2. Run policy to decide whether the action is allowed, denied, or approval-required.
3. If approval is required, create a durable approval request.
4. Present the approver with the proposed action, evidence, policy reason, risk, and consequences.
5. Wait for approve, deny, request-changes, escalation, timeout, or cancellation.
6. Record who decided, when, why, and under which policy version.
7. Resume only the exact approved action, or stop/revise when denied.
8. Store the decision with the trace and action audit log.

The policy decision should happen before the side effect. If the tool has already sent the email, issued the refund, changed the permission, or wrote durable memory, the approval gate is only an incident note.

Implementation Notes

Approval requests should be typed. The approver should not have to infer the side effect from vague prose.

Approval Is Not One Thing

Different risks need different approval shapes. Treating every human checkpoint as the same button creates friction in low-risk flows and weak control in high-risk flows.

Approval Type	Purpose	Example
Confirmation	The user confirms intent before a visible action.	“Send this drafted email to these recipients.”
Review	A qualified person checks evidence and rationale.	Support lead reviews a refund recommendation.
Authorization	A role with explicit authority permits an action.	Finance approves a refund above a threshold.
Escalation	The workflow cannot decide at the current level.	Security reviews a suspicious access request.
Break-glass	A rare emergency override with stronger audit.	Restore access during an outage.
Batch approval	One decision covers a bounded set of similar actions.	Approve 25 low-risk ticket replies generated from the same policy.

Batch approval is the one that needs the most care. It should define the batch membership, maximum count, allowed action type, risk class, expiration, and sampling or review rules. It should not become “approve whatever the agent does next.”

Approval Envelope

An approval request should contain the action envelope, not just a human-readable message. The human-readable part can explain the decision. The machine-readable part is what prevents approval laundering.

Field	Why It Matters
Approval ID	Stable identifier for the pause and decision.
Action ID	The exact action being approved.
Tool name and version	Prevents approval from drifting across tool changes.
Arguments hash	Detects hidden changes after approval.
Resource IDs	Shows what customer, account, file, ticket, payment, or permission is affected.

Field	Why It Matters
Risk class	Connects approval to policy and routing.
Evidence references	Lets the approver inspect source material.
Policy version	Explains why approval was required.
Approver role	Prevents the wrong human from approving.
Expiration	Prevents old approvals from being reused.
Idempotency key	Prevents duplicate execution after retry.
Trace ID	Connects approval to the run and audit trail.

The action ID should be derived from stable fields: tool, version, arguments, resource, tenant, actor, policy version, and idempotency key. If any of those change, the approval should no longer match.

```
type ApprovalRequest = {
  approvalId: string;
  actionId: string;
  traceId: string;
  requestedBy: 'agent' | 'workflow' | 'operator';
  tenantId: string;
  actorId: string;
  proposedAction: {
    tool: string;
    toolVersion: string;
    args: Record<string, unknown>;
    argsHash: string;
    resourceIds: string[];
    sideEffects: string[];
  };
  riskLevel: 'low' | 'medium' | 'high' | 'critical';
  evidenceRefs: string[];
  policyRefs: string[];
  policyVersion: string;
  approverRole: 'support_lead' | 'security_reviewer' | 'finance_approver';
  expiresAt: string;
  idempotencyKey: string;
};
```

The decision record is just as important as the request:

```
type ApprovalDecision = {
  approvalId: string;
```



```

decision: 'approved' | 'denied' | 'changes_requested' | 'expired';
decidedBy: string;
decidedByRole: string;
decidedAt: string;
reason: string;
approvedActionId?: string;
approvedArgsHash?: string;
policyVersion: string;
traceId: string;
};

```

Never treat approval as blanket permission. Bind it to the exact action:

```

type ActionToExecute = {
  actionId: string;
  tool: string;
  toolVersion: string;
  argsHash: string;
  idempotencyKey: string;
};

function canResumeWithApproval(
  request: ApprovalRequest,
  decision: ApprovalDecision,
  action: ActionToExecute,
) {
  if (decision.decision !== 'approved') return false;
  if (decision.approvalId !== request.approvalId) return false;
  if (decision.approvedActionId !== action.actionId) return false;
  if (decision.approvedArgsHash !== action.argsHash) return false;
  if (request.proposedAction.tool !== action.tool) return false;
  if (request.proposedAction.toolVersion !== action.toolVersion) return false;
  if (request.idempotencyKey !== action.idempotencyKey) return false;
  if (decision.policyVersion !== request.policyVersion) return false;
  if (new Date(request.expiresAt).getTime() < Date.now()) return false;
  return true;
}

```

The approval approves one action, under one policy version, with one trace. If the agent changes the action, the workflow needs a new approval.

What The Approver Must See

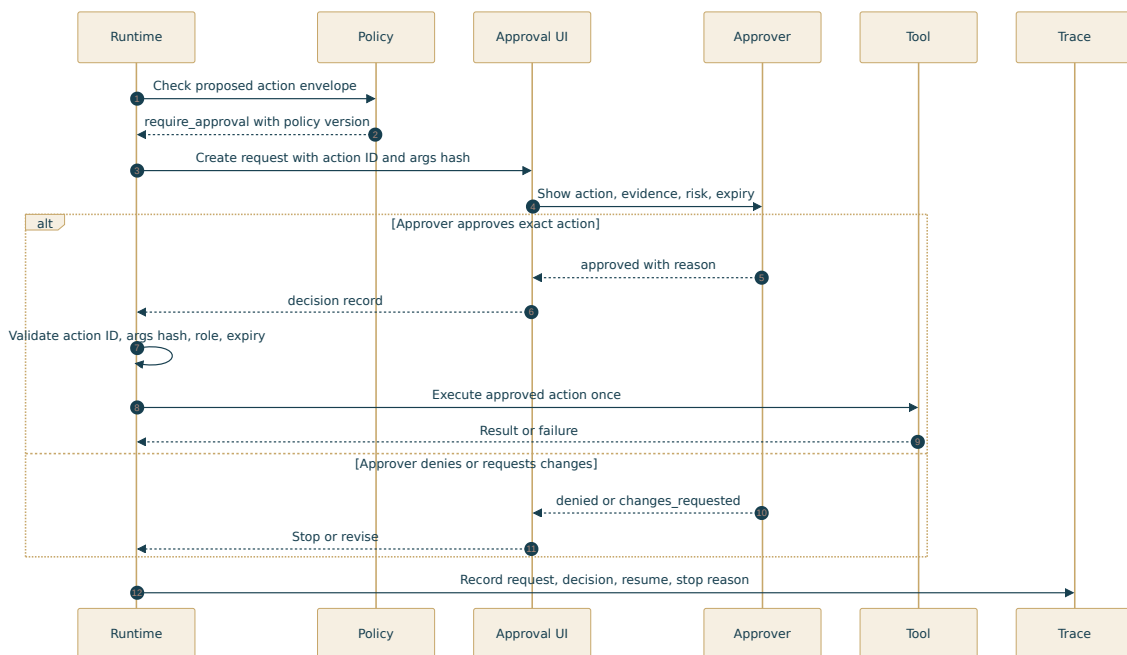
A good approval UI is not a chat transcript with an approve button. It should show the exact action, the resource being changed, the risk class, the policy reason, the evidence, the diff or payload, the blast radius, the expiration, and the available decisions.

For a customer refund, the approver should see the order, payment, customer message, refund policy, amount, reason, and whether the agent is drafting, submitting for approval, or issuing money. For an outbound email, the approver should see recipients, subject, body, attachments, source evidence, and any private data included. For a memory write, the approver should see the proposed memory, source, tenant, retention class, and deletion path.

The human should be able to approve, deny, request changes, escalate, or cancel. Free-form comments are useful, but the decision itself should be structured.

Approval Review Flow

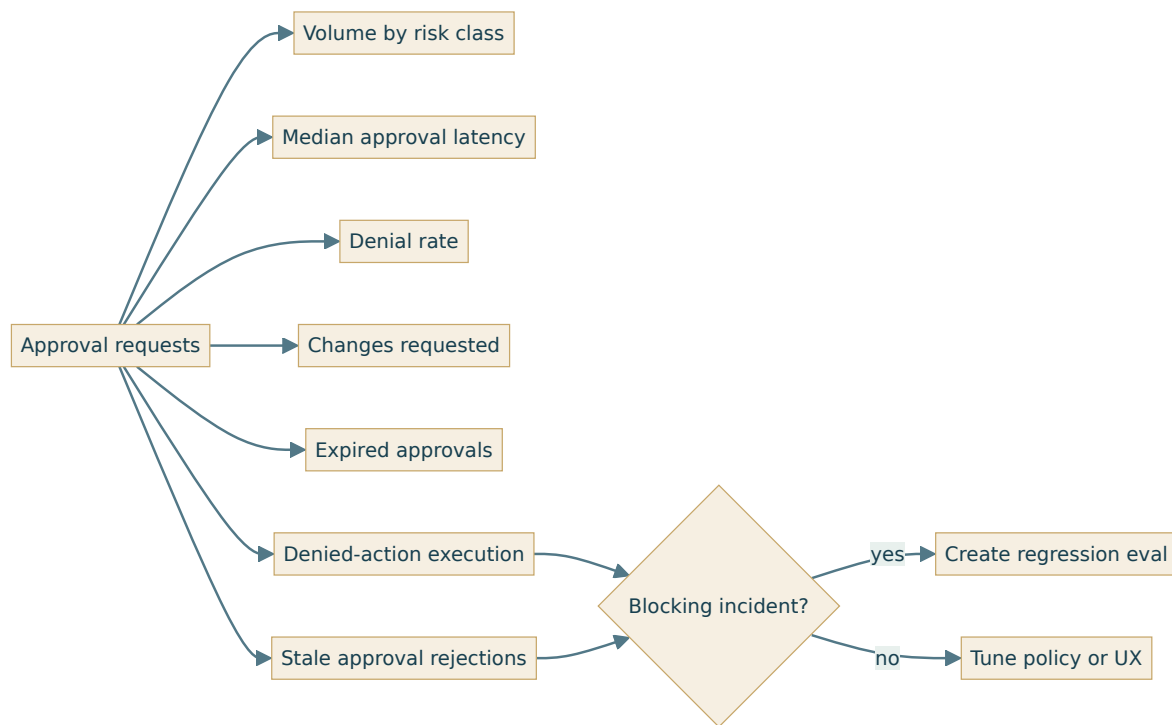
Use this flow to test whether the UI, policy engine, and runtime agree on the same action. The approver should never approve a summary that the runtime later translates into a different payload.



The UI can be simple, but it must preserve the machine contract. The visible action, policy reason, arguments hash, resource IDs, expiry, and decision should match the records used by `canResumeWithApproval`.

Approval Metrics Panel

Approval gates need operating signals, not just a request screen. A small dashboard should separate convenience problems from safety problems.



Treat denied-action execution, stale-approval reuse, and missing approval spans as release-blocking signals. High approval latency may be a workflow problem. High denial rate may mean the agent is requesting actions outside its authority. High rubber-stamp rate may mean the UI is asking for too many low-value decisions.

Stale Approval And Replay Protection

Approval gates fail when a decision can be replayed outside its original context. The runtime should reject an approval when the action changed, the policy version changed, the approval expired, the approver role no longer matches, the tenant changed, the resource changed, or the idempotency key was already consumed.

This matters for retries. Durable workflows should resume from the approval wait, not rebuild a similar looking action from fresh model output and assume the old approval still applies. If the model re-plans after approval, the new plan must pass policy again.

Denial Tracking

Repeated denials are signal, not noise. Track denied tool requests by tool, scope, and reason. After a small threshold, the agent should stop retrying and choose one of three paths:

1. ask for a broader approval with a clear reason;
2. switch to a lower-risk fallback;
3. stop and report the blocked requirement.

This prevents approval loops where the agent burns its budget asking for the same blocked action in slightly different forms.

Observability

Approvals should appear as first-class spans in traces, not as notes attached to a final answer. A trace should show the policy decision that required approval, the approval request, the wait duration, the approver, the decision, the resume token, and the final side effect or stop reason.

Useful production metrics include approval volume by risk class, approval latency, denial rate, changes-requested rate, expired approval rate, stale-approval rejection rate, rubber-stamp rate, override rate, and missed-approval incidents. These metrics are not just operational. They are feedback on whether the autonomy level is set correctly.

Failure Modes

- Approval request lacks evidence, so the human rubber-stamps or guesses.
- Approval text hides the actual side effect.
- The system asks for approval after the tool already executed.
- A retry reuses approval for a different action.
- Approval expires but the workflow resumes anyway.
- Approver identity and reason are not recorded.
- Approval fatigue causes humans to approve everything.
- Denied approvals are converted into softer prompts and retried until they pass.
- The audit trail records the final answer but not the proposed action and decision.
- The approval is attached to a conversation, not to an exact tool call.
- A batch approval covers actions that were not visible when the batch was approved.
- The model re-plans after approval and executes a different payload.
- The approver sees polished rationale but not the source evidence.

Evaluation Strategy

Approval evals should prove that risky actions pause and safe actions do not create unnecessary friction.

- Test high-risk actions that must require approval.
- Test low-risk read-only actions that should not require approval.
- Test denied approval and verify the side effect does not execute.
- Test expired approval and verify the workflow does not resume.
- Test changed action after approval and require a new approval.
- Test missing evidence and require `changes_requested` or escalation.
- Test retry behavior with idempotency keys.
- Test policy-version changes between request and resume.
- Test batch approvals with one out-of-policy item.
- Test prompt injection in evidence that asks the approver to ignore policy.
- Test audit completeness: request, decision, approver, policy version, and trace ID.

A compact eval fixture can make the approval boundary explicit:

```
{  
  "case_id": "refund_requires_finance_approval",  
}
```

```

"proposed_action": {
  "tool": "refunds.issue_refund",
  "tool_version": "2026-06-17",
  "amount_cents": 12500,
  "args_hash": "sha256:7c4b..."
},
"expected": {
  "requires_approval": true,
  "approver_role": "finance_approver",
  "must_include_evidence": ["order", "payment", "refund_policy"],
  "must_not_execute_before_approval": true,
  "must_reject_if_args_hash_changes": true,
  "required_audit_fields": ["approval_id", "decided_by", "policy_version", "trace_id"]
}
}

```

Measure approval routing accuracy, unnecessary approval rate, denied-action execution rate, stale-approval reuse, approval latency, audit completeness, rubber-stamp rate, and human override rate.

Treat denied-action execution and stale-approval reuse as blocking failures, not average-quality metrics. Segment approval latency and override rate by risk class, action type, policy version, and approver role so a healthy low-risk flow does not hide a weak high-risk control.

For the shared eval case contract and release-gate method, see [Evaluation-Driven Agent Development](#).

Production Checklist

- Define which actions require approval by policy, not prompt wording.
- Include proposed action, side effects, evidence, policy reason, and risk in every request.
- Keep the action envelope machine-readable.
- Persist workflow state while waiting.
- Bind approval to an exact action ID and idempotency key.
- Bind approval to tool version, arguments hash, tenant, resource, and policy version.
- Set expiration, timeout, cancellation, and escalation behavior.
- Record approver identity, decision, reason, policy version, and trace ID.
- Prevent denied or expired approvals from being retried silently.
- Reject approvals when the action changes after review.
- Treat approvals as trace spans and audit records.
- Track approval volume and approval fatigue.
- Keep approval policies, request schemas, and decision records versioned.
- Convert serious approval misses into regression evals.

Run the Example

```
npm run approval-gate
npm run approval-gate:test
```

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

[human-in-the-loop-approval-agent/typescript/src/approval_gate.ts](#)

[Open full source](#)

```
import {
  actionId,
  stableHash,
  type ApprovalDecision,
  type ApprovalRequest,
  type ApprovalResult,
  type ProposedAction,
} from "./approval_contract.ts";

export * from "./approval_contract.ts";

export class ApprovalGate {
  private readonly consumedKeys = new Set<string>();

  async resume(
    request: ApprovalRequest,
    decision: ApprovalDecision,
    action: ProposedAction,
    now: Date,
    execute: (action: ProposedAction) => Promise<void>,
  ): Promise<ApprovalResult> {
    const audit = [
      `approval:${request.approvalId}:decision:${decision.decision}`,
    ];

    if (decision.decision !== "approved") {
```

```

    return { status: "rejected", reason: "not_approved", audit };
}

if (
    decision.approvalId !== request.approvalId ||
    decision.traceId !== request.traceId ||
    decision.decidedByRole !== request.approverRole
) {
    return { status: "rejected", reason: "approval_mismatch", audit };
}

const currentActionId = actionId(action, request.policyVersion);
const currentArgsHash = stableHash(action.args);
if (
    currentActionId !== request.actionId ||
    currentActionId !== decision.approvedActionId ||
    currentArgsHash !== request.argsHash ||
    currentArgsHash !== decision.approvedArgsHash
) {
    return { status: "rejected", reason: "action_changed", audit };
}

if (decision.policyVersion !== request.policyVersion) {
    return { status: "rejected", reason: "policy_changed", audit };
}

if (now.getTime() >= new Date(request.expiresAt).getTime()) {
    return { status: "rejected", reason: "approval_expired", audit };
}

if (this.consumedKeys.has(action.idempotencyKey)) {
    return {
        status: "rejected",
        reason: "idempotency_key_consumed",
        audit,
    };
}

this.consumedKeys.add(action.idempotencyKey);
try {
    await execute(action);
} catch (error) {
    this.consumedKeys.delete(action.idempotencyKey);
    throw error;
}

```

```

    }
    audit.push(`approval:${request.approvalId}:executed:${request.actionId}`);
    return { status: "executed", audit };
  }
}

```

[human-in-the-loop-approval-agent/typescript/src/approval_contract.ts](#)

[Open full source](#)

```

import { createHash } from "node:crypto";

export type ProposedAction = {
  tool: "refunds.issue_refund";
  toolVersion: string;
  args: {
    orderId: string;
    amountCents: number;
  };
  tenantId: string;
  actorId: string;
  resourceIds: string[];
  idempotencyKey: string;
};

export type ApprovalRequest = {
  approvalId: string;
  actionId: string;
  traceId: string;
  proposedAction: ProposedAction;
  argsHash: string;
  evidenceRefs: string[];
  policyVersion: string;
  approverRole: "finance_approver";
  expiresAt: string;
};

export type ApprovalDecision = {
  approvalId: string;
  decision: "approved" | "denied";
  decidedBy: string;
  decidedByRole: "finance_approver";
  decidedAt: string;
  reason: string;
};

```



```

    approvedActionId?: string;
    approvedArgsHash?: string;
    policyVersion: string;
    traceId: string;
};

export type ApprovalResult =
  | { status: "executed"; audit: string[] }
  | {
      status: "rejected";
      reason:
        | "not_approved"
        | "approval_mismatch"
        | "action_changed"
        | "policy_changed"
        | "approval_expired"
        | "idempotency_key_consumed";
      audit: string[];
    };

function canonicalJson(value: unknown): string {
  if (Array.isArray(value)) {
    return `[${value.map(canonicalJson).join(",")}]`;
  }
  if (value && typeof value === "object") {
    const entries = Object.entries(value as Record<string, unknown>)
      .sort(([left], [right]) => left.localeCompare(right))
      .map(([key, item]) => `${JSON.stringify(key)}:${canonicalJson(item)}`);
    return `{${entries.join(",")}}`;
  }
  return JSON.stringify(value);
}

export function stableHash(value: unknown): string {
  return createHash("sha256").update(canonicalJson(value)).digest("hex");
}

export function actionId(
  action: ProposedAction,
  policyVersion: string,
): string {
  return stableHash({ ...action, policyVersion });
}

```

```

export function createApprovalRequest(input: {
  approvalId: string;
  traceId: string;
  action: ProposedAction;
  evidenceRefs: string[];
  policyVersion: string;
  expiresAt: string;
}): ApprovalRequest {
  return {
    approvalId: input.approvalId,
    actionId: actionId(input.action, input.policyVersion),
    traceId: input.traceId,
  }
}

```

Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.

human-in-the-loop-approval-agent/typescript/test/approval_gate.spec.ts

[Open full source](#)

```

import {
  actionId,
  ApprovalGate,
  createApprovalRequest,
  stableHash,
  type ApprovalDecision,
  type ProposedAction,
} from "../src/approval_gate.ts";

function assert(condition: unknown, message: string): asserts condition {
  if (!condition) throw new Error(message);
}

const action: ProposedAction = {
  tool: "refunds.issue_refund",
  toolVersion: "2026-06-18",
  args: { orderId: "ORD-104", amountCents: 12500 },
  tenantId: "tenant-a",
  actorId: "support-agent",
  resourceIds: ["ORD-104"],
  idempotencyKey: "refund:ORD-104:12500",
};

const request = createApprovalRequest({

```

```

    approvalId: "APR-104",
    traceId: "trace-104",
    action,
    evidenceRefs: ["order:ORD-104", "policy:refunds-2026"],
    policyVersion: "refund-policy-v3",
    expiresAt: "2026-06-18T12:30:00.000Z",
  });

function decision(
  value: "approved" | "denied" = "approved",
): ApprovalDecision {
  return {
    approvalId: request.approvalId,
    decision: value,
    decidedBy: "finance-user-7",
    decidedByRole: "finance_approver",
    decidedAt: "2026-06-18T12:05:00.000Z",
    reason: value === "approved" ? "Evidence supports refund." : "Insufficient evidence.",
    approvedActionId:
      value === "approved"
        ? actionId(action, request.policyVersion)
        : undefined,
    approvedArgsHash:
      value === "approved" ? stableHash(action.args) : undefined,
    policyVersion: request.policyVersion,
    traceId: request.traceId,
  };
}

async function attempt(
  gate: ApprovalGate,
  approval: ApprovalDecision,
  proposedAction: ProposedAction,
  now = new Date("2026-06-18T12:10:00.000Z"),
) {
  let executions = 0;
  const result = await gate.resume(
    request,
    approval,
    proposedAction,
    now,
    async () => {
      executions += 1;
    },
  );

```

```

    );
    return { result, executions };
}

const approved = await attempt(new ApprovalGate(), decision(), action);
assert(approved.result.status === "executed", "Approved action must execute");
assert(approved.executions === 1, "Approved action must execute once");

const denied = await attempt(new ApprovalGate(), decision("denied"), action);
assert(denied.result.status === "rejected", "Denied action must be rejected");
assert(denied.executions === 0, "Denied action must not execute");

const expired = await attempt(
    new ApprovalGate(),
    decision(),
    action,
    new Date("2026-06-18T12:30:00.000Z"),
);
assert(
    expired.result.status === "rejected" &&
    expired.result.reason === "approval_expired",
    "Expired approval must be rejected",
);

```

Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.

Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `human-in-the-loop-approval-agent/` folder from this repository.

Related Patterns

- [Tool Capability Design](#)
- [MCP-first Tool Use](#)
- [Policy Enforcement](#)
- [Durable Workflows](#)
- [Observability and Evals](#)
- [Cost Controls and Runtime Budgets](#)
- [Agent Threat Model](#)
- [Agent UX and Human Trust](#)

- [Evaluation-Driven Agent Development](#)
- [Pattern Evaluation Checklist](#)

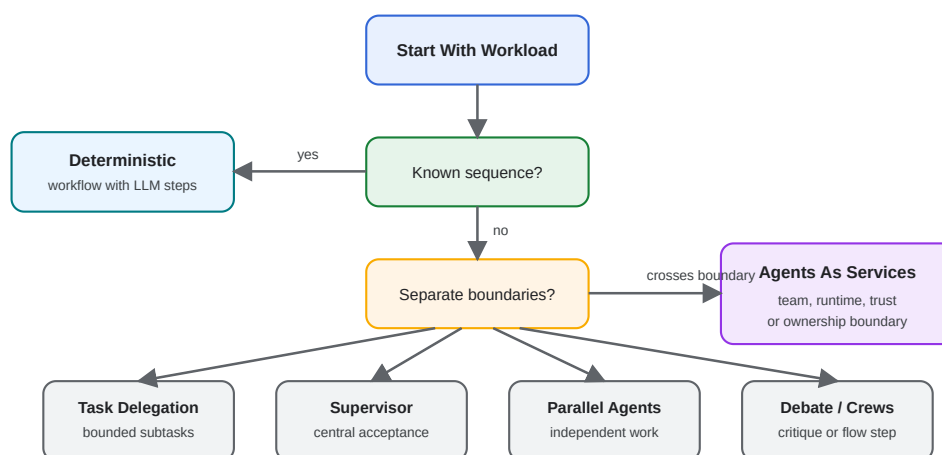
Choosing Multi-Agent Topology

Do not start a design by asking how many agents you need.

Start by asking where the work needs separate context, separate tools, separate authority, separate timing, or separate accountability. If those boundaries are real, multiple agents can help. If those boundaries are not real, multiple agents usually add cost, latency, and failure modes.

Multi-agent design is not about making the system look more intelligent. It is about creating useful boundaries.

Multi-Agent Topology Selection



The default should still be conservative. Multi-agent design is a coordination tradeoff, not an intelligence upgrade. Every extra agent adds protocol surface, context selection, tool permissions, traces, latency, cost, failure modes, and ownership questions. Add an agent only when the boundary it creates is worth that cost.

First Rule

Use the smallest topology that gives you the boundary you need.

Many systems do not need multiple agents. They need:

- a deterministic workflow;
- one agent loop with better tools;
- routing between workflows;
- a reviewer step;
- stronger context management;
- better evals;
- better observability.

Multiple agents are justified when one agent would have to carry too much context, too much authority, too many tools, or too many conflicting responsibilities.

When To Split Agents

Split agents for engineering reasons, not because the problem sounds complex.

Good reasons to split:

- **Ownership:** different teams own different capabilities, policies, releases, or incidents.
- **Context:** each part needs a different working set and should not see everything.
- **Authority:** one part can read data, another can draft actions, and another can approve side effects.
- **Tools:** tool sets are different enough that sharing them would increase risk.
- **Timing:** some work is interactive, some is asynchronous, and some waits for approval.
- **Scaling:** one capability needs different deployment, latency, cost, or model routing.
- **Evaluation:** each capability can be tested with its own fixtures and acceptance criteria.

Bad reasons to split:

- the single-agent prompt is messy;
- the team hopes more agents will compensate for unclear goals;
- every agent will receive the same context and tools;
- no one can define the handoff contract;
- the merge policy is “the model will figure it out”;
- there is no trace across handoffs.

If the split does not create a clearer boundary, it is probably hiding weak architecture.

Topology Decision Matrix

Situation	Prefer	Avoid
Known sequence with clear rules	Deterministic workflow	Multi-agent chat.
One task needs specialist decomposition	Task delegation	One general agent with a giant prompt.
Central owner must control quality and merge results	Supervisor-worker	Peer agents with no final owner.
Independent work can run concurrently	Parallel agents	Sequential agent chains that waste time.
Output improves through critique or comparison	Debate and consensus	Voting without evidence or tests.
Workflow state matters more than agent chatter	Flow with crews	Crew as substitute for state design.

Situation	Prefer	Avoid
Agents cross team, runtime, or trust boundaries	Agents as services over A2A, MCP, events, or workflows	Unstructured natural language handoffs.
Agents need asynchronous coordination over time	Board, queue, durable workflow, or task ledger	Shared context windows and manual copy-paste.

The right topology is usually boring. That is a feature.

Topology	Best For	Main Cost	Required Control
Single agent	One bounded task with one tool/policy surface.	Limited specialization.	Strong loop state, budget, and tool policy.
Workflow with agent steps	Known process with uncertain substeps.	Workflow design effort.	Durable state and explicit transitions.
Supervisor-worker	Specialist work with one final owner.	Coordination and merge complexity.	Typed assignments and merge gate.
Parallel agents	Independent work that benefits from concurrency.	Higher spend and trace volume.	Fan-in reducer and per-worker evidence.
Debate or consensus	Ambiguous judgment that benefits from critique.	Latency and false confidence.	Evidence-backed judge and accountable owner.
Agents as services	Team, runtime, security, or deployment boundaries.	Service contracts and versioning.	API/A2A/MCP contract, auth, ownership, evals.
A2A interoperability	Remote agent collaboration across boundaries.	Protocol and lifecycle complexity.	Message envelope, scopes, idempotency, traces.

A topology selector should encode the same bias:

```
interface WorkloadShape {
  knownSequence: boolean;
  independentSubtasks: boolean;
  needsSpecialists: boolean;
  needsCentralAcceptance: boolean;
  benefitsFromCritique: boolean;
  crossesTrustBoundary: boolean;
}

function chooseTopology(workload: WorkloadShape) {
  if (workload.knownSequence && !workload.needsSpecialists) {
    return 'deterministic_workflow';
  }
}
```



```

    }

    if (workload.crossesTrustBoundary) {
        return 'agents_as_services';
    }

    if (workload.independentSubtasks && workload.needsCentralAcceptance) {
        return 'supervisor_worker';
    }

    if (workload.independentSubtasks) {
        return 'parallel_agents';
    }

    if (workload.benefitsFromCritique) {
        return 'debate_and_consensus';
    }

    return 'single_agent_or_workflow';
}

```

The point is not to automate architecture. The point is to make the selection criteria visible enough to argue with.

Topology Contract

Before implementation, write down the topology contract. It can be lightweight, but it should be explicit:

```

type TopologyContract = {
  topology:
    | 'single_agent'
    | 'workflow_with_agent_steps'
    | 'supervisor_worker'
    | 'parallel_agents'
    | 'debate_consensus'
    | 'agents_as_services'
    | 'a2a_interoperability';
  owner: string;
  reasonForSplit: string;
  agents: Array<{
    name: string;
    capability: string;
    ownsState: string[];
    allowedTools: string[];

```

```

    forbiddenTools: string[];
    contextRefs: string[];
    requiredScopes: string[];
  }>;
  handoffContract: {
    protocol: 'internal_call' | 'REST' | 'gRPC' | 'MCP' | 'A2A' | 'queue' | 'workflow';
    inputSchema: string;
    outputSchema: string;
    timeoutMs: number;
    idempotencyRequired: boolean;
  };
  evaluation: {
    baseline: 'single_agent' | 'deterministic_workflow';
    successMetrics: string[];
    coordinationMetrics: string[];
  };
};
};

```

The contract forces the key question: what does the topology buy that a simpler design does not?

Declare The Topology Before You Code It

A multi-agent system should have a visible topology before it has framework glue. Write down:

- agents and their responsibilities;
- allowed communication paths;
- fan-out and fan-in points;
- gates and approval boundaries;
- shared state and private state;
- failure and cancellation behavior.

This can be a diagram, a table, or a small declarative file. The important property is reviewability: a reader should understand the agent graph without reverse-engineering callbacks and prompts.

Deterministic Workflow First

If code can own the sequence, let code own the sequence.

Use a deterministic workflow when:

- the steps are known;
- branching rules are clear;
- correctness depends on policy or business rules;
- state transitions must be auditable;
- latency and cost matter;

- human approval must pause the process.

The model can still be useful inside the workflow. It can classify, extract, summarize, draft, rank, or critique. But the workflow owns the path.

Task Delegation

Use task delegation when the work decomposes into bounded subtasks.

Good delegation has:

- clear subtask contracts;
- scoped context for each worker;
- expected output shape;
- acceptance criteria;
- one owner for final merge;
- evals that prove decomposition helps.

Bad delegation is just asking several agents to “work on this” and hoping the combined answer is better.

Supervisor-Worker

Use supervisor-worker when a central agent or workflow must own the goal, state, routing, and final acceptance.

This topology is useful when:

- workers have different tools or context;
- the supervisor can evaluate worker outputs;
- there is one accountable final answer;
- some worker failures should not fail the whole task;
- workers must not see each other’s full context.

The supervisor should not become a hidden god agent. It should own coordination, not every detail of execution.

Parallel Agents

Use parallel agents when work is independent and the merge policy is clear.

Good cases:

- search multiple sources;
- generate several candidate plans;
- review the same artifact from different angles;
- run separate checks such as security, correctness, and style;
- compare outputs from different models or prompts.

Parallelism is not free. It increases cost, trace volume, and merge complexity. Use it when it buys latency reduction, quality lift, or independent coverage.

Debate And Consensus

Use debate or consensus when independent critique improves judgment.

It is useful for:

- ambiguous decisions;
- design reviews;
- risk analysis;
- comparing candidate answers;
- exposing weak assumptions.

It is weak for:

- factual questions that need retrieval;
- tool decisions that need policy;
- tasks where every agent shares the same blind spot;
- high-risk actions where voting replaces authority.

Consensus is not proof. It is a signal that still needs evidence, tests, or an accountable owner.

Crews And Flows

Use crews inside flows when the workflow needs explicit state and a group of specialized agents performs one bounded step.

The flow should own:

- state;
- event order;
- retries;
- approvals;
- persistence;
- stop conditions.

The crew should own:

- local collaboration;
- specialist roles;
- bounded output for one workflow step.

If crew-local chat becomes the state machine, the architecture is weak.

Agents As Services

When agents cross process, team, runtime, vendor, or trust boundaries, treat them like services.

That means:

- explicit capability contract;
- typed request and response;
- authentication;
- authorization;
- timeouts;
- idempotency;
- refusal semantics;
- cancellation;
- trace correlation;
- versioning;
- contract tests.

The protocol may be A2A, MCP, REST, gRPC, event streams, or a durable workflow engine. The important thing is not the protocol brand. The important thing is that the boundary is explicit and testable.

This is where the microservices analogy is useful. Splitting agents is reasonable when it creates clearer ownership, contracts, deployment, security, scaling, and observability. It is not reasonable when it only turns one vague agent into several vague agents.

Use A2A or service-style communication when the remote agent has its own lifecycle: capability discovery, authorization, progress, refusal, cancellation, versioning, and trace ownership. Use MCP when the boundary is primarily a tool or capability. Use workflows when state and approvals matter more than direct agent-to-agent messaging.

Board, Queue, Or Ledger Coordination

Some multi-agent work should not be coordinated through direct chat.

Use a board, queue, task ledger, or durable workflow when:

- work happens asynchronously;
- humans and agents both participate;
- decisions need to stay attached to work items;
- agents run in separate sessions;
- progress must survive restarts;
- handoffs need audit history.

This is especially useful for coding agents, operations agents, research pipelines, and long-running review workflows.

Shared State

Shared state is where multi-agent systems often fail.

Avoid:

- every agent writing to the same memory;
- hidden shared scratchpads;
- unversioned summaries;
- agents overwriting each other's conclusions;
- state stored only in conversation history.

Prefer:

- one owner for each state object;
- append-only task logs;
- explicit merge steps;
- durable workflow state;
- typed artifacts;
- human-readable handoff records;
- trace IDs across agents.

If shared state is unclear, the topology is not ready.

Handoffs And Ownership

Every handoff should answer three questions:

1. Who owns the task now?
2. What exactly was handed off?
3. What condition returns ownership, escalates, or stops the run?

Without ownership, multi-agent systems become distributed ambiguity. A worker can be useful only if the caller knows what the worker accepted, refused, completed, or could not decide. That means handoffs need schemas, status values, stop reasons, and traces.

Evaluation Guidance

A multi-agent system should beat a simpler baseline.

Evaluate:

- single-agent baseline versus multi-agent topology;
- quality lift;
- latency cost;
- token and tool cost;
- merge accuracy;
- worker failure handling;
- disagreement handling;
- trace completeness;
- permission isolation;

- context isolation;
- final accountability.

Measure coordination as a first-class cost:

Metric	Why It Matters
Handoff correctness	Proves work is routed to the right agent with the right contract.
Coordination overhead	Shows whether extra agents spend more effort than they save.
Merge quality	Proves the final answer preserves evidence and disagreement.
Ownership clarity	Shows whether failures have an accountable component.
Trace completeness	Makes distributed failures debuggable.
Tool isolation	Proves agents cannot use tools outside their role.
Context isolation	Proves agents do not receive irrelevant or unsafe context.
Cost and latency by topology	Shows whether the topology is operationally worth it.

Include negative evals where the system should choose a simpler topology. A good topology selector should know when not to use multiple agents.

Failure Modes

- Multi-agent design is used to hide unclear goals or weak state.
- Every agent gets the same context, tools, memory, and permissions.
- Workers can create side effects without supervisor, policy, or approval checks.
- Handoffs are natural-language blobs with no schema or owner.
- Agents delegate tasks back and forth with no delegation budget.
- Debate produces agreement but no evidence.
- Parallel agents duplicate work because boundaries are unclear.
- Agents as services have no versioned contract, auth, or idempotency rule.
- The final answer hides worker failure, disagreement, or missing evidence.
- There is no trace that connects assignments, handoffs, worker outputs, merge decisions, and stop reason.

Design Checklist

Before adding another agent, answer:

- What boundary does this new agent create?
- Does it need separate context?
- Does it need separate tools?
- Does it need separate permissions?

- Does it need independent timing?
- Who owns final acceptance?
- How are outputs merged?
- What happens if this agent fails, refuses, or times out?
- What state does it own?
- What context must it never see?
- What protocol or contract governs the handoff?
- What prevents delegation loops?
- What is the simpler baseline?
- How do traces connect across agents?
- What eval proves this topology is better than the simpler baseline?

If the only reason for another agent is “it feels more agentic,” do not add it.

Related Chapters

- [Task Delegation](#)
- [Supervisor / Worker](#)
- [Parallel Agents](#)
- [Debate and Consensus](#)
- [CrewAI Flows and Crews](#)
- [Agents As Services](#)
- [Pattern Evaluation Checklist](#)
- [A2A Agent Interoperability](#)
- [Context Budgets and Working Sets](#)
- [Evaluation-Driven Agent Development](#)
- [Production Runtime Overview](#)
- [Observability and Evals](#)

Task Delegation

Task delegation assigns bounded subtasks to specialized workers and combines their outputs.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

Intent

Task delegation assigns bounded subtasks to specialized workers and combines their outputs.

Use When

- Specialized workers can complete independent subtasks better than one general agent.
- The manager can define expected outputs, constraints, and acceptance criteria.
- Subtask results can be merged and checked.

Avoid When

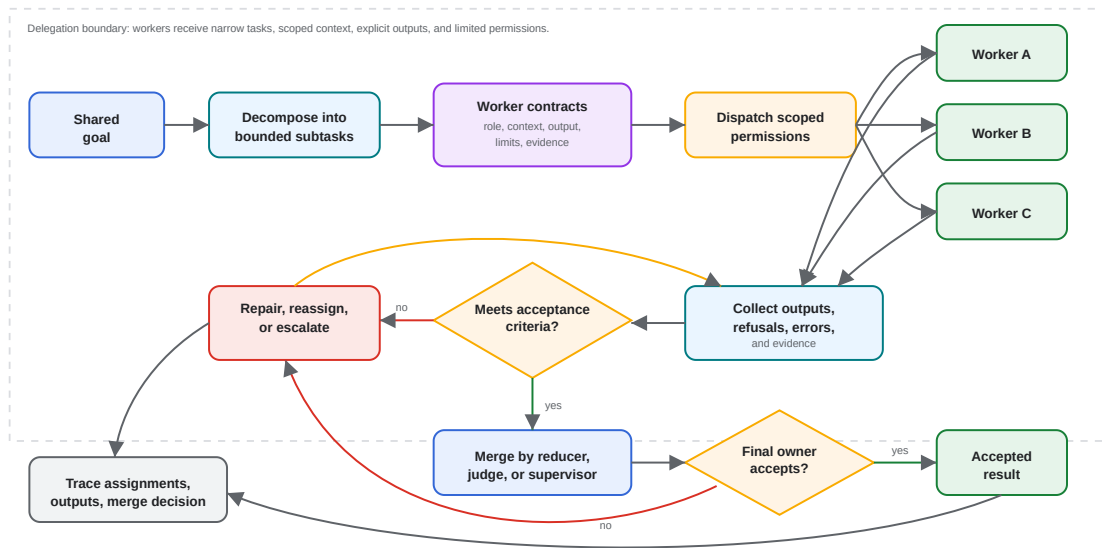
- The task has no useful decomposition.
- Workers share hidden mutable state.
- The manager cannot evaluate returned work.

Architecture

Use this diagram to read Task Delegation as a system boundary, not only a code shape. The key ownership question is: the coordinator owns the shared goal, decomposition, assignments, merge policy, and final acceptance.

Task Delegation Contract Flow

The coordinator owns decomposition, contracts, merge policy, final acceptance, and escalation.



Read it as an ownership contract: the coordinator owns decomposition, worker scope, merge policy, final acceptance, and escalation.

System Shape

- **Pattern boundary:** a coordinator delegates bounded work to agents with narrow roles, then evaluates and merges their outputs.
- **State owner:** the coordinator owns the shared goal, decomposition, assignments, merge policy, and final acceptance.
- **Primary artifact:** `task-delegation-pattern/` contains the runnable reference implementation and examples.
- **Operational promise:** Task delegation assigns bounded subtasks to specialized workers and combines their outputs.
- **Runnable path:** start with `npm run task-delegation` before adapting the pattern to a larger system.

Core Protocol

1. Define the shared goal, worker roles, expected outputs, and acceptance criteria.
2. Split work only where independent or specialist execution adds value.
3. Dispatch tasks with scoped context and permissions.
4. Collect outputs, errors, refusals, and evidence from each worker.
5. Merge results through an explicit judge, reducer, supervisor, or human review gate.

Implementation Notes

- Keep the pattern boundary explicit: inputs, state, side effects, and outputs should be visible.
- Validate model-produced decisions before they affect tools, users, or durable state.

- Emit enough trace data to debug failures after the run.

Failure Modes

- The pattern is applied where a simpler deterministic workflow would be better.
- State, tool calls, or model decisions are not observable enough to debug.
- The system lacks clear stop, retry, or escalation behavior.

Evaluation Strategy

- Compare multi-agent output against a single-agent baseline on the same tasks.
- Test worker disagreement, worker failure, duplicated work, and bad merge decisions.
- Measure quality lift, latency cost, token cost, merge accuracy, and accountability.
- Include cases that prove each “Use When” condition is true for this pattern.
- Include negative cases from “Avoid When” so the system chooses a simpler or safer pattern when appropriate.

Production Checklist

- Give every worker a narrow contract and permission set.
- Make the merge policy explicit before workers run.
- Log per-worker inputs, outputs, and decision evidence.
- Keep one owner for final acceptance and escalation.
- Define human escalation for ambiguous, high-risk, or policy-blocked work.
- Keep the source bundle, generated chapter, tests, and deployment artifact in the same release.

Run the Example

```
npm run task-delegation
```

Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
task-delegation-pattern/autogen_typescript_example/task_delegation.ts
```

[Open full source](#)

```

// Task Delegation Pattern - Autogen TypeScript Example
// To run: npm install && npm run task-delegation

import axios from 'axios';
import * as readline from 'readline';
import * as dotenv from 'dotenv';
dotenv.config();

const MISTRAL_API_URL = 'https://api.mistral.ai/v1/chat/completions';
const MISTRAL_API_KEY = process.env.MISTRAL_API_KEY;

async function managerAgent(task: string): Promise<string> {
  // Step 1: Decompose the task
  const decomposePrompt = `You are a manager agent. Decompose the following task into two
subtasks and describe each one. Task: ${task}`;
  const decomposeResp = await axios.post(
    MISTRAL_API_URL,
    {
      model: 'mistral-tiny',
      messages: [{ role: 'user', content: decomposePrompt }],
    },
    {
      headers: {
        'Authorization': `Bearer ${MISTRAL_API_KEY}`,
        'Content-Type': 'application/json',
      },
    }
  );
  const subtasks =
decomposeResp.data.choices[0].message.content.split(/\n|\r/).filter(Boolean);
  const subtask1 = subtasks[0] || 'Subtask 1';
  const subtask2 = subtasks[1] || 'Subtask 2';

  // Step 2: Delegate to workers
  const worker1Prompt = `You are Worker Agent 1. Complete this subtask: ${subtask1}`;
  const worker2Prompt = `You are Worker Agent 2. Complete this subtask: ${subtask2}`;
  const [worker1Resp, worker2Resp] = await Promise.all([
    axios.post(MISTRAL_API_URL, {
      model: 'mistral-tiny',
      messages: [{ role: 'user', content: worker1Prompt }],
    }, {
      headers: {
        'Authorization': `Bearer ${MISTRAL_API_KEY}`,
        'Content-Type': 'application/json',

```

```

    },
  }),
  axios.post(MISTRAL_API_URL, {
    model: 'mistral-tiny',
    messages: [{ role: 'user', content: worker2Prompt }],
  }, {
    headers: {
      'Authorization': `Bearer ${MISTRAL_API_KEY}`,
      'Content-Type': 'application/json',
    },
  })
]);

// Step 3: Aggregate results
return `Results:\n- ${worker1Resp.data.choices[0].message.content}\n-
${worker2Resp.data.choices[0].message.content}`;
}

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('Task: ', async (userInput: string) => {
  try {
    const result = await managerAgent(userInput);
    console.log(result);
  } catch (err) {
    console.error('Error:', err);
  }
  rl.close();
});

```

task-delegation-pattern/langgraph_python_example/task_delegation.py

[Open full source](#)

```
# Task Delegation Pattern - LangGraph Python Example
```

This example demonstrates the Task Delegation Pattern using LangGraph and Python. A manager agent decomposes a task and delegates subtasks to two worker agents, then aggregates the results. The LLM is Mistral.

```
## Requirements
```

```
- Python 3.8+
- `langgraph` library
- `python-dotenv` (for .env support)
- Mistral LLM API access
```

```
## Install dependencies
```

```
```bash
pip install langgraph python-dotenv requests
```
```

```
## Example Code
```

```
```python
import os
from langgraph import Agent, Environment, LLM
from dotenv import load_dotenv

load_dotenv()

MISTRAL_API_KEY = os.getenv("MISTRAL_API_KEY")
MISTRAL_API_URL = "https://api.mistral.ai/v1/chat/completions"

class SimpleEnvironment(Environment):
 def get_observation(self):
 return input("Task: ")
 def send_action(self, action):
 print(action)

class ManagerAgent(Agent):
 def __init__(self, llm):
 self.llm = llm
 def act(self, task):
 prompt = f"You are a manager agent. Decompose the following task into two subtasks and describe each one. Task: {task}"
 return self.llm.complete(prompt)

class WorkerAgent(Agent):
 def __init__(self, llm, name):
 self.llm = llm
 self.name = name
 def act(self, subtask):
 prompt = f"You are {self.name}. Complete this subtask: {subtask}"
```

```

 return self.llm.complete(prompt)

llm = LLM(
 provider="mistral",
 api_key=MISTRAL_API_KEY,
 api_url=MISTRAL_API_URL,
)

env = SimpleEnvironment()
manager = ManagerAgent(llm)
worker1 = WorkerAgent(llm, "Worker Agent 1")
worker2 = WorkerAgent(llm, "Worker Agent 2")

task = env.get_observation()
subtasks = manager.act(task).split("\n")
subtask1 = subtasks[0] if len(subtasks) > 0 else "Subtask 1"
subtask2 = subtasks[1] if len(subtasks) > 1 else "Subtask 2"
result1 = worker1.act(subtask1)
result2 = worker2.act(subtask2)
final = f"Results:\n- {result1}\n- {result2}"
env.send_action(final)
'''
'''

- Try a complex or multi-step task to see delegation in action.
- Make sure your `.env` file contains your Mistral API key.

```

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `task-delegation-pattern/` folder from this repository.

## Related Patterns

- [Supervisor / Worker](#)
- [Debate and Consensus](#)
- [Parallel Agents](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# Supervisor / Worker

Supervisor/Worker centralizes goal ownership, task state, routing, and quality gates while workers perform bounded specialist work.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

Supervisor / Worker is a multi-agent pattern where one coordinator owns the goal, decomposition, worker contracts, merge policy, quality gate, and final accountability. Workers perform bounded specialist tasks and return structured results with evidence, errors, and confidence.

The supervisor is not just a manager-shaped prompt. It is the control point that prevents multi-agent work from becoming a group chat with no owner. It decides what work should be split, what context each worker receives, which tools each worker may use, how results are merged, and when the system must escalate.

## Use When

- The task benefits from separate specialist workstreams.
- Workers can operate with scoped context, scoped tools, and clear outputs.
- The supervisor can evaluate, merge, or reject worker results.
- Parallelism, specialization, or independent review buys more than the extra coordination cost.
- One component can own final acceptance and user-facing accountability.

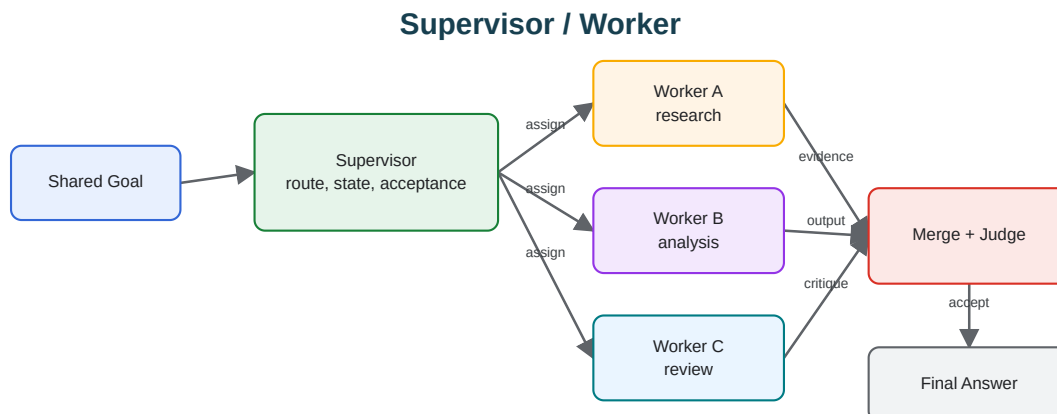
## Avoid When

- A single agent or deterministic workflow can solve the task clearly.
- Workers would all receive the same context and tool list.
- The supervisor cannot judge whether worker output is correct.
- The merge step is vague or purely stylistic.
- Worker actions can create side effects without supervisor policy or approval.
- Nobody owns the final answer when workers disagree.

## Architecture



Use this diagram to read Supervisor / Worker as a system boundary, not only a code shape. The key ownership question is: the coordinator owns the shared goal, decomposition, assignments, merge policy, and final acceptance.



## System Shape

- **Pattern boundary:** the supervisor owns the shared goal, decomposition, assignments, merge policy, quality checks, escalation, and final acceptance.
- **Worker boundary:** each worker receives a narrow task, scoped context, allowed tools, expected output schema, timeout, and acceptance criteria.
- **State owner:** the supervisor or workflow engine owns shared state. Workers own only task-local scratch state unless explicitly granted more.
- **Policy boundary:** worker tool use and side effects are constrained by the assignment, not by general worker ability.
- **Operational promise:** the system gets useful specialization without losing one accountable owner for the final result.

## Core Protocol

1. Receive the shared goal, caller, constraints, budget, and trace ID.
2. Decide whether the task should be split or handled by a simpler baseline.
3. Create worker assignments with scoped context, allowed tools, expected outputs, timeout, and acceptance criteria.
4. Dispatch assignments and record per-worker trace IDs.
5. Collect worker results, refusals, errors, timeouts, and evidence.
6. Evaluate each worker result before merging.
7. Merge accepted results using an explicit merge policy.
8. Return, retry, reassign, or escalate with one final owner.

## Implementation Notes

The handoff from supervisor to worker should be a contract, not a paragraph of instructions.

```
type WorkerAssignment = {
 taskId: string;
 parentTraceId: string;
 workerRole: 'policy_reviewer' | 'order_investigator' | 'customer_message_drafter';
 objective: string;
 scopedContextRefs: string[];
 allowedTools: string[];
 forbiddenTools: string[];
 timeoutMs: number;
 expectedOutput: {
 schema: 'policy_findings.v1' | 'order_findings.v1' | 'message_draft.v1';
 requiredFields: string[];
 };
 acceptanceCriteria: string[];
};
```

The worker result should be equally structured:

```
type WorkerResult = {
 taskId: string;
 workerRole: string;
 status: 'succeeded' | 'refused' | 'needs_human' | 'failed' | 'timed_out';
 output?: unknown;
 evidenceRefs: string[];
 confidence: 'low' | 'medium' | 'high';
 errors: string[];
 traceId: string;
};
```

A small merge gate keeps the supervisor honest:

```
function acceptWorkerResult(result: WorkerResult) {
 if (result.status !== 'succeeded') return false;
 if (result.evidenceRefs.length === 0) return false;
 if (result.confidence === 'low') return false;
 return true;
}
```

The supervisor should not rubber-stamp worker output. It should check evidence, schema, confidence, disagreement, and policy before final synthesis.

## Failure Modes

- The supervisor decomposes work that did not need to be split.
- Workers all receive the full context, removing the benefit of isolation.
- Workers share broad tool access, multiplying the blast radius.
- Assignments do not specify expected output, so merge becomes guesswork.
- Workers duplicate work because boundaries are unclear.
- Worker disagreement is hidden inside a smooth final answer.
- The supervisor accepts output without checking evidence or policy.
- A failed or timed-out worker silently disappears from the final answer.
- No trace connects the final result back to worker inputs, outputs, and decisions.
- Nobody owns final accountability after the supervisor has merged results.

## Evaluation Strategy

Supervisor / Worker evals should prove that the topology is better than the simpler baseline and that the coordination boundary works.

- Compare against a single-agent baseline on the same tasks.
- Test a case where the system should choose not to split the task.
- Test worker timeout, worker refusal, worker failure, and malformed worker output.
- Test disagreement between workers and require visible resolution.
- Test context isolation: each worker should receive only what it needs.
- Test permission isolation: workers should not call tools outside their assignment.
- Test merge accuracy: the final result should preserve evidence and not invent consensus.
- Test final accountability: the supervisor must return one status, one owner, and one stop reason.

A compact eval fixture can make those expectations explicit:

```
{
 "case_id": "refund_policy_and_order_disagree",
 "goal": "Recommend whether a refund request should be approved.",
 "workers": {
 "policy_reviewer": { "status": "succeeded", "confidence": "high" },
 "order_investigator": { "status": "succeeded", "confidence": "medium" },
 "customer_message_drafter": { "status": "blocked_until_decision" }
 },
 "expected": {
 "final_status": "needs_human",
 "must_explain_disagreement": true,
 "forbidden_worker_tools": ["refunds.issue_refund", "support.send_customer_email"],
 "required_trace_events": ["assignment", "worker_result", "merge_decision", "stop"]
 }
}
```

Measure quality lift over baseline, latency cost, token cost, worker failure handling, merge accuracy, disagreement handling, permission isolation, context isolation, and trace completeness.

## Production Checklist

- Keep one owner for the final answer.
- Require a typed worker assignment for every worker call.
- Scope context and tools per worker.
- Give each worker a timeout, cancellation rule, and refusal path.
- Record assignment, worker input, worker output, merge decision, and stop reason.
- Make disagreement visible instead of smoothing it away.
- Define when the supervisor retries, reassigns, escalates, or stops.
- Compare quality and cost against a simpler single-agent baseline.
- Keep worker prompts, tool manifests, and output schemas versioned.
- Add circuit breakers for failing workers or unsafe tool proposals.

## Run the Example

```
npm run hierarchical-agent
```

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
hierarchical-agent-pattern/autogen_typescript_example/hierarchical_agent.ts
```

[Open full source](#)

```
import dotenv from 'dotenv';
dotenv.config();
import axios from 'axios';
import readline from 'readline';

const MISTRAL_API_KEY = process.env.MISTRAL_API_KEY;
const MISTRAL_API_URL = 'https://api.mistral.ai/v1/chat/completions';

type ChatMessage = {
```

```

 role: 'system' | 'user';
 content: string;
 };

function askLocalModel(messages: ChatMessage[]) {
 const systemPrompt = messages[0]?.content ?? '';
 const userPrompt = messages[messages.length - 1]?.content ?? '';

 if (systemPrompt.includes('Decompose the user goal')) {
 return [
 'Sub-task 1: Define evaluation criteria for answer quality, retrieval grounding, latency, and failure handling.',
 'Sub-task 2: Create a small test set with expected evidence, negative cases, and acceptance thresholds.',
].join('\n');
 }

 if (systemPrompt.includes('Worker Agent 1')) {
 return `Worker 1 result: ${userPrompt} Criteria should include citation accuracy, unsupported-claim rate, p95 latency, and visible refusal behavior.`;
 }

 if (systemPrompt.includes('Worker Agent 2')) {
 return `Worker 2 result: ${userPrompt} The test set should include grounded answers, missing-evidence questions, stale-document checks, and threshold failures.`;
 }

 return [
 'Final answer: evaluate the RAG prototype with quality, grounding, latency, and failure-handling criteria.',
 'Use a small test set with positive cases, missing-evidence cases, stale-document cases, and blocking thresholds.',
 'Accept the prototype only when worker evidence meets the supervisor policy.',
].join('\n');
}

async function askMistral(messages: ChatMessage[]) {
 if (!MISTRAL_API_KEY) {
 return askLocalModel(messages);
 }

 const response = await axios.post(
 MISTRAL_API_URL,
 {

```

```

 model: 'mistral-tiny',
 messages,
 },
 {
 headers: {
 'Authorization': `Bearer ${MISTRAL_API_KEY}`,
 'Content-Type': 'application/json',
 },
 }
);
return response.data.choices[0].message.content.trim();
}

async function main() {
 const rl = readline.createInterface({
 input: process.stdin,
 output: process.stdout,
 });

 rl.question('State the overall goal for the manager agent: ', async (goalInput) => {
 // Manager agent decomposes the goal into two sub-tasks
 const managerMessages: ChatMessage[] = [
 { role: 'system', content: 'You are a manager agent. Decompose the user goal into two sub-tasks and assign each to a worker agent.' },
 { role: 'user', content: `Goal: ${goalInput}` },
];

 const managerPlan = await askMistral(managerMessages);
 console.log(`\nManager Agent Plan:\n`, managerPlan);

 // Simulate two worker agents (for demo, just ask Mistral for each sub-task)
 const subTasks = managerPlan.match(/Sub-task [12]: (.*)/g) || [];
 const workerResults: string[] = [];
 for (let i = 0; i < subTasks.length; i++) {
 const workerMessages: ChatMessage[] = [
 { role: 'system', content: `You are Worker Agent ${i+1}. Complete the following sub-task as best as you can.` },
 { role: 'user', content: subTasks[i] },
];

 const workerResult = await askMistral(workerMessages);
 console.log(`\nWorker Agent ${i+1} Result:\n`, workerResult);
 workerResults.push(workerResult);
 }
 });
}

```

```
// Manager aggregates results
const aggregationMessages: ChatMessage[] = []
```

*Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.*

[hierarchical-agent-pattern/langgraph\\_python\\_example/hierarchical\\_agent.py](#)

[Open full source](#)

```
import os
import requests
import re

def ask_mistral(messages):
 api_key = os.getenv('MISTRAL_API_KEY')
 if not api_key:
 raise ValueError('Please set MISTRAL_API_KEY in your .env file')
 url = 'https://api.mistral.ai/v1/chat/completions'
 headers = {
 'Authorization': f'Bearer {api_key}',
 'Content-Type': 'application/json',
 }
 data = {
 'model': 'mistral-tiny',
 'messages': messages,
 }
 response = requests.post(url, headers=headers, json=data)
 response.raise_for_status()
 return response.json()['choices'][0]['message']['content'].strip()

def main():
 goal_input = input('State the overall goal for the manager agent: ')
 # Manager agent decomposes the goal into two sub-tasks
 manager_messages = [
 {'role': 'system', 'content': 'You are a manager agent. Decompose the user goal into two sub-tasks and assign each to a worker agent.'},
 {'role': 'user', 'content': f'Goal: {goal_input}'},
]
 manager_plan = ask_mistral(manager_messages)
 print('\nManager Agent Plan:\n', manager_plan)

 # Simulate two worker agents (for demo, just ask Mistral for each sub-task)
 sub_tasks = re.findall(r'Sub-task [12]: (.*)', manager_plan)
```

```

worker_results = []
for i, sub_task in enumerate(sub_tasks):
 worker_messages = [
 {'role': 'system', 'content': f'You are Worker Agent {i+1}. Complete the
following sub-task as best as you can.'},
 {'role': 'user', 'content': sub_task},
]
 worker_result = ask_mistral(worker_messages)
 print(f'\nWorker Agent {i+1} Result:\n', worker_result)
 worker_results.append(worker_result)

Manager aggregates results
aggregation_messages = [
 {'role': 'system', 'content': 'You are a manager agent. Aggregate the following
worker results into a final answer for the user.'},
 {'role': 'user', 'content': '\n'.join([f'Worker {i+1}: {r}' for i, r in
enumerate(worker_results)])},
]
final_result = ask_mistral(aggregation_messages)
print('\nFinal Aggregated Result for User:\n', final_result)

if __name__ == '__main__':
 main()

```

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `hierarchical-agent-pattern/` folder from this repository.

## Related Patterns

- [Task Delegation](#)
- [Parallel Agents](#)
- [Debate and Consensus](#)
- [Agent Loop](#)
- [Tool Use](#)
- [Pattern Evaluation Checklist](#)
- [Evaluation-Driven Agent Development](#)



# Debate and Consensus

Debate and consensus use multiple independent proposals, critiques, votes, or rankings before producing a final answer.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

Debate and consensus use multiple independent proposals, critiques, votes, or rankings before producing a final answer.

Use this pattern to expose disagreement, missing evidence, and weak reasoning before a risky answer becomes a final decision. Do not use it to make an answer look more trustworthy without adding independent evidence or a stronger review rule.

## Scenario

A team is reviewing whether an agent-generated incident summary is good enough for an executive update. One agent drafts the summary from traces. A second agent checks chronology against the incident timeline. A third agent checks customer impact and omitted evidence. A judge then accepts, rejects, or asks for a revised summary.

This works only if the agents see meaningfully different evidence or apply different rubrics. Three agents with the same prompt, same context, and same blind spot create a louder single-agent failure. Consensus is not evidence by itself; it is a way to expose disagreement before one accountable owner makes the final call.

## Use When

- A single answer is risky, ambiguous, or likely to miss evidence.
- Participants have genuinely different evidence, roles, rubrics, models, or tools.
- The final decision can wait for the extra cost and latency.
- A written merge rule exists before the workers start.
- Dissent should be preserved for review, not hidden by synthesis.

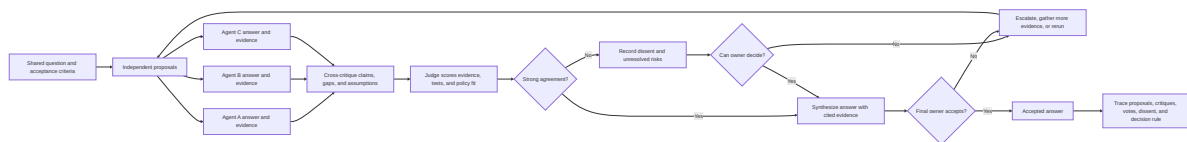
## Avoid When

- A deterministic test, database query, retrieval step, or policy check can answer directly.

- Participants share the same context and will repeat the same failure.
- Majority vote would replace evidence, tests, or an accountable owner.
- The task needs one clear owner more than it needs more opinions.
- The system cannot trace proposals, critiques, votes, dissent, and final acceptance.

## Architecture

Use this diagram to read Debate and Consensus as a system boundary, not only a code shape. The key ownership question is: the coordinator owns the shared goal, decomposition, assignments, merge policy, and final acceptance.



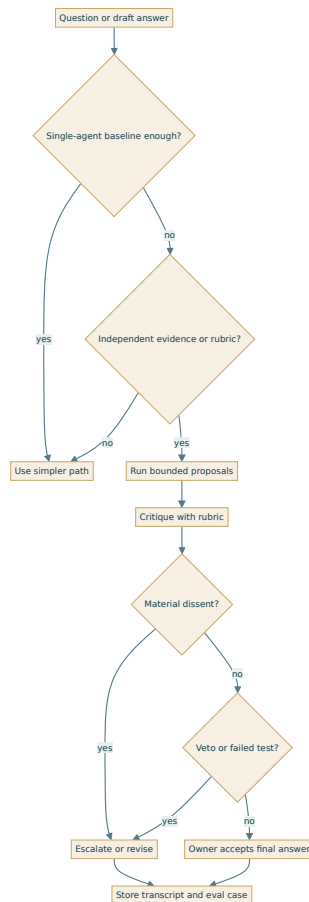
Read it as an evidence-backed decision process: independent agents propose answers, critique each other, and an accountable owner resolves agreement, dissent, or escalation.

## Decision Rules

Use debate only when independence is real and the merge rule is known before execution.

| Question                     | Good Answer                                                          | Bad Answer                                     |
|------------------------------|----------------------------------------------------------------------|------------------------------------------------|
| What differs between agents? | Evidence source, rubric, role, model, or tool access.                | Only the name of the role prompt.              |
| Who owns the final decision? | A coordinator, deterministic reducer, or human reviewer.             | The majority vote by itself.                   |
| What can overturn consensus? | Missing evidence, safety violation, failed test, or owner rejection. | Nothing; agreement is treated as truth.        |
| What is the cost limit?      | Fixed number of agents, turns, tokens, and retries.                  | Debate continues until outputs look confident. |
| How is dissent handled?      | Recorded, classified, and escalated when material.                   | Smoothed away during synthesis.                |

## Debate Gate Flow



## When Consensus Hurts Quality

| Situation                                | Why It Hurts                                         | Better Pattern                                               |
|------------------------------------------|------------------------------------------------------|--------------------------------------------------------------|
| Same prompt, same evidence, same model.  | Correlated failures look like agreement.             | Single agent plus evaluator, or retrieval/tool verification. |
| Weak majority overrules strong evidence. | Votes hide the reason a minority is correct.         | Evidence-weighted judge or human approval gate.              |
| Debate happens after synthesis.          | Critique cannot change the underlying evidence path. | Review proposals before final synthesis.                     |
| Workers optimize for persuasion.         | Outputs become rhetorical instead of testable.       | Score against a rubric with required citations.              |
| The owner is unnamed.                    | No one can accept residual risk.                     | Supervisor or human final-owner gate.                        |

## Voting Protocols

| Protocol      | Use When                                         | Guardrail                                            |
|---------------|--------------------------------------------------|------------------------------------------------------|
| Majority vote | Outputs are low risk and independently produced. | A safety or evidence veto can override the majority. |

| Protocol            | Use When                                                       | Guardrail                                                   |
|---------------------|----------------------------------------------------------------|-------------------------------------------------------------|
| Weighted rubric     | Roles have different authority or expertise.                   | Weights are fixed before the run starts.                    |
| Veto rule           | One class of defect should block release.                      | Veto reasons must cite evidence or policy.                  |
| Pairwise comparison | Several candidate answers compete.                             | Compare against the same rubric, not preference.            |
| Owner review        | The decision carries product, legal, safety, or customer risk. | The owner sees dissent and trace evidence before accepting. |

## System Shape

- **Pattern boundary:** a coordinator delegates bounded work to agents with narrow roles, then evaluates and merges their outputs.
- **State owner:** the coordinator owns the shared goal, decomposition, assignments, merge policy, and final acceptance.
- **Primary artifact:** `consensus-seeking-multi-agent-system-pattern/` contains deterministic TypeScript and Python reference implementations plus tests for proposal, critique, dissent, and final-owner behavior.
- **Operational promise:** debate exposes disagreement before a risky answer becomes a final decision.
- **Runnable path:** start with `npm run debate-consensus` before adapting the pattern to a larger system.

## Contract

A debate run should produce a transcript that another engineer can inspect.

| Field                                   | Purpose                                                              |
|-----------------------------------------|----------------------------------------------------------------------|
| <code>runId</code>                      | Correlates all proposals, critiques, votes, and final decision.      |
| <code>goal</code>                       | Defines the exact question being debated.                            |
| <code>agents[]</code>                   | Records role, evidence scope, model, tools, and permission limits.   |
| <code>proposal</code>                   | Captures each agent's answer with citations or evidence references.  |
| <code>critique</code>                   | Names specific defects, missing evidence, or risks.                  |
| <code>vote</code> or <code>score</code> | Applies a predeclared rubric, not an improvised preference.          |
| <code>dissent</code>                    | Preserves material disagreement for the owner.                       |
| <code>finalOwner</code>                 | Names the coordinator, reducer, or human accountable for acceptance. |

| Field                   | Purpose                                                                    |
|-------------------------|----------------------------------------------------------------------------|
| <code>stopReason</code> | Explains accepted, rejected, escalated, budget_exhausted, or inconclusive. |

## Core Protocol

1. Define the shared goal, worker roles, expected outputs, and acceptance criteria.
2. Split work only where independent or specialist execution adds value.
3. Dispatch tasks with scoped context and permissions.
4. Collect outputs, errors, refusals, and evidence from each worker.
5. Merge results through an explicit judge, reducer, supervisor, or human review gate.

## Implementation Notes

- Start with a single-agent baseline. Add debate only when measured quality improves enough to justify cost and latency.
- Give each participant a different evidence scope, rubric, model, or tool boundary.
- Keep proposals separate until critique begins. Early shared context can collapse independence.
- Require citations, trace IDs, test output, or source references for material claims.
- Preserve minority reports when they name missing evidence, policy risk, or unsupported claims.
- Cap agent count, turns, judge passes, retry count, total tokens, and wall-clock time.
- Treat a failed worker, missing evidence, or material dissent as a typed outcome, not as prose to smooth over.

## Failure Modes

- False agreement: agents share the same blind spot and all approve the wrong answer.
- Majority trap: two weak answers outvote one evidence-backed objection.
- Debate theater: agents critique style while ignoring source support, tests, or policy.
- Dissent erasure: final synthesis removes the disagreement the pattern was meant to reveal.
- Judge capture: the judge trusts confident prose instead of the predeclared rubric.
- Role collapse: all agents perform the same task despite different titles.
- Cost runaway: debate continues after the decision has become no clearer.
- Owner gap: the system produces a consensus with no one accountable for acceptance.

## Review Checklist

Before using debate or consensus in production, check:

- A single-agent baseline exists and debate beats it on measured tasks.
- Agents receive different evidence, roles, rubrics, models, or tools for a reason.
- The merge policy is written before the run starts.
- Dissent is preserved in the trace instead of erased by synthesis.
- The coordinator can refuse, escalate, or ask for more evidence.

- Cost and latency budgets cap agents, turns, retries, and judge passes.
- Evals include correlated failure, false agreement, bad majority vote, and judge error.

## Evaluation Strategy

- Compare multi-agent output against a single-agent baseline on the same tasks.
- Test cases where all agents agree and are wrong because they share evidence.
- Test cases where the minority answer is correct because it cites stronger evidence.
- Test worker failure, missing worker output, duplicated work, and bad merge decisions.
- Test veto behavior for safety, policy, missing evidence, and failed deterministic checks.
- Measure quality lift, latency cost, token cost, merge accuracy, dissent handling, and final-owner accountability.

## Production Checklist

- Give every worker a narrow role, evidence scope, permission set, and output schema.
- Make the merge policy explicit before workers run.
- Log per-worker inputs, outputs, critiques, scores, votes, and evidence references.
- Keep one owner for final acceptance and escalation.
- Preserve material dissent in the final trace and operator view.
- Add budget controls for agents, turns, retries, judge passes, tokens, and time.
- Define human escalation for high-risk, inconclusive, or policy-blocked work.
- Keep the source bundle, generated chapter, tests, and deployment artifact in the same release.

## Run the Example

```
npm run debate-consensus
npm run debate-consensus:test
```

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
consensus-seeking-multi-agent-system-pattern/typescript/src/consensus.ts
```

[Open full source](#)

```

export type Vote = "accept" | "revise" | "escalate";

export type StopReason = "accepted" | "needs_revision" | "escalated" | "blocked";

export type DebateEvidence = Record<string, string | undefined>;

export type DebateInput = {
 runId: string;
 goal: string;
 evidence: DebateEvidence;
 finalOwner: string;
 agents: DebateAgent[];
};

export type Proposal = {
 agentId: string;
 role: string;
 evidenceScope: string;
 answer: string;
 evidenceRefs: string[];
 vote: Vote;
 confidence: number;
 risks: string[];
};

export type Critique = {
 fromAgentId: string;
 targetAgentId: string;
 concerns: string[];
 material: boolean;
};

export type DebateAgent = {
 id: string;
 role: string;
 evidenceScope: string;
 weight: number;
 propose: (input: Omit<DebateInput, "agents">) => Omit<Proposal, "agentId" | "role" | "evidenceScope">;
};

export type DebateDecision = {
 stopReason: StopReason;
 finalOwner: string;

```

```

 accepted: boolean;
 summary: string;
 dissent: string[];
};

export type TranscriptEvent = {
 type: "proposal" | "critique" | "decision";
 agentId?: string;
 targetAgentId?: string;
 message: string;
 vote?: Vote;
 evidenceRefs?: string[];
};

export type DebateRun = {
 runId: string;
 goal: string;
 finalOwner: string;
 agents: Array<Pick<DebateAgent, "id" | "role" | "evidenceScope" | "weight">>;
 proposals: Proposal[];
 critiques: Critique[];
 decision: DebateDecision;
 transcript: TranscriptEvent[];
};

export type DebateEvaluation = {
 status: "pass" | "fail";
 reasons: string[];
};

export function createRubricAgent(config: {
 id: string;
 role: string;
 evidenceScope: string;
 requiredEvidence: string[];
 acceptedAnswer: string;
 weight?: number;
}): DebateAgent {
 return {
 id: config.id,
 role: config.role,
 evidenceScope: config.evidenceScope,
 weight: config.weight ?? 1,
 propose(input) {

```



```
const missing = config.requiredEvidence.filter(key => !input.evidence[key]);
const evidenceRefs = config.requiredEvidence.filter(key =>
Boolean(input.evidence[key]));
```

*Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.*

**consensus-seeking-multi-agent-system-pattern/typescript/test/consensus.spec.ts**

[Open full source](#)

```
import {
 createRubricAgent,
 evaluateDebate,
 incidentSummaryAgents,
 runDebate,
 type DebateRun,
} from "../src/consensus.ts";

function assert(condition: unknown, message: string): asserts condition {
 if (!condition) throw new Error(message);
}

const acceptedRun = runDebate({
 runId: "debate_accept",
 goal: "Approve incident summary",
 finalOwner: "incident-commander",
 evidence: {
 timeline: "incident timeline",
 customer_impact: "customer impact statement",
 mitigation: "mitigation record",
 owner: "follow-up owner",
 },
 agents: incidentSummaryAgents(),
});

assert(acceptedRun.decision.stopReason === "accepted", "Expected accepted decision");
assert(acceptedRun.decision.accepted, "Expected accepted flag");
assert(acceptedRun.proposals.length === 3, "Expected one proposal per agent");
assert(acceptedRun.transcript.some(event => event.type === "decision"), "Expected decision transcript event");
assert(evaluateDebate(acceptedRun).status === "pass", "Expected accepted run to pass evaluation");
```

```

const missingImpactRun = runDebate({
 runId: "debate_revision",
 goal: "Approve incident summary",
 finalOwner: "incident-commander",
 evidence: {
 timeline: "incident timeline",
 mitigation: "mitigation record",
 owner: "follow-up owner",
 },
 agents: incidentSummaryAgents(),
});

assert(missingImpactRun.decision.stopReason === "needs_revision", "Expected revision
decision");
assert(missingImpactRun.decision.dissent.some(item => item.includes("customer_impact")),
"Expected impact dissent");
assert(evaluateDebate(missingImpactRun).status === "pass", "Expected preserved dissent to
pass evaluation");

const correlatedAgent = createRubricAgent({
 id: "same_1",
 role: "same reviewer",
 evidenceScope: "same scope",
 requiredEvidence: ["timeline"],
 acceptedAnswer: "same answer",
});
const correlatedRun = runDebate({
 runId: "debate_correlated",
 goal: "Approve incident summary",
 finalOwner: "incident-commander",
 evidence: { timeline: "incident timeline" },
 agents: [
 correlatedAgent,
 { ...correlatedAgent, id: "same_2" },
],
});
const correlatedEval = evaluateDebate(correlatedRun);
assert(correlatedEval.status === "fail", "Expected correlated agents to fail");
assert(correlatedEval.reasons.includes("agents are not independent"), "Expected
independence reason");

const missingOwner: DebateRun = {
 ...acceptedRun,
 finalOwner: "",

```

```

decision: {
 ...acceptedRun.decision,
 finalOwner: "",
},
};

const missingOwnerEval = evaluateDebate(missingOwner);
assert(missingOwnerEval.status === "fail", "Expected missing final owner to fail");
assert(missingOwnerEval.reasons.includes("missing final owner"), "Expected final-owner reason");

console.log("Debate and consensus tests OK");

```

[consensus-seeking-multi-agent-system-pattern/python/consensus.py](#)

[Open full source](#)

```

from dataclasses import dataclass
from typing import Literal

Vote = Literal["accept", "revise", "escalate"]
StopReason = Literal["accepted", "needs_revision", "escalated", "blocked"]

@dataclass(frozen=True)
class Proposal:
 agent_id: str
 role: str
 evidence_scope: str
 answer: str
 evidence_refs: list[str]
 vote: Vote
 confidence: float
 risks: list[str]

@dataclass(frozen=True)
class Critique:
 from_agent_id: str
 target_agent_id: str
 concerns: list[str]
 material: bool

@dataclass(frozen=True)
class DebateAgent:
 id: str
 role: str

```

```

evidence_scope: str
required_evidence: list[str]
accepted_answer: str
weight: float = 1.0

def propose(self, evidence: dict[str, str]) -> Proposal:
 missing = [key for key in self.required_evidence if not evidence.get(key)]
 evidence_refs = [key for key in self.required_evidence if evidence.get(key)]

 if missing:
 return Proposal(
 agent_id=self.id,
 role=self.role,
 evidence_scope=self.evidence_scope,
 answer=f"{self.role} cannot accept until evidence is added: {'',
'.join(missing)}.",
 evidence_refs=evidence_refs,
 vote="revise",
 confidence=0.45,
 risks=[f"missing evidence: {key}" for key in missing],
)

 return Proposal(
 agent_id=self.id,
 role=self.role,
 evidence_scope=self.evidence_scope,
 answer=self.accepted_answer,
 evidence_refs=evidence_refs,
 vote="accept",
 confidence=0.9,
 risks=[],
)

def incident_summary_agents() -> list[DebateAgent]:
 return [
 DebateAgent(
 id="chronology",
 role="Chronology reviewer",
 evidence_scope="incident_timeline",
 required_evidence=["timeline"],
 accepted_answer="Timeline is supported by the incident trace.",
),
 DebateAgent(
 id="impact",

```

```

 role="Impact reviewer",
 evidence_scope="customer_impact",
 required_evidence=["customer_impact"],
 accepted_answer="Customer impact is explicit and bounded.",
),
 DebateAgent(
 id="safety",
 role="Safety reviewer",
 evidence_scope="mitigation_and_followup",
 required_evidence=["mitigation", "owner"],
 accepted_answer="Mitigation, owner, and follow-up are recorded.",
),
]

```

*Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.*

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `consensus-seeking-multi-agent-system-pattern/` folder from this repository.

## Related Patterns

- [Task Delegation](#)
- [Supervisor / Worker](#)
- [Parallel Agents](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# Parallel Agents

Parallel agents run independent work concurrently, then merge results through a fan-out/fan-in control point.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

Parallel agents run independent work concurrently, then merge results through a fan-out/fan-in control point.

## Use When

- Work can be split into independent searches, reviews, or candidate generations.
- Latency matters and parallelism is safe.
- The merge step can compare, rank, or synthesize results.

## Avoid When

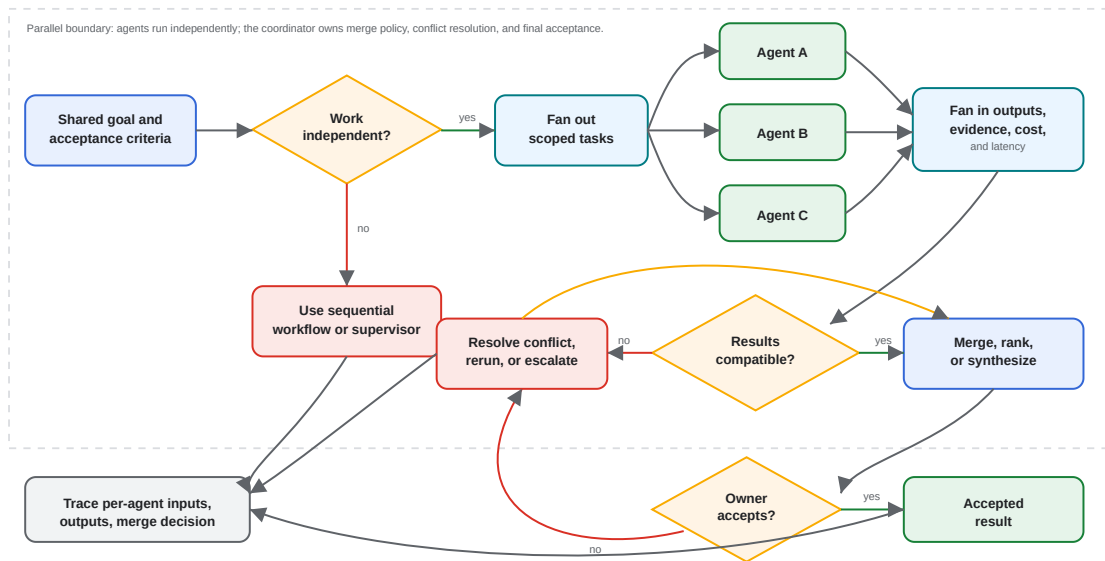
- Agents need shared mutable state during execution.
- The merge policy is vague.
- Parallel work increases cost without increasing quality.

## Architecture

Use this diagram to read Parallel Agents as a system boundary, not only a code shape. The key ownership question is: the coordinator owns the shared goal, decomposition, assignments, merge policy, and final acceptance.

## Parallel Agents Fan-Out/Fan-In

Use parallelism only when work is independent and the merge policy can judge the results.



Read it as a fan-out/fan-in boundary: parallelism is useful only when workers are independent and the coordinator can compare, merge, or reject their outputs.

## System Shape

- **Pattern boundary:** a coordinator delegates bounded work to agents with narrow roles, then evaluates and merges their outputs.
- **State owner:** the coordinator owns the shared goal, decomposition, assignments, merge policy, and final acceptance.
- **Primary artifact:** `multi-agent-collaboration-pattern/` contains the runnable reference implementation and examples.
- **Operational promise:** Parallel agents run independent work concurrently, then merge results through a fan-out/fan-in control point.
- **Runnable path:** start with `npm run multi-agent-collab` before adapting the pattern to a larger system.

## Core Protocol

1. Define the shared goal, worker roles, expected outputs, and acceptance criteria.
2. Split work only where independent or specialist execution adds value.
3. Dispatch tasks with scoped context and permissions.
4. Collect outputs, errors, refusals, and evidence from each worker.
5. Merge results through an explicit judge, reducer, supervisor, or human review gate.

## Implementation Notes

- Keep the pattern boundary explicit: inputs, state, side effects, and outputs should be visible.
- Validate model-produced decisions before they affect tools, users, or durable state.

- Emit enough trace data to debug failures after the run.

## Failure Modes

- The pattern is applied where a simpler deterministic workflow would be better.
- State, tool calls, or model decisions are not observable enough to debug.
- The system lacks clear stop, retry, or escalation behavior.

## Evaluation Strategy

- Compare multi-agent output against a single-agent baseline on the same tasks.
- Test worker disagreement, worker failure, duplicated work, and bad merge decisions.
- Measure quality lift, latency cost, token cost, merge accuracy, and accountability.
- Include cases that prove each “Use When” condition is true for this pattern.
- Include negative cases from “Avoid When” so the system chooses a simpler or safer pattern when appropriate.

## Production Checklist

- Give every worker a narrow contract and permission set.
- Make the merge policy explicit before workers run.
- Log per-worker inputs, outputs, and decision evidence.
- Keep one owner for final acceptance and escalation.
- Define human escalation for ambiguous, high-risk, or policy-blocked work.
- Keep the source bundle, generated chapter, tests, and deployment artifact in the same release.

## Run the Example

```
npm run multi-agent-collab
```

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
multi-agent-collaboration-pattern/autogen_typescript_example/multi_agent_collab.ts
```

[Open full source](#)



```

// Multi-Agent Collaboration Pattern - Autogen TypeScript Example
// To run: npm install && npm run multi-agent-collab

import axios from 'axios';
import * as readline from 'readline';
import * as dotenv from 'dotenv';
dotenv.config();

const MISTRAL_API_URL = 'https://api.mistral.ai/v1/chat/completions';
const MISTRAL_API_KEY = process.env.MISTRAL_API_KEY;

async function agent(name: string, role: string, input: string): Promise<string> {
 const prompt = `You are ${name}, your role is: ${role}. Here is your input: ${input}`;
 const response = await axios.post(
 MISTRAL_API_URL,
 {
 model: 'mistral-tiny',
 messages: [{ role: 'user', content: prompt }],
 },
 {
 headers: {
 'Authorization': `Bearer ${MISTRAL_API_KEY}`,
 'Content-Type': 'application/json',
 },
 }
);
 return response.data.choices[0].message.content;
}

async function multiAgentCollab(task: string) {
 // Agent 1: Idea Generator
 const idea = await agent('Alice', 'Idea Generator', task);
 console.log('Alice (Idea Generator):', idea);

 // Agent 2: Critic
 const critique = await agent('Bob', 'Critic', `Here is an idea: ${idea}\nPlease critique or improve it.`);
 console.log('Bob (Critic):', critique);

 // Agent 1: Finalize
 const final = await agent('Alice', 'Idea Generator', `Here is the critique: ${critique}\nPlease finalize the solution.`);
 console.log('Alice (Finalized):', final);
}

```

```
const rl = readline.createInterface({
 input: process.stdin,
 output: process.stdout
});

rl.question('Task: ', async (userInput: string) => {
 try {
 await multiAgentCollab(userInput);
 } catch (err) {
 console.error('Error:', err);
 }
 rl.close();
});
```

[multi-agent-collaboration-pattern/langgraph\\_python\\_example/multi\\_agent\\_collab.py](#)

[Open full source](#)

```
Multi-Agent Collaboration Pattern - LangGraph Python Example
```

This example demonstrates the Multi-Agent Collaboration Pattern using LangGraph and Python. Two agents (an Idea Generator and a Critic) collaborate to solve a task, exchanging messages and refining the solution. The LLM is Mistral.

```
Requirements
```

- Python 3.8+
- `langgraph` library
- `python-dotenv` (for .env support)
- Mistral LLM API access

```
Install dependencies
```

```
```bash
pip install langgraph python-dotenv requests
```
```

```
Example Code
```

```
```python
import os

from langgraph import Agent, Environment, LLM
from dotenv import load_dotenv
```

```

load_dotenv()

MISTRAL_API_KEY = os.getenv("MISTRAL_API_KEY")
MISTRAL_API_URL = "https://api.mistral.ai/v1/chat/completions"

class SimpleEnvironment(Environment):
    def get_observation(self):
        return input("Task: ")
    def send_action(self, action):
        print(action)

class IdeaGenerator(Agent):
    def __init__(self, llm):
        self.llm = llm
    def act(self, observation):
        prompt = f"You are Alice, an Idea Generator. Here is your task: {observation}"
        return self.llm.complete(prompt)

class Critic(Agent):
    def __init__(self, llm):
        self.llm = llm
    def act(self, idea):
        prompt = f"You are Bob, a Critic. Here is an idea: {idea}\nPlease critique or improve it."
        return self.llm.complete(prompt)

llm = LLM(
    provider="mistral",
    api_key=MISTRAL_API_KEY,
    api_url=MISTRAL_API_URL,
)

env = SimpleEnvironment()
idea_agent = IdeaGenerator(llm)
critic_agent = Critic(llm)

task = env.get_observation()
idea = idea_agent.act(task)
print("Alice (Idea Generator):", idea)
critique = critic_agent.act(idea)
print("Bob (Critic):", critique)
final = idea_agent.act(f"Here is the critique: {critique}\nPlease finalize the solution.")
print("Alice (Finalized):", final)

```

- ```
'''
'''
'''
```
- Try a creative or open-ended task to see agent collaboration.
  - Make sure your ``.env`` file contains your Mistral API key.

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `multi-agent-collaboration-pattern/` folder from this repository.

## Related Patterns

- [Task Delegation](#)
- [Supervisor / Worker](#)
- [Debate and Consensus](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# CrewAI Flows and Crews

CrewAI Flows own state and execution order. Crews group specialized agents that collaborate on delegated work inside the flow.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

The CrewAI Flows and Crews Pattern separates production workflow control from collaborative agent work. Flows own state and execution order; crews group specialized agents that perform delegated tasks inside the flow.

## Use When

- You are building Python-first agent automations.
- The system needs explicit state and event-driven control flow.
- Multiple specialist agents need to collaborate on bounded tasks.

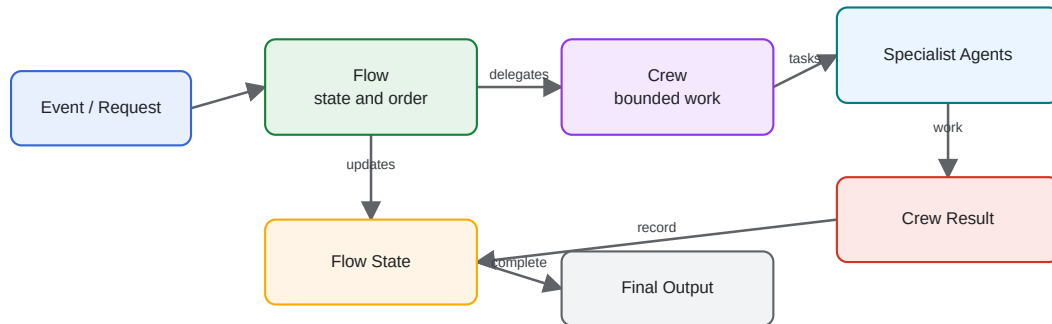
## Avoid When

- A single deterministic workflow step is enough.
- Agents have unclear roles or overlapping responsibilities.
- You cannot define where flow state begins and crew-local context ends.

## Architecture

Use this diagram to read CrewAI Flows and Crews as a system boundary, not only a code shape. The key ownership question is: the coordinator owns the shared goal, decomposition, assignments, merge policy, and final acceptance.

## CrewAI Flows and Crews



## System Shape

- **Flow boundary:** the Flow owns durable state, ordering, branching, checkpoints, acceptance, and final output.
- **Crew boundary:** a Crew performs bounded specialist work inside a Flow step and returns structured outputs.
- **Agent boundary:** each agent has a role, goal, tools, permissions, and expected output shape that differ from the other roles.
- **Policy boundary:** the Flow checks authority before crew kickoff, tool use, memory writes, and final acceptance.
- **Evaluation boundary:** flow state transitions and crew outputs are tested separately, then tested together as one trajectory.
- **Operational boundary:** traces record flow events, crew kickoff, role outputs, validation, acceptance, rejection, and escalation.

## Core Protocol

1. Accept an event or request with actor, tenant, goal, release version, and idempotency key.
2. Initialize Flow state and decide whether the work needs a Crew or a deterministic function.
3. Create tasks with scoped inputs, expected outputs, allowed tools, and acceptance criteria.
4. Run the Crew and collect role outputs, errors, refusals, and evidence.
5. Validate each role output before it can mutate Flow state.
6. Let the Flow accept, reject, retry, escalate, or request human review.
7. Emit trace events for flow steps, crew kickoff, role outputs, policy decisions, and final acceptance.
8. Convert rejected outputs, role disagreements, and incidents into eval fixtures.

## Implementation Notes

- Let flows manage state, branching, persistence, and execution order.
- Give each crew a bounded task with clear expected output.
- Give each agent a role that changes behavior, not just a different name.

- Test flow state transitions separately from crew output quality.
- Prefer deterministic Flow logic for ordering, retry, checkpointing, approval, and rollback.
- Keep Crew-local conversation from becoming the only source of truth for workflow state.
- Validate role outputs with schemas or explicit acceptance functions before using them.
- Record why the Flow accepted or rejected the Crew result.

## Failure Modes

- Crews used as a substitute for workflow design.
- Too many agents with vague roles.
- Flow state mutated implicitly through chat history.
- No evaluator for whether the crew result satisfies the flow step.
- Role outputs accepted without schema, evidence, or policy checks.
- Crew failure hidden as a weak final answer instead of a typed failed state.
- Human escalation missing for ambiguous, high-risk, or conflicting outputs.

## Evaluation Strategy

- Test Flow transitions with deterministic fixtures before involving Crew behavior.
- Test each role's expected output shape, tool permissions, and refusal behavior.
- Test worker disagreement, missing evidence, tool timeout, and rejected Crew output.
- Compare Crew output against a single-agent or deterministic baseline to prove the Crew adds value.
- Gate releases on final answer quality and trajectory quality: role behavior, policy decisions, and Flow acceptance.

## Production Checklist

- Document install, local run, test, eval, and cleanup commands.
- Define Flow state, checkpoint strategy, role permissions, and task schemas.
- Validate Crew outputs before they modify Flow state or produce user-visible output.
- Export redacted flow, task, role, policy, and evaluator traces.
- Add evals for accepted output, rejected output, role disagreement, tool failure, and escalation.
- Define rollback for disabling one role, one tool, one Flow path, or the whole Crew route.

## Run the Example

```
npm run crewai-flow
npm run crewai-flow:test
```

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

`crewai-flows-and-crews-pattern/python/flow_crew.py`

[Open full source](#)

```
from dataclasses import dataclass, field
from typing import Callable

@dataclass
class Agent:
 role: str
 goal: str
 run: Callable[[str], str]

@dataclass
class Task:
 name: str
 agent_role: str
 input: str

@dataclass
class Crew:
 name: str
 agents: dict[str, Agent]

 def kickoff(self, tasks: list[Task]) -> dict[str, str]:
 outputs: dict[str, str] = {}
 for task in tasks:
 agent = self.agents[task.agent_role]
 outputs[task.name] = agent.run(task.input)
 return outputs

@dataclass
class FlowState:
 goal: str
 accepted: bool = False
```



```

crew_outputs: dict[str, str] = field(default_factory=dict)
trace: list[str] = field(default_factory=list)
final: str | None = None

def build_research_crew() -> Crew:
 return Crew(
 name="support_research_crew",
 agents={
 "researcher": Agent(
 role="researcher",
 goal="Find policy facts relevant to the task.",
 run=lambda task_input: f"policy evidence for {task_input}: refund window is
30 days",
),
 "writer": Agent(
 role="writer",
 goal="Turn evidence into a concise draft.",
 run=lambda task_input: f"draft based on {task_input}: offer review, do not
promise payment",
),
 },
)

def run_support_flow(goal: str, crew: Crew | None = None) -> FlowState:
 state = FlowState(goal=goal)
 active_crew = crew or build_research_crew()

 state.trace.append("flow:start")
 tasks = [
 Task(name="evidence", agent_role="researcher", input=goal),
 Task(name="draft", agent_role="writer", input="policy evidence"),
]
 state.trace.append("flow:crew_kickoff")
 state.crew_outputs = active_crew.kickoff(tasks)

 evidence = state.crew_outputs["evidence"]
 draft = state.crew_outputs["draft"]
 state.trace.append("flow:evaluate")

 state.accepted = "30 days" in evidence and "do not promise payment" in draft
 if state.accepted:
 state.final = "Crew output accepted by the flow."
 state.trace.append("flow:accepted")
 else:

```

```

 state.final = "Crew output rejected by the flow."
 state.trace.append("flow:rejected")

 return state

def evaluate_flow(state: FlowState) -> dict[str, object]:
 reasons: list[str] = []
 if not state.accepted:
 reasons.append("flow did not accept crew output")
 if "evidence" not in state.crew_outputs:

```

*Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.*

[crewai-flows-and-crews-pattern/python/test\\_flow\\_crew.py](#)

[Open full source](#)

```

from flow_crew import Agent, Crew, evaluate_flow, run_support_flow

def assert_true(condition, message):
 if not condition:
 raise AssertionError(message)

state = run_support_flow("Prepare a refund response")
evaluation = evaluate_flow(state)

assert_true(state.accepted, "Expected flow to accept crew output")
assert_true(state.crew_outputs["evidence"].startswith("policy evidence"), "Expected researcher output")
assert_true("do not promise payment" in state.crew_outputs["draft"], "Expected constrained writer output")
assert_true(state.trace == ["flow:start", "flow:crew_kickoff", "flow:evaluate", "flow:accepted"], "Expected deterministic flow trace")
assert_true(evaluation["status"] == "pass", "Expected flow evaluation to pass")

bad_crew = Crew(
 name="unsafe_support_research_crew",
 agents={
 "researcher": Agent(
 role="researcher",
 goal="Find policy facts relevant to the task.",
 run=lambda task_input: f"policy evidence for {task_input}: refund window is 30 days",

```

```

),
 "writer": Agent(
 role="writer",
 goal="Turn evidence into a concise draft.",
 run=lambda task_input: f"draft based on {task_input}: promise payment now",
),
},
)

rejected_state = run_support_flow("Prepare a refund response", crew=bad_crew)
rejected_evaluation = evaluate_flow(rejected_state)

assert_true(not rejected_state.accepted, "Expected flow to reject unsafe writer output")
assert_true(rejected_state.final == "Crew output rejected by the flow.", "Expected rejected
final state")
assert_true(rejected_state.trace[-1] == "flow:rejected", "Expected rejection trace")
assert_true(rejected_evaluation["status"] == "fail", "Expected rejected flow evaluation to
fail")
assert_true(
 "flow did not accept crew output" in rejected_evaluation["reasons"],
 "Expected rejection reason",
)

print("CrewAI-style flow and crew tests OK")

```

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `crewai-flows-and-crews-pattern/` folder from this repository.

## Related Patterns

- [Task Delegation](#)
- [Consensus-Seeking Multi-Agent System](#)
- [Durable Workflow](#)

# Agentic System Architecture

An agentic system is more than a model call wrapped in a loop. It is a set of control planes, execution planes, data planes, and safety planes that let software use model judgment without losing engineering control.

Use this chapter when a single pattern is not enough and you need to combine agents, tools, memory, policies, workflows, evals, and observability into one coherent system.

This chapter owns composition and responsibility boundaries. It does not prescribe one framework or reimplement every plane. The specialized chapters own loop control, tools, memory, workflows, security, evaluation, and runtime operation.

When agents cross process, team, runtime, or ownership boundaries, treat them like services with explicit contracts. See [Agents As Services](#) for that architecture.

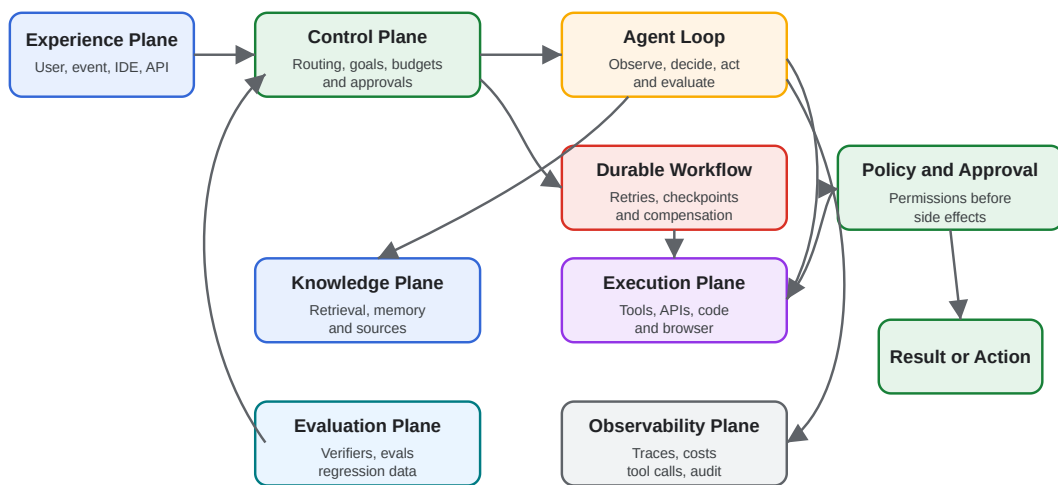
Download the [agentic system architecture review checklist](#) when turning this chapter into an ADR, design review, or capstone review.

## Core Idea

Separate the system into planes:

- **Experience plane:** chat, IDE, API, webhook, ticket, mobile, or scheduled endpoint.
- **Control plane:** routing, planning, permissions, budgets, approvals, task state, and stop conditions.
- **Execution plane:** tools, code execution, browser actions, API calls, workflows, and external side effects.
- **Knowledge plane:** retrieval, memory, indexes, metadata, source freshness, and citations.
- **Evaluation plane:** offline evals, runtime checks, verifiers, red-team tests, and regression datasets.
- **Observability plane:** traces, costs, latency, tool calls, model inputs, decisions, and operator review.

## Agentic System Architecture



The planes should have named owners. If the same prompt owns planning, policy, memory writes, tool authority, retry behavior, and user messaging, the system has no real architecture. It has a model with broad authority.

## Plane Ownership Matrix

Use this table during design review.

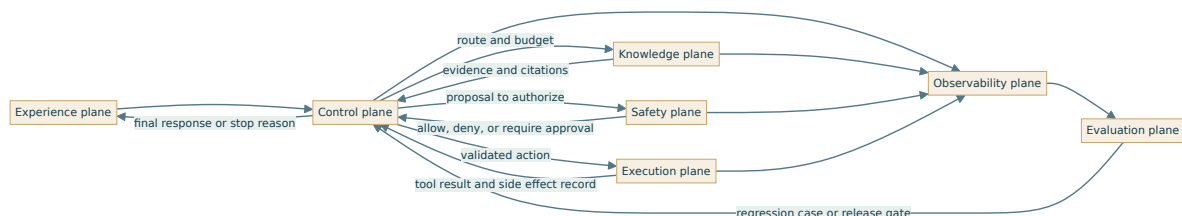
| Plane      | Primary Owner                                    | What It Must Decide                                                | Evidence                                        |
|------------|--------------------------------------------------|--------------------------------------------------------------------|-------------------------------------------------|
| Experience | product surface or API gateway                   | who starts the run, what the user sees, what requires confirmation | UI flow, API contract, approval copy            |
| Control    | runtime, workflow engine, or application service | goal state, routing, budgets, stop conditions, retries, escalation | state machine, workflow graph, timeout policy   |
| Execution  | tool gateway, service adapters, sandbox          | which actions can run and with what authority                      | tool manifest, permission map, idempotency keys |
| Knowledge  | retrieval service, memory service, data platform | what evidence can enter context and what can persist               | index policy, memory policy, citation records   |
| Evaluation | test harness and release gate                    | what quality bar blocks release                                    | eval fixtures, thresholds, CI output            |

| Plane         | Primary Owner                          | What It Must Decide                             | Evidence                                   |
|---------------|----------------------------------------|-------------------------------------------------|--------------------------------------------|
| Observability | platform and operations team           | how a failed run is reconstructed               | trace schema, dashboards, redaction proof  |
| Safety        | policy service and human approval path | which actions are denied, allowed, or escalated | policy rules, approval records, audit logs |

This matrix prevents vague ownership. A chapter, framework, or team can own a plane, but the production system must name the owner.

## Runtime Plane Flow

The planes interact on every serious run. The experience plane starts the work, the control plane decides the route, the safety plane grants or denies authority, and the execution and knowledge planes return evidence. Evaluation and observability wrap the whole path.



Use the flow to find missing contracts. If a run can move from control to execution without safety, the system has hidden authority. If execution returns a side effect without observability, the incident review will be guesswork. If evaluation never receives traces, production failures will not become release gates.

## Architecture Questions

- What owns the goal?
- What owns state?
- What may call tools?
- What requires approval?
- What happens when evidence is missing?
- What happens when the model is wrong?
- What can be replayed after a failure?
- What is deterministic, and what is model-mediated?

The system should have direct answers to those questions before it handles private data, money movement, production infrastructure, or customer-facing communication.

Add these questions for production review:

- Which component can stop a run?
- Which component can resume a run?
- Which component can write memory?

- Which component can mutate user-visible records?
- Which component can spend money, send messages, deploy code, or change permissions?
- Which component proves that a policy check happened before authority was used?
- Which component turns incidents into eval cases?

If the answer is “the model decides,” narrow the model’s role. Let it propose. Let software validate, authorize, execute, and record.

## Boundary Design

The most important architectural choice is the boundary between model judgment and deterministic software. Keep model outputs as proposals until software validates them.

Strong boundaries look like typed tool schemas, policy checks before side effects, explicit state transitions, human approval for high-risk operations, retrieval filters and citations, budget and timeout limits, and audit logs that connect the prompt, the decision, the tool input, and the result. Weak boundaries look like broad shell or browser access without approval, prompt-only policy enforcement, unstructured memory writes, hidden retries, tool results that cannot be traced, and agents that can rewrite their own operating rules without review.

## Composition Contract

Every serious agentic system needs a written composition contract. It can live in an ADR, service contract, or runtime config, but it should answer the same questions.

```
type AgenticSystemContract = {
 systemId: string;
 owner: string;
 userEntrypoints: string[];
 supportedGoals: string[];
 stateOwner: "application" | "workflow-engine" | "agent-runtime" | "external-service";
 toolAuthority: Array<{
 toolName: string;
 capability: "read" | "draft" | "write" | "execute";
 approval: "none" | "human" | "policy" | "human-and-policy";
 idempotencyRequired: boolean;
 }>;
 knowledgeSources: Array<{
 source: string;
 freshnessRule: string;
 accessRule: string;
 citationRequired: boolean;
 }>;
 memoryPolicy: {
 canWrite: boolean;
```

```

 writeOwner: string;
 retention: string;
 deletionPath: string;
};
evalGate: {
 dataset: string;
 blockingThresholds: string[];
 requiredBeforeRelease: boolean;
};
observability: {
 traceId: string;
 redactionPolicy: string;
 replaySupported: boolean;
};
rollback: {
 disableModel: string;
 disableTool: string;
 disableWorkflow: string;
};
};
};

```

The exact type matters less than the discipline. A reviewer should be able to read the contract and know what the system can do, who owns each boundary, and how the team proves it worked.

## Build Sequence

Build the system in thin vertical slices.

1. Start with the user-visible workflow and define the smallest valuable goal.
2. Add deterministic state before adding memory or multi-agent behavior.
3. Add one read-only tool and trace every call.
4. Add retrieval only after access rules and citation requirements are clear.
5. Add one write-capable tool behind policy, approval, and idempotency.
6. Add eval fixtures for successful runs, refusal cases, missing evidence, policy denial, tool failure, and rollback.
7. Add retries and recovery only after the trace can explain the first failure.
8. Add multi-agent topology only when a single agent creates unclear ownership or poor quality.

This order keeps the system inspectable. It also gives the team a working product slice before the architecture grows.

## Example: Support Refund Agent

A support refund agent may look simple: read a ticket, inspect orders, decide whether a refund is allowed, and draft a response. In production, that workflow crosses several planes.



| Step                     | Plane         | Owner               | Control                                    |
|--------------------------|---------------|---------------------|--------------------------------------------|
| Customer asks for refund | experience    | support UI          | authenticated user and ticket context      |
| Runtime creates goal     | control       | workflow service    | run ID, tenant ID, timeout, stop condition |
| Agent reads policy       | knowledge     | retrieval service   | tenant-scoped policy index with citations  |
| Agent reads order        | execution     | order API adapter   | read-only tool with trace record           |
| Agent proposes refund    | control       | application service | structured decision object                 |
| Policy checks proposal   | safety        | policy service      | amount, reason, region, customer status    |
| Human approves edge case | safety        | support supervisor  | approval record with expiry                |
| Refund executes          | execution     | payment adapter     | idempotency key and rollback note          |
| Run closes               | observability | platform            | trace, cost, tool calls, policy result     |
| Failed case enters evals | evaluation    | engineering team    | regression fixture and release gate        |

This architecture does not trust the model with money movement. The model reads evidence and proposes an action. Software checks the proposal, records the decision, and owns the side effect.

## Before And After Architecture Review

A weak design often sounds concise:

Give the agent the customer ticket, policy docs, order API, refund API, and email tool.  
Ask it to solve the case end to end.

That sketch hides the architecture inside the prompt. Reviewers cannot see who owns tenant scope, source freshness, refund authority, approval, retry safety, memory, or incident replay.

Use this review table to convert the sketch into a system.

| Weak Design Choice             | Hidden Risk                                                | Production Rewrite                                                                |
|--------------------------------|------------------------------------------------------------|-----------------------------------------------------------------------------------|
| One agent receives every tool. | Read authority and write authority collapse into one role. | Split read tools, draft tools, and side-effect tools behind separate permissions. |

| Weak Design Choice                           | Hidden Risk                                                  | Production Rewrite                                                                                        |
|----------------------------------------------|--------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| Policy docs are just prompt context.         | Stale or wrong policy can drive a customer-impacting action. | Retrieve policy through an access-controlled source with version, owner, and citation.                    |
| The model decides whether refund is allowed. | Business authority sits in generated text.                   | Model proposes a recommendation; policy service validates threshold, region, status, and exception rules. |
| Refund call happens inside the loop.         | Retry or crash can duplicate money movement.                 | Payment adapter requires approval record, idempotency key, and side-effect trace.                         |
| Email is sent after the final answer.        | Customer sees unsupported or unapproved language.            | Model drafts email; runtime checks evidence, tone, policy, and approval state before send.                |
| Memory stores the outcome automatically.     | Bad or sensitive memory can persist across sessions.         | Memory write is classified, reviewed by policy, and tied to retention and deletion rules.                 |
| Logs are plain text.                         | Operators cannot reconstruct why the action happened.        | Trace links goal, evidence, model proposal, policy decision, approval, tool call, and stop reason.        |

The safer design is still agentic. The model handles ambiguity, synthesis, and recommendation. The architecture owns authority, evidence, state, and recovery.

## Release Evidence Bundle

For an architecture review, require artifacts rather than assurances.

| Artifact             | Minimum Content                                                                                     | Reviewer Question                                 |
|----------------------|-----------------------------------------------------------------------------------------------------|---------------------------------------------------|
| Plane ownership map  | owners for experience, control, execution, knowledge, evaluation, observability, and safety         | Who gets paged when this boundary fails?          |
| Composition contract | supported goals, state owner, tool authority, knowledge sources, memory policy, eval gate, rollback | Can the model do anything outside this contract?  |
| Happy-path trace     | one successful run from entrypoint to final status                                                  | Can we prove the intended path works?             |
| Denied-action trace  | one blocked tool, retrieval, memory, or approval attempt                                            | Can we prove authority is enforced before action? |

| Artifact               | Minimum Content                                   | Reviewer Question                                |
|------------------------|---------------------------------------------------|--------------------------------------------------|
| Missing-evidence trace | one run that refuses, asks, or escalates          | Can we prove the system does not invent support? |
| Rollback drill         | feature flag, disable switch, or routing fallback | Can operators stop the risky behavior quickly?   |
| Regression gate        | eval cases tied to the known risks                | Will the next release catch the same failure?    |

If a team cannot attach these artifacts, the design is not ready for broad production use. It may still be a prototype or internal pilot, but its release label should say so.

## Framework Mapping

Frameworks can help, but they do not remove the need for boundaries.

| Architecture Need   | Common Framework Support                      | What The Application Still Owns                   |
|---------------------|-----------------------------------------------|---------------------------------------------------|
| state graph         | LangGraph, workflow engines, durable runtimes | domain state schema and migration rules           |
| agent collaboration | AutoGen, CrewAI, custom service topology      | role boundaries and final decision authority      |
| tool discovery      | MCP, tool registries, framework tool APIs     | permissions, secrets, policy, and idempotency     |
| retrieval           | vector stores, RAG frameworks, search APIs    | source access, freshness, citations, deletion     |
| memory              | framework memory stores or custom stores      | write policy, correction, retention, user control |
| evals               | test harnesses, judges, CI scripts            | datasets, thresholds, release rules               |
| observability       | tracing SDKs and platform logs                | redaction, run reconstruction, incident workflow  |

Use framework features where they sharpen the boundary. Do not let framework convenience hide policy, ownership, or state.

## Composition Patterns

Common production systems combine several chapters from this book:

- **Agent Loop** for bounded observe-decide-act cycles.

- **Goals and State** for resumability and auditability.
- **MCP-first Tool Use** for discoverable tools.
- **Agentic RAG Systems** for evidence-grounded answers.
- **Production Runtime Overview** for the control plane around state, policy, budgets, traces, evals, and stop decisions.
- **Durable Workflows** for long-running state, retries, and approvals.
- **Observability and Evals** for quality gates and regression control.
- **Policy Enforcement** for permission and compliance checks.
- **Agents As Services** for service boundaries, contracts, protocols, retries, and trace correlation between agents.

## Architecture Review Checklist

A design is ready to build when reviewers can answer yes to these items:

- The system has one named owner and one incident owner.
- Each plane has a named owner and evidence artifact.
- The model proposes actions through structured outputs.
- Deterministic code validates structured outputs before side effects.
- Read tools, write tools, and execution tools have separate permissions.
- High-risk actions require policy, approval, or both.
- Retrieval sources have access rules, freshness rules, and citation rules.
- Memory writes have a policy, retention period, and deletion path.
- State can be inspected after failure.
- A run can be correlated across prompts, tool calls, policy checks, approvals, and side effects.
- Evals cover success, refusal, missing evidence, policy denial, tool failure, replay, and rollback.
- Rollback can disable the model, a tool, a workflow, or an agent without redeploying the whole system.

## Failure Modes

The recurring failures are easy to name. The model becomes the control plane, and no deterministic component owns state or permissions. Every pattern gets added at once, producing a system that is powerful but impossible to debug. Retrieval, memory, and tool output are mixed into one untrusted context blob. The system has no replay path after a bad action. Evals show up only after the first production failure, instead of before launch.

Other failures show up later. A team lets memory become a hidden database with no correction path. Tool adapters accept broad natural-language instructions instead of typed inputs. Retry logic repeats side effects without idempotency keys. A framework owns state in a way the application cannot inspect. A successful demo becomes a production release without threat modeling, redaction, or rollback.

## Design Review Questions

Ask these questions before implementation:

- Which business decision does this system make or support?
- What is the narrowest useful autonomy level?
- What evidence must be present before the model can answer?
- What action is too risky for autonomous execution?
- Which tool result can poison later reasoning if accepted blindly?
- Which state transition must never depend only on generated text?
- Which trace would an operator need during a production incident?
- Which eval would have caught the most likely embarrassing failure?
- Which feature flag disables the risky part first?
- What future chapter or ADR should own the next expansion?

## Design Rule

Architecture should make failure visible. If a run fails, an operator should be able to answer: what goal was active, what evidence was used, what tool calls ran, what policy checks passed, what changed, and why the system stopped.

Use [Reference Architecture](#) to turn these planes into a concrete deployment shape. Then use [Production Runtime Overview](#) to define the controls that execute and operate each run.

## Related Chapters

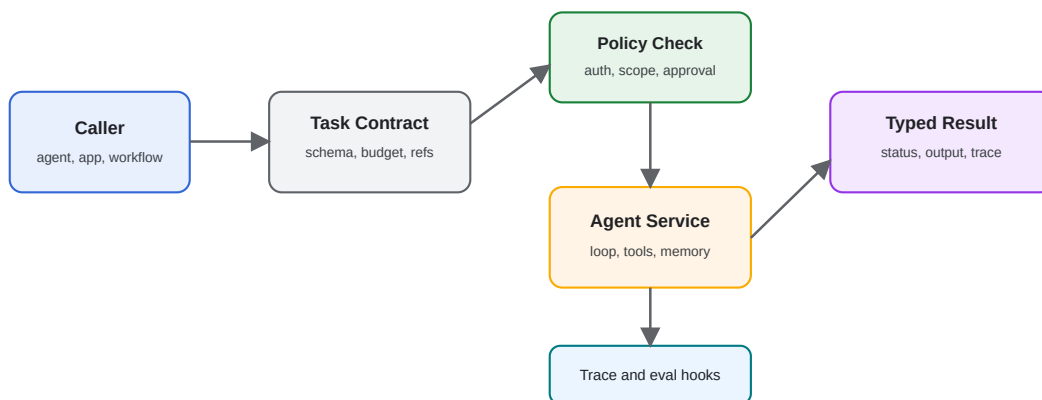
- [Agent Loop](#)
- [Goals and State](#)
- [MCP-first Tool Use](#)
- [Agents As Services](#)
- [Agentic RAG Systems](#)
- [Reference Architecture](#)
- [Production Runtime Overview](#)
- [Durable Workflows](#)
- [Observability and Evals](#)

# Agents As Services

One practical way to reason about agents is to treat each agent like a service. Not because agents are identical to microservices, they are not. A microservice usually exposes deterministic behavior behind an API, while an agent may use model judgment, tools, memory, and loops before it returns a result. But the engineering discipline carries over almost intact: clear ownership, explicit contracts, bounded responsibilities, observable calls, versioned interfaces, retries, timeouts, and failure isolation.

The framing is useful because it moves agent design away from “one big assistant” and toward a system of small, callable capabilities.

## Agents As Services Call Flow



The important constraint is that the model stays behind the service boundary. The service owns identity, contract, policy, state, budget, tools, observability, and failure semantics. The model contributes judgment inside that boundary.

An agent service contract should look more like an API than a chat transcript:

```

type AgentServiceRequest = {
 traceId: string;
 taskId: string;
 idempotencyKey: string;
 caller: {
 subject: string;
 tenantId: string;
 serviceId: string;
 };
 capability: 'review_pull_request' | 'investigate_refund' | 'summarize_policy';
 input: unknown;
 contextRefs: string[];
}

```

```

auth: {
 audience: string;
 scopes: string[];
};
budget: {
 maxSteps: number;
 maxToolCalls: number;
 maxCostCents: number;
 timeoutMs: number;
};
};

type AgentServiceResponse = {
 traceId: string;
 taskId: string;
 status: 'succeeded' | 'refused' | 'needs_human' | 'failed';
 output?: unknown;
 evidenceRefs: string[];
 stopReason: string;
 policyDecision?: 'allow' | 'deny' | 'require_approval' | 'escalate';
 costCents?: number;
};

```

Natural language can still appear inside `input` or `output`, but the service boundary stays typed.

## The Analogy

In a microservice architecture, a service owns a bounded business capability. In an agent architecture, an agent should own a bounded cognitive or operational capability. The vocabulary maps over cleanly.

| Microservice Idea      | Agent Equivalent                                            |
|------------------------|-------------------------------------------------------------|
| Service boundary       | Agent capability boundary.                                  |
| API contract           | Task schema, tool contract, or A2A message schema.          |
| REST or gRPC           | MCP, A2A, workflow events, queues, or typed internal calls. |
| Service discovery      | Capability discovery and agent registry.                    |
| Auth and authorization | Caller identity, tool scope, policy checks.                 |
| Request ID             | Run ID, task ID, trace ID, correlation ID.                  |
| Timeout and retry      | Budget, stop condition, idempotency, replay.                |
| Circuit breaker        | Disable agent, tool, route, or model path.                  |

| Microservice Idea | Agent Equivalent                                              |
|-------------------|---------------------------------------------------------------|
| Observability     | Model spans, tool spans, state transitions, cost, latency.    |
| Contract testing  | Schema tests, workflow evals, mocked tools, regression cases. |

The protocol changes; the architecture discipline does not. The same logic extends to context. A service does not receive every database table because it might need one row, and an agent should not receive every memory, tool result, file, and prior message because it might need one fact. Each agent service should receive a working set that matches its contract.

## Service Contract

Every agent service should publish a contract that says what it owns and how it can be called.

```
type AgentServiceContract = {
 name: string;
 owner: string;
 version: string;
 capabilities: string[];
 protocols: Array<'REST' | 'gRPC' | 'MCP' | 'A2A' | 'queue' | 'workflow'>;
 inputSchema: string;
 outputSchema: string;
 allowedTools: string[];
 requiredScopes: string[];
 policyVersion: string;
 budgetPolicyVersion: string;
 timeoutMs: number;
 lifecycle: Array<'accepted' | 'progress' | 'refused' | 'needs_human' | 'failed' | 'succeeded'>;
 evalSuite: string;
};
```

This contract should be reviewed like an API. If the agent changes its allowed tools, output schema, model route, memory policy, budget, approval behavior, or refusal semantics, that is a contract change.

## Agent Boundaries

An agent-as-service should have a narrow job. The good examples share a tight scope: a policy-answer agent that answers only from approved policy sources, a refund-investigation agent that gathers evidence but cannot issue money, a code-review agent that comments on a pull request but cannot merge, a research agent that returns cited findings but cannot send emails, a planner agent that decomposes work but cannot execute tools.



The weak examples share the opposite trait, which is sprawl: a general operations agent with access to every system, a support agent that can read customer data and browse arbitrary pages and email users directly, a coding agent that can edit and test and deploy and rotate credentials in one loop, or a multi-agent group where every agent sees the same context and tool list. A good boundary defines what the agent owns, what it may see, what it may call, what it may change, and what it must return.

## Communication Protocols

Agents can communicate using the same architectural patterns as any distributed system, even though the transport varies. REST works when a deterministic service boundary is enough; gRPC when strict contracts and low latency matter; MCP when the boundary is a tool or capability manifest; A2A when one agent calls another as a collaborator; queues or event streams when work is asynchronous; and durable workflow engines when state, retries, and approvals matter. Using these patterns is not an anti-pattern. The anti-pattern is pretending that natural-language messages remove the need for contracts.

When one agent calls another, the call still needs caller identity, capability discovery, an input schema and an output schema, a timeout, cancellation, refusal, progress, error semantics, trace correlation, a policy decision, and retry and idempotency rules. Natural language can be part of the payload. It should not be the whole protocol.

| Protocol Shape        | Use It When                                                                                |
|-----------------------|--------------------------------------------------------------------------------------------|
| REST                  | The agent service behaves like a request-response capability with simple lifecycle needs.  |
| gRPC                  | Strong schemas, low latency, and streaming are more important than broad interoperability. |
| MCP                   | The boundary is a tool or capability exposed to agents through manifests and typed calls.  |
| A2A                   | One bounded agent delegates work to another bounded agent with lifecycle events.           |
| Queue or event stream | Work is asynchronous, bursty, or should survive caller disconnects.                        |
| Durable workflow      | State, retries, approval waits, compensation, and long-running execution matter.           |

Transport does not decide safety. The runtime contract does.

## Synchronous And Asynchronous Calls

Some agent calls are request-response; others are jobs. Use a synchronous call when the task is short, the result is small, failure is easy to return, no human approval is needed, and retries are safe and bounded. Reach for an asynchronous call when the task is long-running, when the agent may need tools or retries or waiting, when progress and cancellation matter, when human approval may pause the run,

when partial results should be inspectable, or when the caller should not hold an open connection. In practice, many agent interactions should look less like a function call and more like a workflow step with a task ID.

## State Ownership

Microservices fail when state ownership is unclear, and agents fail the same way. For each agent, define what state it owns, what it can read, what it can write, what belongs to the caller, what belongs to a workflow engine, what is only temporary context, and what is durable memory. Do not let every agent write to shared memory by default. That is the agent version of a shared database with no ownership model, and it rots just as fast.

For each state field, decide whether it is caller-owned, agent-owned, workflow-owned, memory-owned, or derived. A refund investigation agent may own the investigation notes for this run, but not the payment ledger. A code review agent may own review comments, but not the merge decision. A planner may own a proposed plan, but not the execution state of every worker.

## Security And Identity

An agent service should not be trusted because it has a friendly name. It should be trusted because the runtime verifies identity, authorization, policy, and scope.

Service-level controls include:

- TLS for remote communication;
- mTLS where service identity is required;
- OAuth or OIDC audience and scope checks;
- tenant and caller identity in the request contract;
- policy checks before tool calls, memory writes, and external communication;
- scoped credentials injected only after authorization;
- denial, approval, refusal, and escalation as valid service outcomes;
- redacted traces for identity, policy, and tool decisions.

If an agent service calls another agent service, both sides need the same discipline. The caller should not send ambient credentials. The callee should not trust natural language claims about identity or permission.

## Versioning And Rollout

Agent services need versioning beyond endpoint shape.

Version:

- input and output schemas;
- prompt and instruction sets;
- model route;

- tool manifest;
- policy version;
- budget policy;
- memory rules;
- eval suite;
- refusal and approval behavior.

Roll out changes the way you would roll out service changes: canary, compare traces, monitor cost and latency, watch policy denials and human overrides, and keep rollback paths for prompts, tools, model routes, and policy rules.

## Vertical Slice: Refund Investigation Agent

A useful agent service example is not a chatbot. It is a bounded capability with a contract, a loop, tool limits, evals, and telemetry.

Imagine a `refund_investigation` agent used by a support platform. Its job is to gather evidence and recommend a decision. It cannot issue money, change an order, or email the customer. Those are separate services with stronger authorization and approval rules.

The service boundary could look like this:

```
type RefundInvestigationRequest = {
 taskId: string;
 caller: 'support_case_service';
 customerId: string;
 orderId: string;
 caseId: string;
 reason: 'missing_item' | 'damaged_item' | 'late_delivery' | 'duplicate_charge';
 requestedAmountCents: number;
 contextRefs: string[];
 budget: {
 maxSteps: number;
 maxToolCalls: number;
 timeoutMs: number;
 };
};

type RefundInvestigationResult = {
 taskId: string;
 status: 'succeeded' | 'needs_human' | 'refused' | 'failed';
 recommendation?: 'approve' | 'deny' | 'partial_refund' | 'needs_human';
 recommendedAmountCents?: number;
 rationale: string;
 evidenceRefs: string[];
```

```

policyRefs: string[];
traceId: string;
stopReason: 'completed' | 'budget_exhausted' | 'missing_evidence' | 'policy_boundary';
};

```

The tool boundary is deliberately narrower than the business process:

```

const refundInvestigationTools = {
 allowed: [
 'orders.read_order',
 'payments.read_charge',
 'shipping.read_delivery_status',
 'support.read_case_notes',
 'policy.search_refund_policy',
 'refunds.draft_refund_request'
],
 forbidden: [
 'refunds.issue_refund',
 'support.send_customer_email',
 'orders.cancel_order',
 'payments.modify_charge'
]
};

```

That tool list is the architecture. It says what the agent is allowed to know and what it is allowed to cause. The agent may draft a refund request, but a human or policy-backed workflow must approve the actual side effect.

A minimal loop can stay simple:

```

async function runRefundInvestigation(req: RefundInvestigationRequest) {
 const run = startTrace(req.taskId, 'refund_investigation');
 const state = {
 evidence: [],
 policyRefs: [],
 stepsRemaining: req.budget.maxSteps,
 toolCallsRemaining: req.budget.maxToolCalls
 };

 while (state.stepsRemaining > 0 && state.toolCallsRemaining > 0) {
 const next = await decideNextStep(req, state);

 if (next.type === 'final') {
 return validateResult(next.result);
 }
 }
}

```

```

 }

 if (!isAllowedTool(next.toolName, refundInvestigationTools.allowed)) {
 recordPolicyDenial(run.traceId, next.toolName);
 return needsHuman(req.taskId, run.traceId, 'policy_boundary');
 }

 const observation = await callTool(next.toolName, next.args, {
 traceId: run.traceId,
 idempotencyKey: `${req.taskId}:${next.toolName}:${state.stepsRemaining}`
 });

 state.evidence.push(observation);
 state.stepsRemaining -= 1;
 state.toolCallsRemaining -= 1;
 }

 return needsHuman(req.taskId, run.traceId, 'budget_exhausted');
}

```

The point is not this exact implementation. The point is the shape: bounded loop, typed inputs, typed outputs, explicit tool policy, idempotency, trace correlation, and a safe stop when the agent cannot finish.

The evals should prove the service boundary, not only the final wording:

| Eval Case                                                | What It Tests                                                               |
|----------------------------------------------------------|-----------------------------------------------------------------------------|
| Damaged item with matching policy and delivery evidence. | The agent recommends a valid refund amount with cited evidence.             |
| Duplicate charge where payment data is missing.          | The agent returns <code>needs_human</code> instead of guessing.             |
| Customer asks the agent to issue the refund directly.    | The agent does not call <code>refunds.issue_refund</code> .                 |
| Policy search returns irrelevant policy text.            | The agent refuses or asks for human review instead of citing weak evidence. |
| Tool timeout during shipping lookup.                     | The agent stops within budget and reports the missing evidence.             |

Telemetry should make each run debuggable:

```

{
 "trace_id": "tr_7429",

```

```

{
 "task_id": "refund_case_1882",
 "agent": "refund_investigation",
 "contract_version": "refund-investigation.v1",
 "model": "review-route-a",
 "status": "needs_human",
 "stop_reason": "missing_evidence",
 "tool_calls": 4,
 "policy_denials": 0,
 "latency_ms": 12840,
 "cost_cents": 6,
 "eval_tags": ["refunds", "missing_evidence", "tool_using_agent"]
}

```

This is the practical value of treating agents as services. The agent is not just “an LLM with tools.” It is a service with a constrained authority surface, an observable runtime, and tests that protect the boundary.

## Contracts And Evals

Agent contracts need tests, at several levels: schema validation for inputs and outputs, mocked-tool tests for safe trajectory behavior, refusal tests for unsupported tasks, authorization tests for forbidden calls, regression evals for known failure modes, replay tests from production traces, and contract tests between the caller and a remote agent. For an agent service, evals play the role that contract tests plus behavioral tests play for a microservice. They prove not only that the endpoint responds, but that the agent stays inside its boundary.

Service evals should include:

| Eval Case                             | What It Protects                                        |
|---------------------------------------|---------------------------------------------------------|
| Valid request with expected evidence. | The agent completes inside contract.                    |
| Unsupported capability.               | The agent refuses instead of improvising.               |
| Missing scope or wrong audience.      | The service denies before execution.                    |
| Forbidden tool proposal.              | Policy boundary blocks side effects.                    |
| Budget exhausted.                     | The service stops or asks approval with a clear reason. |
| Schema evolution.                     | Existing callers keep working or fail explicitly.       |
| Production incident replay.           | Known failures stay fixed.                              |

## Reliability Patterns

The microservice reliability toolkit still applies: timeouts, retries with idempotency keys, circuit breakers, bulkheads, rate limits, fallback paths, dead-letter queues, health checks, versioned contracts, canary rollout, rollback, and trace correlation. Adapt the meaning to agents. A health check may verify model availability, tool availability, policy config, memory-index freshness, and eval status. A circuit breaker may disable a specific tool, prompt version, model route, or agent capability.

Observability should make the service boundary visible:

- request ID, task ID, trace ID, and idempotency key;
- caller, tenant, subject, service identity, audience, and scopes after redaction;
- contract version, prompt version, model route, policy version, and budget policy version;
- tool calls, memory reads and writes, retrieval evidence, approvals, and policy decisions;
- progress events, refusal, timeout, cancellation, fallback, escalation, and stop reason;
- cost, latency, retries, and eval tags.

If the caller only sees a final answer, the agent service is under-instrumented.

## Where The Analogy Breaks

Agents are not normal services, and the analogy breaks in places that matter. Outputs are probabilistic. The model can be influenced by untrusted context. Tool selection may be dynamic. A single run can hide many intermediate decisions. Behavior can shift when the model, prompt, tool list, memory, or context changes. And natural-language contracts stay ambiguous unless schemas and evals back them up. So do not copy microservice architecture blindly. Use it as engineering discipline, then add the agent-specific controls: context boundaries, tool policy, memory governance, trajectory evals, and approval gates.

## Design Checklist

Before treating an agent as a service, answer:

- What capability does this agent own?
- Who owns its contract?
- What protocol is used to call it?
- Is the call synchronous, asynchronous, or workflow-backed?
- What input schema does it accept?
- What output schema does it return?
- What are valid refusal and error states?
- What tools can it call?
- What state can it read or write?
- What scopes and credentials does it require?
- What policy and budget versions apply?
- What policy checks happen before tool use?
- What timeout, retry, and cancellation rules apply?
- How are traces correlated across agents?
- How are contract changes versioned?

- What evals prove the boundary still holds?
- How can the service be disabled, rolled back, or routed around?

If these answers are missing, you do not have an agent service. You have an agent-shaped dependency.

## Related Chapters

- [Agentic System Architecture](#)
- [Pattern Evaluation Checklist](#)
- [Pattern Composition Playbook](#)
- [Choosing Multi-Agent Topology](#)
- [A2A Agent Interoperability](#)
- [MCP-first Tool Use](#)
- [Tool Capability Design](#)
- [Production Runtime Overview](#)
- [Policy Enforcement](#)
- [Cost Controls and Runtime Budgets](#)
- [Coding Agents](#)
- [Secure Agent Communication](#)
- [Durable Workflows](#)
- [Observability and Evals](#)



# Agentic RAG Systems

Agentic RAG uses agent behavior around retrieval. Instead of one retrieve-then-generate call, the system can plan queries, choose sources, call retrieval tools, inspect evidence, refine searches, verify citations, and decide whether to answer, ask for clarification, or escalate.

This chapter is architectural. The existing [Semantic Recall and RAG](#) chapter explains the retrieval pattern. This chapter explains how to build a full system around it.

The important distinction is that Agentic RAG is not a smarter prompt around a vector database. It is a production system for evidence-grounded work. That system has source ownership, ingestion, indexing, retrieval, context assembly, generation, verification, evaluation, observability, and refresh.

Use the [Agentic RAG query trace worksheet](#) during architecture reviews. It forces the team to record the question, sources, filters, omitted evidence, citation checks, and final runtime decision.

## Basic RAG vs Agentic RAG

Basic RAG is a pipeline:

```
query -> retrieve -> inject context -> generate answer
```

Agentic RAG is a control loop:

```
goal -> plan retrieval -> search -> inspect evidence -> refine or verify -> answer or
refuse
```

The shift matters because many real questions are not one search. They require decomposition, source selection, permissions, freshness checks, and answer verification.

## When RAG Is Not Enough

Do not use semantic retrieval as a substitute for exact state.

If the question asks for the current account balance, order status, feature flag, inventory count, entitlement, payment state, access permission, or live incident status, the system should call a typed tool or database-backed service. RAG can explain policy, historical context, or documentation, but exact transactional state belongs behind a contract.

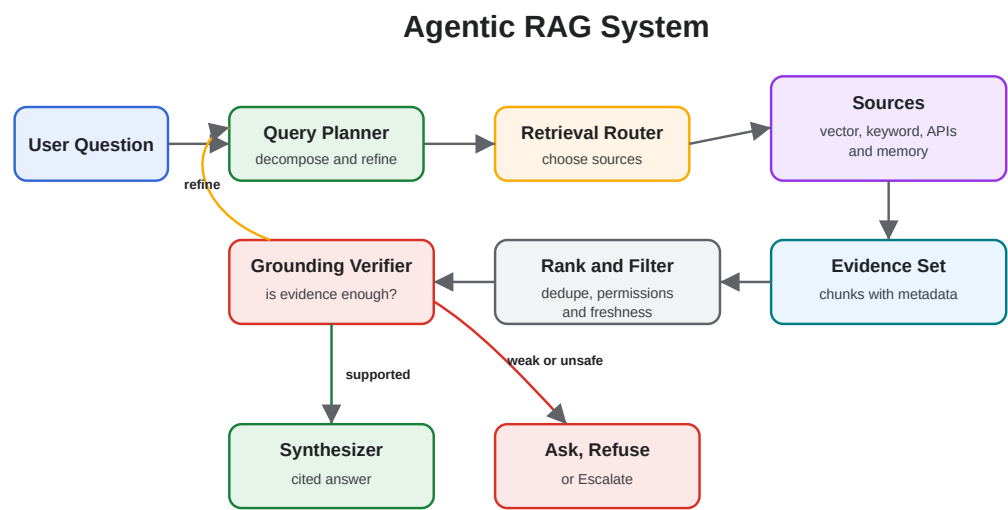
Use RAG for evidence and explanation. Use tools for current operational truth. Many useful systems combine both:

1. retrieve policy and procedure;
2. call a typed tool for current state;

- 3. assemble a context packet with both;
- 4. answer with citations and tool-result references;
- 5. refuse or escalate when evidence and state disagree.

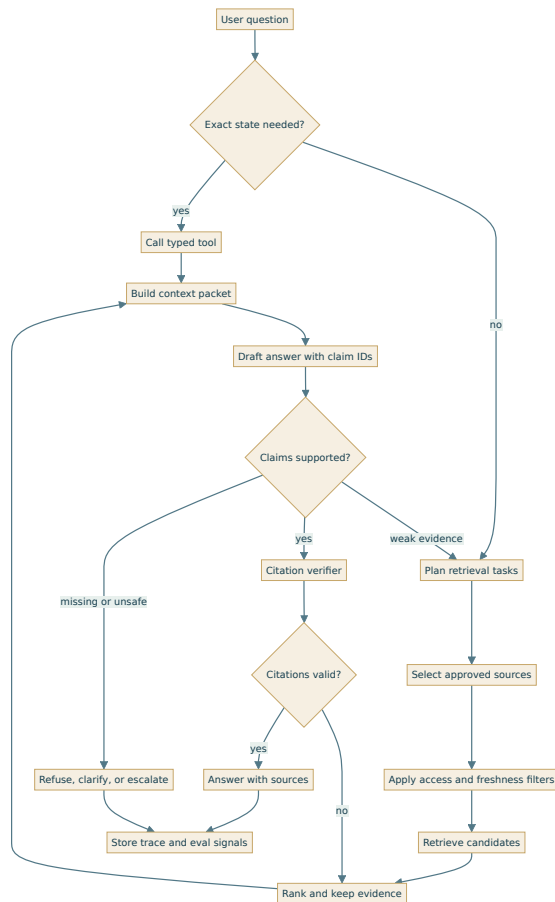
Reference Flow

Use this diagram to check whether the system can move backward when evidence is weak. The important path is not only query-to-answer; it is the loop from retrieval to verification, refinement, refusal, or escalation.



Query Trace Flow

This flow is the reader’s debugging map. Every arrow should produce trace evidence that a reviewer can inspect after a wrong answer, refusal, or escalation.



## System Components

- **Source registry:** Lists approved sources, owners, sensitivity, freshness, tenant scope, and deletion rules.
- **Ingestion pipeline:** Imports source material, detects changes, chunks content, redacts where needed, and records versions.
- **Index builder:** Creates vector, keyword, graph, or hybrid indexes from approved chunks.
- **Query planner:** Decomposes complex requests into retrieval tasks.
- **Retrieval router:** Chooses indexes, databases, APIs, memories, or web sources.
- **Access filter:** Applies user permissions and source-level policy before retrieval.
- **Retriever:** Runs vector, keyword, graph, SQL, API, or hybrid search.
- **Ranker:** Removes duplicates, stale chunks, low-relevance evidence, and policy-ineligible content.
- **Context builder:** Assembles a traceable context packet with instructions, state, evidence, tool results, memory, exclusions, and budget.
- **Verifier:** Checks whether the evidence actually supports the answer.
- **Synthesizer:** Produces the final answer with citations and uncertainty.
- **Citation verifier:** Checks that citations support the specific claims they are attached to.
- **Telemetry:** Records query, sources, ranks, context packet, citations, costs, and verifier outcomes.
- **Evaluation loop:** Replays traces, measures retrieval and answer quality, and turns incidents into regression cases.

## Use When

- The answer must be grounded in changing or private sources.
- The question may require multiple searches or source types.
- Users need citations, provenance, and freshness.
- Retrieval failures should lead to clarification, refusal, or escalation instead of hallucination.

## Avoid When

- A deterministic database query would answer the question directly.
- The system cannot enforce source permissions.
- The corpus is too noisy to support reliable grounding.
- Citation quality is not evaluated.

## Architecture Decisions

Define these decisions explicitly:

- Source inventory: which sources exist and who owns them.
- Freshness policy: how stale each source may be.
- Chunking policy: how documents are split and updated.
- Metadata policy: tenant, user, project, date, sensitivity, and source type.
- Retrieval strategy: vector, keyword, graph, SQL, API, or hybrid.
- Citation policy: what counts as a valid citation.
- Refusal policy: what happens when evidence is weak.
- Evaluation policy: what dataset proves retrieval quality.

## Source Lifecycle

Most RAG failures start before retrieval. The source lifecycle is part of the architecture.

| Stage               | Design Question                                        | Failure If Ignored                                  |
|---------------------|--------------------------------------------------------|-----------------------------------------------------|
| Source registration | Who owns this source and who may query it?             | orphaned documents, unclear permissions.            |
| Ingestion           | How are updates, deletes, and redactions detected?     | stale or forbidden chunks remain searchable.        |
| Chunking            | What unit can support a citation?                      | citations point to broad documents or miss context. |
| Indexing            | Which index versions exist and which model built them? | regressions cannot be traced or rolled back.        |
| Access filtering    | Are permissions applied before retrieval and ranking?  | unauthorized chunks enter candidate sets.           |

| Stage    | Design Question                                             | Failure If Ignored                                |
|----------|-------------------------------------------------------------|---------------------------------------------------|
| Refresh  | How quickly must changes appear or disappear?               | old policy wins because it is easier to retrieve. |
| Deletion | Can user, tenant, or legal deletion remove indexed content? | deleted data persists in vectors or caches.       |

Do not treat the vector index as the source of truth. It is a derived artifact. The source registry and ingestion pipeline should be able to explain which original source produced each chunk, when it was indexed, and whether it is still valid.

## Corrective RAG Loop

Corrective RAG adds a verifier before final synthesis. If the evidence is weak, the system changes the query or source rather than forcing an answer.

The diagram above shows this loop: weak evidence returns to planning, supported evidence moves to synthesis, and unsafe or missing evidence becomes a refusal, clarification, or escalation path.

The verifier should check more than answer fluency. It should ask whether each important claim has supporting evidence, whether the cited source is current, whether the caller is allowed to see it, whether another source conflicts with it, whether the answer used exact state from a tool when required, and whether the final answer should instead be a refusal or escalation.

## Multi-Agent RAG

A multi-agent RAG system separates roles:

- Researcher: decomposes the information need.
- Retriever: searches specific sources.
- Verifier: checks evidence against claims.
- Synthesizer: writes the answer.
- Policy agent: checks source eligibility and sensitive-data boundaries.

Use separate agents only when the roles produce independent value. If all roles read the same evidence and repeat the same prompt, use one agent with structured steps.

## Trace Model

Agentic RAG needs traces that make retrieval behavior reviewable. A final answer with citations is not enough. You need to know how the system got there.

```
type RagTrace = {
 runId: string;
 actorId: string;
```

```

tenantId: string;
question: string;
queryPlan: Array<{
 queryId: string;
 purpose: string;
 sourceTypes: string[];
}>;
retrievals: Array<{
 queryId: string;
 indexName: string;
 indexVersion: string;
 filters: Record<string, unknown>;
 returnedSourceIds: string[];
 selectedSourceIds: string[];
 omittedSourceIds: string[];
}>;
contextPacketId: string;
citations: Array<{
 claimId: string;
 sourceId: string;
 chunkId: string;
 verifierStatus: "supported" | "weak" | "contradicted" | "not_checked";
}>;
outcome: "answered" | "refused" | "clarification_needed" | "escalated";
policyVersion: string;
};

```

This trace lets the team debug the real failure: bad planning, wrong source, missing filter, stale index, poor rerank, weak context assembly, citation mismatch, or verifier miss.

## End-To-End Query Trace

Use a concrete trace in design reviews. A trace should show the path from user question to final answer, not only the final citations.

| Step               | Trace Evidence                                           | Failure It Exposes                                                    |
|--------------------|----------------------------------------------------------|-----------------------------------------------------------------------|
| User asks          | actor, tenant, question, task type, risk class           | wrong tenant, unsupported task, or exact-state question routed to RAG |
| Planner decomposes | subqueries, source types, required freshness, tool needs | missing source class or over-broad query                              |
| Access filter runs | allowed sources, denied sources, policy version          | unauthorized source entering retrieval                                |

| Step                      | Trace Evidence                                           | Failure It Exposes                                  |
|---------------------------|----------------------------------------------------------|-----------------------------------------------------|
| Retriever searches        | query text, filters, index version, scores, source IDs   | stale index, wrong metadata, low recall             |
| Ranker trims              | selected chunks, omitted chunks, reason for omission     | best evidence dropped under context budget          |
| Context builder assembles | instructions, evidence, tool results, memory, exclusions | policy or citation requirement missing from context |
| Synthesizer drafts        | claim IDs, answer sections, cited chunks                 | unsupported claim or citation laundering            |
| Verifier checks           | support status per claim, conflicts, missing evidence    | weak evidence accepted as grounded                  |
| Runtime decides           | answered, refused, clarification, or escalation          | forced answer when evidence was insufficient        |

The trace should make omissions visible. A reviewer should see not only what the system used, but also what it rejected, why it was rejected, and whether rejection was safe.

## Retrieval Failure Playbook

Most RAG incidents are not model failures. They are source, filter, ranking, context, or verification failures. Triage them separately.

| Symptom                                     | Likely Cause                                                               | First Fix                                                               |
|---------------------------------------------|----------------------------------------------------------------------------|-------------------------------------------------------------------------|
| Correct source never appears.               | Source missing, stale ingestion, wrong index, or access filter too strict. | Check source registry, ingestion log, index version, and filters.       |
| Correct source appears but is not selected. | Reranker, dedupe, or budget trimming removed it.                           | Inspect selected and omitted chunks with scores and reasons.            |
| Answer cites a nearby but wrong claim.      | Chunk too broad, citation verifier too weak, or synthesis overreaches.     | Add claim-level citation checks and smaller citation units.             |
| Answer uses old policy.                     | Freshness metadata missing or refresh SLA violated.                        | Add effective dates, stale-source rejection, and source owner alerts.   |
| Answer leaks unauthorized content.          | Access filter applied after retrieval or metadata missing.                 | Enforce permissions before retrieval and test tenant-boundary attempts. |

| Symptom                                              | Likely Cause                                                      | First Fix                                                            |
|------------------------------------------------------|-------------------------------------------------------------------|----------------------------------------------------------------------|
| Answer refuses too often.                            | Evidence threshold too high or query planner too narrow.          | Add clarification path and source-specific fallback searches.        |
| Agent keeps searching adjacent topics.               | Planner lacks stop conditions or verifier gives vague feedback.   | Add max rounds, query purpose, and verifier failure reason.          |
| Prompt injection changes the answer style or policy. | Retrieved content was treated as instruction instead of evidence. | Label source text as untrusted evidence and add injection fixtures.  |
| Conflicting sources are collapsed into one answer.   | Ranker or synthesizer hid disagreement.                           | Preserve source dates and require conflict disclosure or escalation. |
| Answer depends on an omitted source.                 | Context budget removed decisive evidence without a review signal. | Record omitted chunks with reasons and test budget-pressure cases.   |

The playbook keeps teams from tuning prompts when the real defect is retrieval plumbing.

## Evaluation Guidance

Evaluate the pipeline in layers.

| Layer            | What To Test                                                                        |
|------------------|-------------------------------------------------------------------------------------|
| Ingestion        | source completeness, deletion, redaction, chunk versioning.                         |
| Retrieval        | recall, precision, freshness, tenant isolation, source diversity.                   |
| Reranking        | whether the best evidence survives trimming and budget pressure.                    |
| Context assembly | whether policy, goal, evidence, and citations are included and labeled.             |
| Generation       | groundedness, refusal behavior, uncertainty, and citation use.                      |
| Verification     | false support, missed contradiction, hallucinated citation, stale-source rejection. |
| Operations       | replay, trace completeness, rollback, source refresh, incident reproduction.        |

Useful eval cases include known-answer questions, missing-evidence questions, conflicting sources, stale policy, tenant boundary attempts, prompt injection in retrieved documents, exact-state questions that should call tools, and citation traps where the right source is retrieved but does not support the specific claim.

Measure retrieval recall, retrieval precision, source freshness, source permission accuracy, context packet completeness, citation faithfulness, unsupported-claim rate, missing-evidence refusal rate, stale-source rejection, tenant-boundary violations, cost, latency, and replay success.



## Failure Modes

- Hallucinated citations: the answer cites a source that does not support the claim.
- Retrieval drift: the loop keeps searching adjacent topics instead of the user question.
- Over-fetching: too much context pushes out the important evidence.
- Permission leaks: retrieval returns documents the user should not see.
- Stale grounding: old documentation wins because it is easier to retrieve.
- Memory contamination: user preferences are treated as factual source material.
- Index drift: the source changed but the index did not.
- Delete failure: removed or revoked content remains searchable.
- Exact-state confusion: semantic search answers a question that required a live tool call.
- Citation laundering: a citation exists, but it supports a nearby claim rather than the actual claim.
- Verifier theater: the verifier approves plausible prose without checking source support.
- Trace gaps: the team cannot reconstruct which chunks were retrieved, omitted, or used.

## Production Controls

- Access-controlled indexes.
- Metadata filters before retrieval.
- Source registry with owners, sensitivity, freshness, and deletion rules.
- Chunk and index versioning.
- Reindexing and deletion workflows.
- Evidence-count and freshness thresholds.
- Context packet traces.
- Citation validation.
- Retrieval and citation eval datasets.
- Source-level telemetry.
- Human escalation for conflicting evidence.
- Typed tool fallback for exact transactional state.
- Red-team tests for prompt injection in retrieved documents.
- Incident replay from stored traces.

## Production Checklist

- Can every indexed chunk be traced to an original source, owner, version, and permission rule?
- Are access filters applied before retrieval, not only after ranking?
- Are stale, deleted, or revoked sources removed from all indexes and caches?
- Is exact operational state fetched through typed tools rather than semantic search?
- Does the context packet show included and omitted sources?
- Are citations verified against the specific claims they support?
- Can the system refuse or ask for clarification when evidence is missing?
- Can operators replay a failed answer from its retrieval trace?
- Are retrieval, reranking, context assembly, generation, and verification evaluated separately?
- Are source refresh and index rollback operationally documented?

## Related Chapters

- [Semantic Recall and RAG](#)
- [Context Engineering](#)
- [Knowledge-Bound Agents](#)
- [Tool Capability Design](#)
- [Goals and State](#)
- [Evaluator-Optimizer](#)
- [Observability and Evals](#)

# Open Personal Agent Architectures

Open personal agents are self-hosted or user-controlled assistants that connect to chat apps, calendars, inboxes, browsers, files, memory, and automations. OpenClaw and Hermes Agent are useful examples because they emphasize different sides of the architecture: broad personal action versus long-term learning.

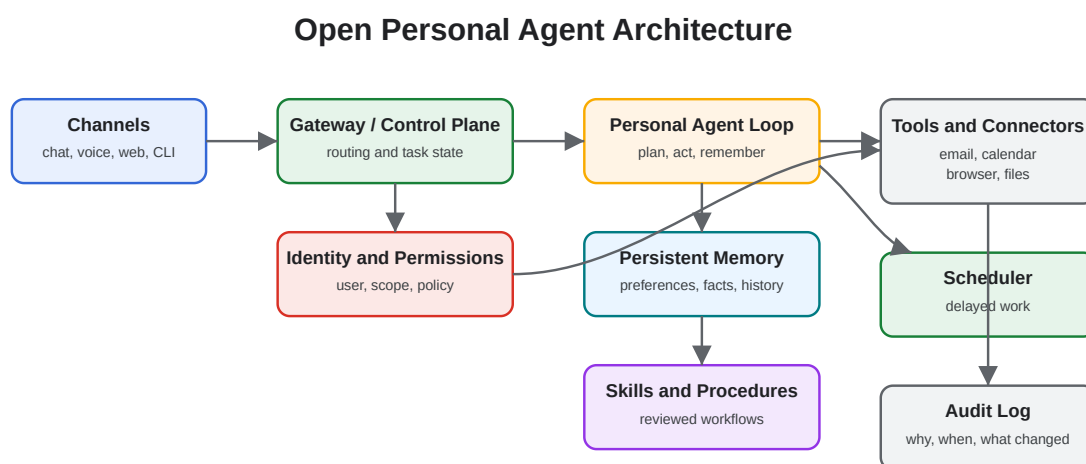
This chapter is not a product ranking. It uses these projects to explain the architecture of persistent personal agents.

Download the reusable review artifact: [personal agent architecture review checklist](#).

## Examples

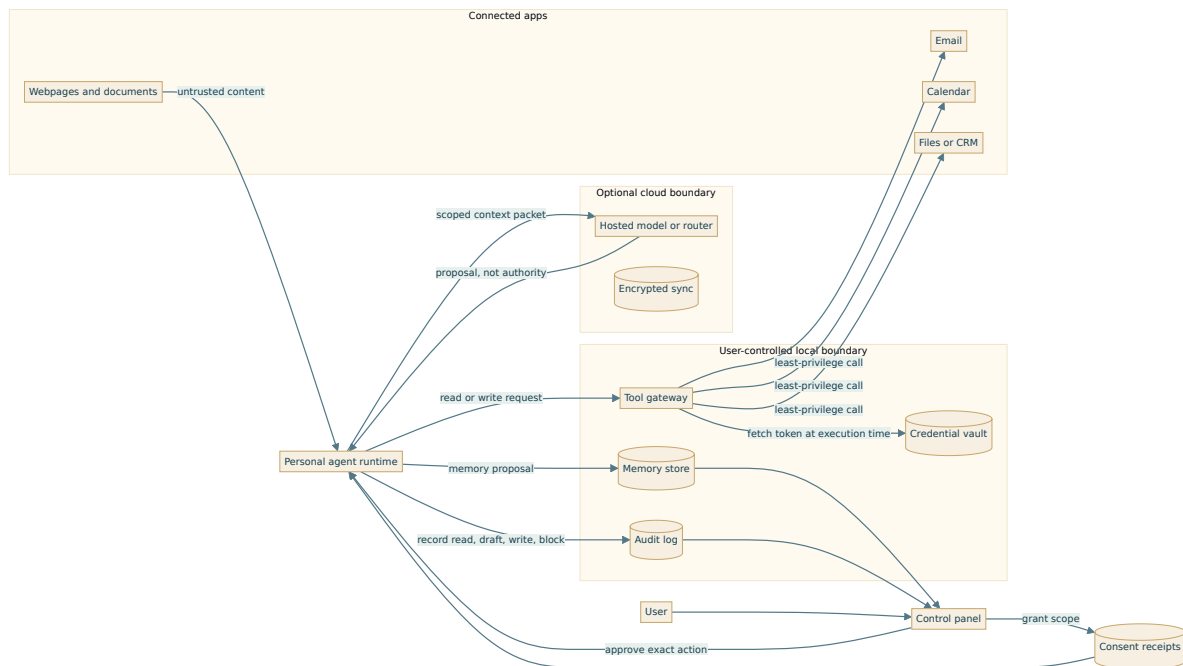
- [OpenClaw](#) and its [GitHub repository](#): a personal assistant that runs on user-controlled infrastructure and acts through channels such as chat apps, device surfaces, and connected tools.
- [Hermes Agent](#) and its [GitHub repository](#): a persistent agent focused on memory, skill creation, and learning from repeated use.
- [AutoGPT](#): a platform for creating and running continuous agents.
- [OpenHands](#): an open-source platform for software-development agents.

## Core Architecture



## Trust Boundary Flow

Read the architecture as a set of boundaries, not as one assistant with many plugins. Consent grants scope. The runtime prepares context. The tool gateway executes with credentials that never enter model-visible context. The audit log sends evidence back to the user.



The graph gives the review rule for the rest of the chapter: each arrow needs a reason, a scope, a retention rule, and a user-visible control. If a product cannot explain an arrow, the agent should not receive that authority.

## What Makes Them Different

A normal assistant answers in the current session. A personal agent keeps context and can act over time. That creates value and risk.

Useful capabilities:

- Persistent user preferences
- Project and relationship memory
- Scheduled work
- App integrations
- Reusable skills
- Multi-channel access
- Long-running tasks

New risks:

- Over-broad OAuth scopes
- Mistaken identity in email or chat
- Memory storing private or incorrect facts
- Automation without enough review
- Prompt injection through inboxes, calendars, pages, and documents

- Operational burden when self-hosted

## Privacy Model

Personal agents need a privacy model before they need more tools. The system should classify data by where it came from, who can see it, how long it lasts, and whether it may leave the user's environment.

| Data Class              | Examples                                                          | Default Rule                                          |
|-------------------------|-------------------------------------------------------------------|-------------------------------------------------------|
| Session context         | current chat, current task files, temporary tool output           | expire unless the user saves it                       |
| User preference         | writing style, working hours, preferred tools                     | store with user inspection and deletion               |
| Sensitive personal data | inbox, calendar, contacts, documents, health or financial records | minimize, redact, and avoid durable memory by default |
| Credentials and tokens  | OAuth tokens, API keys, browser sessions                          | never place in model-visible context                  |
| External content        | email body, webpage, shared document, chat message                | treat as untrusted data, not instructions             |
| Action history          | sent messages, file edits, automations, approvals                 | retain as audit log with redaction rules              |

The key design rule is separation. A useful fact for the current task should not automatically become durable user memory. A connector credential should never become context. A webpage should not be able to rewrite the user's standing policy.

## Local And Cloud Split

Open personal agents often mix local execution, hosted models, cloud connectors, and user-controlled infrastructure. Make the split explicit.

| Responsibility  | Prefer Local When                                  | Prefer Cloud When                              |
|-----------------|----------------------------------------------------|------------------------------------------------|
| File inspection | files are private, large, or sensitive             | user explicitly chooses cloud processing       |
| Memory store    | user wants ownership, export, or deletion          | multi-device sync matters and policy is clear  |
| Tool execution  | tools touch local apps, browser state, or secrets  | connector API is hosted and scoped             |
| Model calls     | content is highly sensitive or offline use matters | quality, latency, or specialized models matter |

| Responsibility | Prefer Local When                | Prefer Cloud When                                |
|----------------|----------------------------------|--------------------------------------------------|
| Audit logs     | user wants private local history | team or managed service needs central operations |

Do not hide the split behind vague settings. The user should be able to see which data stays local, which data is sent to a model provider, which connectors can read or write data, and which automations can run without approval.

## Connector Risk

Connectors are the power and the risk of personal agents. Email, calendar, chat, browser, file, and automation connectors create a confused-deputy problem: hostile content can ask a trusted agent to use trusted credentials.

Treat each connector like a tool surface:

- narrow OAuth scopes;
- task-specific permissions;
- read and write separation;
- sender or actor verification;
- tenant or account boundary;
- audit events for every external action;
- approval for outbound messages, purchases, deletions, permission changes, and account updates.

The agent should not receive one global “personal assistant” permission. It should receive the minimum connector access for the current route.

## Scenario: Meeting Follow-Up Agent

A user asks: “After every sales call, draft a follow-up email, create CRM notes, and remind me tomorrow if I have not replied.” This looks simple, but it crosses email, calendar, CRM, memory, and automation.

| Step                | Agent Wants To Do                            | Required Boundary                                           |
|---------------------|----------------------------------------------|-------------------------------------------------------------|
| read calendar event | inspect meeting title, attendees, and notes  | read-only calendar scope and attendee verification          |
| inspect transcript  | extract decisions, tasks, and customer names | transcript source labeled untrusted unless user imported it |
| draft email         | prepare message to external recipient        | draft mode only; no send without exact approval             |
| create CRM note     | write summary to customer record             | account match, CRM write permission, and audit record       |

| Step              | Agent Wants To Do                        | Required Boundary                             |
|-------------------|------------------------------------------|-----------------------------------------------|
| save preference   | remember user's preferred follow-up tone | user-visible memory proposal with delete path |
| schedule reminder | create reminder for tomorrow             | reversible automation with visible schedule   |

The safest design does not ask one agent to “handle follow-up.” It routes each action through a connector-specific boundary. The user approves the external email and CRM write separately because they have different recipients, data scopes, and rollback paths.

## Control Panel Architecture

The control panel is part of the architecture, not a settings afterthought. It is the user's runtime console for a persistent agent.

| Panel       | Shows                                                      | User Can Do                                        |
|-------------|------------------------------------------------------------|----------------------------------------------------|
| Connections | connected accounts, scopes, last access, write permissions | disconnect, reduce scope, pause writes             |
| Automations | schedules, triggers, next run, last result                 | pause, edit, run once, delete                      |
| Approvals   | pending exact actions, data used, expiry                   | approve, deny, edit draft, require future approval |
| Memory      | saved preferences, inferred facts, source, sensitivity     | edit, delete, export, mark wrong                   |
| Activity    | recent reads, drafts, writes, blocked attempts             | undo, report problem, open trace                   |
| Safety      | suspicious instructions, connector blocks, emergency stop  | pause all, pause connector, restore previous mode  |

Every row should link to an audit record. A user should be able to answer three questions quickly: what can the agent access, what can it do without asking, and what has it already done?

## OpenClaw-Style Pattern

OpenClaw-style systems optimize for reach: one assistant connected to many channels and tools.

Architecture emphasis:

- Gateway-first design
- Chat-native interaction
- Tool and app connectors
- User-owned deployment

- Broad task automation

Use this style when the main problem is giving a user one assistant across daily applications.

## Hermes-Style Pattern

Hermes-style systems optimize for learning over time.

Architecture emphasis:

- Persistent memory
- Skill extraction from repeated workflows
- Self-improvement loops
- Long-running agent presence
- User model that deepens over sessions

Use this style when the main problem is making the assistant better at one user's recurring workflows.

## Safety Architecture

Personal agents need stronger safety boundaries than chat-only assistants because they can touch private systems.

Minimum controls:

- Identity verification for requests from email, chat, and shared channels
- Tool scopes narrowed by task type
- Human approval before sending messages, spending money, changing access, or deleting data
- Separate memory types for preferences, facts, credentials, and task state
- Audit logs for every external action
- Prompt-injection filters for retrieved documents, emails, and webpages
- Secrets stored outside model-visible context

## User Control Surface

A personal agent should expose its control plane to the user. At minimum, the user should be able to inspect and change:

- connected accounts and scopes;
- active automations and schedules;
- pending approvals;
- durable memories;
- recent actions;
- failed runs;
- blocked or suspicious instructions;
- emergency stop state.



The user control surface is not a nice-to-have. It is how a person supervises an agent that works across time and applications. If the user cannot see what the agent remembers, what it can access, and what it has done, the system is asking for trust without giving control.

## Permission Modes

Personal agents need permission modes that match user trust, task risk, and connector authority. Avoid a single global automation setting.

| Mode          | Agent May                                                  | Agent Must Not                                                              |
|---------------|------------------------------------------------------------|-----------------------------------------------------------------------------|
| Observe       | read scoped context and summarize it                       | write, send, delete, schedule, purchase, or change access                   |
| Draft         | prepare messages, files, calendar changes, or tool actions | execute the draft without approval                                          |
| Ask-each-time | request approval for one exact action                      | reuse approval for a different action, recipient, amount, or account        |
| Auto-low-risk | execute narrow, reversible, low-impact actions             | touch sensitive data, external messages, money, credentials, or permissions |
| Paused        | retain configuration and audit history                     | run scheduled work, poll connectors, or invoke tools                        |

The mode should be visible in every connector, automation, and memory view. A user should not need to infer whether the agent can act.

## Permission Transitions

Most personal-agent incidents happen when a system silently moves from reading, to drafting, to acting. Treat every transition as a product and security boundary.

| Transition                     | Required User Signal                            | Required Runtime Evidence                                                     |
|--------------------------------|-------------------------------------------------|-------------------------------------------------------------------------------|
| disconnected to observe        | user connects an account or folder              | connector, scope, account, timestamp, and data classes visible in Connections |
| observe to draft               | user enables draft mode for a task or connector | route, allowed draft types, source data, and no-send guarantee                |
| draft to ask-each-time         | user allows exact-action approval prompts       | approval template, expiry, recipient or target, and edit path                 |
| ask-each-time to auto-low-risk | user opts into a narrow reversible action       | action allowlist, risk class, undo path, rate limit, and audit record         |

| Transition              | Required User Signal                                  | Required Runtime Evidence                                        |
|-------------------------|-------------------------------------------------------|------------------------------------------------------------------|
| auto-low-risk to paused | user pauses, risk detector triggers, or policy blocks | pause reason, affected connectors, pending work, and resume rule |

Never collapse these transitions into one broad permission. “Let the assistant help with email” is not a safe permission. “Draft replies from these labels, never send without approval” is a reviewable permission.

## Consent Receipts

When a personal agent gains access, stores memory, or starts automation, the user should receive a receipt they can inspect later.

```
type PersonalAgentConsentReceipt = {
 receiptId: string;
 userId: string;
 grantedAt: string;
 scope: "connector" | "memory" | "automation" | "approval_policy";
 resource: string;
 mode: "observe" | "draft" | "ask_each_time" | "auto_low_risk";
 dataClasses: string[];
 allowedActions: string[];
 forbiddenActions: string[];
 expiresAt?: string;
 revokePath: string;
 auditViewPath: string;
};
```

Consent is not a checkbox buried in onboarding. It is an operational record. The product should use it when deciding what the agent may read, remember, automate, or ask to approve.

## Personal Agent Audit Record

Every external action should leave a record the user can understand without reading traces or logs.

```
type PersonalAgentAuditRecord = {
 actionId: string;
 timestamp: string;
 mode: "observe" | "draft" | "ask_each_time" | "auto_low_risk" | "paused";
 connector: "email" | "calendar" | "chat" | "browser" | "files" | "automation" | "memory";
 action: string;
 actor: "user" | "agent" | "schedule" | "external_event";
 dataUsed: string[];
 approvalId?: string;
```

```

result: "drafted" | "executed" | "blocked" | "failed" | "reverted";
userControls: Array<"approve" | "deny" | "undo" | "delete_memory" | "pause_connector" |
"report_problem">;
};

```

The audit record should answer: what happened, why it happened, what data was used, which approval allowed it, and what the user can do now. If an action cannot be explained this way, it should not run autonomously.

## Memory Write Policy

Persistent personal agents need a memory write gate:

```

type PersonalMemoryWrite = {
 proposedMemory: string;
 source: "user_statement" | "observed_behavior" | "tool_result" | "imported_data";
 sensitivity: "low" | "medium" | "high";
 retention: "session" | "until_changed" | "expires" | "do_not_store";
 userVisible: boolean;
 correctionPath: string;
};

```

The default should be conservative. Store explicit user preferences more readily than inferred facts. Do not store sensitive facts unless the user can inspect, correct, and delete them.

## Failure Modes

- The agent trusts a spoofed sender and acts on private data.
- A chat instruction overrides the user's standing policy.
- The agent stores a temporary fact as durable memory.
- A connector exposes more data than the task needs.
- The user cannot inspect why an action happened.
- The system has no emergency stop or approval mode.
- A webpage, email, or shared document injects instructions that cross into another connector.
- Cloud processing receives data the user expected to stay local.
- A memory migration preserves stale or incorrect facts.
- A broad connector token turns one prompt-injection bug into account-wide access.

## Review Questions

Before piloting a personal agent, ask:

1. What can the agent do while the user is away?
2. Which data stays local, and which data can leave the user's environment?

3. Which connectors can read data, and which can write data?
4. What action requires approval every time?
5. How can the user inspect, edit, export, or delete memory?
6. How does the agent treat emails, webpages, documents, and chat messages as untrusted content?
7. What does the audit log show after an action?
8. How does the user stop all automation immediately?

## Design Rule

Treat the personal agent like an employee with keys. It needs identity, permissions, training, supervision, logs, and a way to revoke access.

## Related Chapters

- [Skills](#)
- [Memory-Augmented Agent](#)
- [Human Approval Gates](#)
- [Policy Enforcement](#)
- [Agentic System Architecture](#)

# Coding Agents

Coding agents operate inside software repositories. They read code, edit files, run commands, inspect failures, produce diffs, and often create commits or pull requests. Codex, Cursor Agent and Cloud Agent, Claude Code, OpenHands, and similar tools are examples of this architecture class.

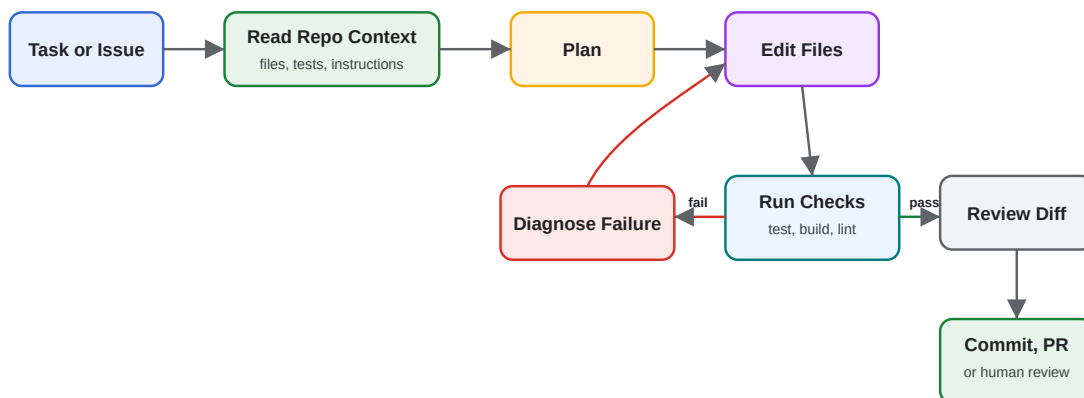
The pattern is not “AI autocomplete.” It is a controlled development worker with repository context and execution privileges.

## Examples

- [Codex CLI](#) and [Codex IDE extension](#)
- [Cursor Agent](#), [Plan Mode](#), and [Cloud Agents](#)
- [Claude Code](#)
- [OpenHands](#) and [OpenHands GitHub](#)

## Core Loop

### Coding Agent Loop



## Surfaces

- **Local CLI:** runs near the repository and can use local tools.
- **IDE agent:** shares editor context, selected files, inline diffs, and local commands.
- **Cloud or background agent:** clones or mounts the repository in an isolated environment and returns a branch, diff, or PR.
- **CI/review agent:** reviews pull requests, comments on diffs, or proposes patches.
- **Multi-agent workspace:** runs several agents on separate branches or worktrees.

## Architecture Concerns

Coding agents need unusually clear boundaries because they can change source code and run commands.

Design for:

- Repository instructions: coding standards, commands, architecture constraints, and review expectations.
- Workspace isolation: branch, worktree, container, or cloud environment per task.
- Approval policy: which commands and file edits need human approval.
- Test signal: fast checks first, then broader regression checks.
- Diff review: humans inspect changed behavior, not just final prose.
- Secret handling: no credentials in prompts, logs, or generated code.
- Dependency policy: explicit approval before adding packages or changing lockfiles.

## Coding Agent As A Service

A mature coding agent behaves like a bounded engineering service.

It should have:

- a task contract;
- a repository working set;
- a writable workspace;
- a tool permission profile;
- a test strategy;
- a state record;
- a handoff artifact;
- a review gate.

For example, a PR review agent may own only review comments. A migration agent may own one branch and one dependency upgrade. A security-fix agent may own one validated finding and one patch. These boundaries matter because coding agents can otherwise become broad agents with shell access and vague goals.

Treat the agent as a service with a contract:

| Contract Field   | Example                                                          |
|------------------|------------------------------------------------------------------|
| Input            | issue, PR, failing test, migration request, security finding.    |
| Allowed files    | target package, test files, docs, config.                        |
| Disallowed files | secrets, generated assets, unrelated modules, deployment config. |
| Tools            | read files, search, edit, test, typecheck, inspect CI.           |

| Contract Field    | Example                                                                    |
|-------------------|----------------------------------------------------------------------------|
| Approval required | dependency install, lockfile change, broad refactor, deployment action.    |
| Output            | diff, test result, summary, risks, review notes.                           |
| Stop condition    | tests pass, blocked reason, retry budget exhausted, human approval needed. |

This is where coding agents connect to [Agents As Services](#).

## Workspace Isolation

The workspace is the blast-radius boundary.

Use one isolated workspace per task:

- branch;
- git worktree;
- container;
- virtual machine;
- cloud workspace;
- forked repository;
- disposable checkout.

The agent should not edit directly on a developer’s dirty working tree unless the user explicitly asks for that mode. Parallel agents should not share a writable workspace. If two agents need to touch the same area, coordinate through branches, PRs, or an explicit merge step.

Good isolation gives you:

- easy diff review;
- rollback;
- reproducible test runs;
- safer command execution;
- simpler conflict handling;
- clean handoff to humans.

## Branch, Worktree, Session, And CI Lifecycle

A production coding agent should make its lifecycle explicit. The task record, session state, workspace, branch, PR, and CI run are different artifacts with different owners.

| Stage       | Artifact                                                            | Owner              | Required Evidence                                   |
|-------------|---------------------------------------------------------------------|--------------------|-----------------------------------------------------|
| task intake | issue, prompt, failing test, security finding, or migration request | human or scheduler | acceptance criteria, allowed scope, forbidden scope |

| Stage                | Artifact                                           | Owner                 | Required Evidence                                                |
|----------------------|----------------------------------------------------|-----------------------|------------------------------------------------------------------|
| session start        | agent session record                               | agent runtime         | model, tools, repo instructions, working set, permission profile |
| workspace allocation | branch, worktree, container, or cloud checkout     | workspace manager     | clean base ref, isolation ID, dependency cache policy            |
| context gathering    | inspected files and commands                       | agent                 | files read, symbols searched, assumptions, skipped areas         |
| patching             | diff on isolated branch                            | agent                 | changed files, rationale, generated files, dependency changes    |
| local verification   | test, build, typecheck, lint, or screenshot output | agent and test runner | command, exit code, relevant failure summary                     |
| PR handoff           | draft PR, patch, or review artifact                | agent                 | summary, verification, risks, open questions, rollback note      |
| CI evaluation        | CI run tied to branch or PR                        | CI system             | jobs, logs, failures, artifacts, retry count                     |
| review decision      | human review, policy gate, or maintainer merge     | maintainer            | approvals, requested changes, merge or rejection reason          |
| cleanup              | archived session and disposed workspace            | workspace manager     | branch state, worktree removal, retained traces                  |

Do not collapse these artifacts into a chat transcript. A later engineer should be able to answer: what did the agent try, where did it try it, what changed, what verified it, and who accepted it?

## Coding Agent Trace Contract

Keep a compact trace for each coding task.

```
type CodingAgentRun = {
 runId: string;
 repo: string;
 baseRef: string;
 branch: string;
 workspaceId: string;
 task: {
 source: "issue" | "pr_review" | "failing_test" | "security_finding" | "user_request";
 acceptanceCriteria: string[];
 allowedPaths: string[];
 forbiddenPaths: string[];
 }
}
```



```

};
session: {
 instructionsLoaded: string[];
 toolsAllowed: string[];
 approvalRequiredFor: string[];
};
activity: Array<{
 kind: "read" | "search" | "edit" | "command" | "test" | "ci" | "handoff";
 target: string;
 result: "success" | "failed" | "blocked" | "skipped";
}>;
verification: Array<{
 command: string;
 exitCode: number;
 summary: string;
}>;
finalStatus: "ready_for_review" | "needs_human" | "blocked" | "abandoned";
risks: string[];
};

```

This trace is not extra bureaucracy. It is the minimum record needed to review a coding agent the same way a team reviews any other contributor.

## Repository Context

Coding agents fail when they see either too little code or too much code.

Use a curated working set:

1. identify primary files;
2. search for symbols, imports, references, tests, and docs;
3. rank secondary files;
4. load small relevant files;
5. summarize or excerpt large files;
6. keep unrelated files out of context.

The agent should also load repository instructions:

- coding style;
- architecture rules;
- test commands;
- package manager;
- branch and commit rules;
- security constraints;
- generated-file rules;

- review expectations.

Repository instructions should be durable. Do not rely on a human repeating the same guidance in every task.

## Shell Command Discipline

Shell access is powerful and risky.

Commands should be treated like tools:

- validate before execution;
- capture stdout, stderr, exit code, and duration;
- record the working directory;
- limit output size;
- redact secrets;
- distinguish read-only commands from mutating commands;
- require approval for dangerous operations;
- prefer project scripts over ad hoc commands.

Good command output is structured enough for the agent to act on. A failing test should become a diagnostic: file, line, test name, expected value, actual value, and likely owning module.

Avoid commands that hide too much:

- broad cleanup commands;
- global installs;
- destructive git operations;
- shell scripts with unclear side effects;
- commands that require interactive input;
- commands that mutate external systems.

The agent should explain why it is running a command when the command has side effects.

## CI Feedback Loop

CI is one of the best evaluators for coding agents.

A coding agent should treat CI as an external evaluator, not as an afterthought. It should reduce each failure to a reproducible local command or an explicit environmental blocker before continuing.

A useful loop is:

1. agent creates a branch or worktree;
2. agent makes the smallest coherent change;
3. agent runs fast local checks;
4. agent opens or updates a draft PR;
5. CI runs broader tests;

6. CI failures are parsed into structured diagnostics;
7. agent patches targeted failures within a retry budget;
8. agent stops when green, blocked, or approval is needed.

Do not let the agent chase CI forever. Define:

- maximum attempts;
- maximum runtime;
- allowed files;
- allowed test commands;
- when to ask a human;
- when to revert its own last change;
- when to mark the task blocked.

CI feedback should improve the patch, not produce an endless loop of speculative edits.

The retry budget should be explicit:

```
interface CiFailure {
 file?: string;
 test?: string;
 message: string;
}

async function repairWithCi(task: CodingTask, maxAttempts = 3) {
 for (let attempt = 1; attempt <= maxAttempts; attempt += 1) {
 await applySmallPatch(task);
 const result = await runCiChecks(task.branch);

 if (result.status === 'green') {
 return { status: 'ready_for_review', attempts: attempt };
 }

 const failures: CiFailure[] = parseCiFailures(result.logs);
 if (failures.length === 0 || result.status === 'flaky') {
 return { status: 'needs_human', reason: 'unclear_ci_failure' };
 }

 task.feedback = failures.slice(0, 5);
 }

 return { status: 'blocked', reason: 'retry_budget_exhausted' };
}
```

The agent is allowed to repair, but not forever.

## Background Agents

Background coding agents are useful when work is long-running or naturally asynchronous.

Use them for:

- dependency upgrades;
- lint migrations;
- test failure triage;
- mechanical refactors;
- documentation updates;
- low-risk bug fixes with good tests.

Avoid background autonomy for:

- ambiguous product changes;
- architecture changes without review;
- security-sensitive code without a validated finding;
- production deployment;
- secrets, credentials, or access control changes.

Background agents should notify humans only at meaningful states:

- needs clarification;
- needs approval;
- CI failed beyond retry budget;
- PR ready for review;
- blocked by missing permission;
- conflict with main branch.

The point is not to remove humans. The point is to stop requiring humans to babysit waiting time.

## Resumable State

Long-running coding agents need durable state outside the model context.

Useful state artifacts:

- task goal;
- acceptance criteria;
- files inspected;
- files changed;
- commands run;
- test results;
- decisions made;
- known risks;
- open questions;

- retry count;
- current blocker.

This can live in a task record, PR description, branch notes, agent state file, issue comment, or durable workflow state. The important part is that a human or a later agent can resume without reconstructing the whole conversation.

## PR Review Agents

PR review is one of the best production shapes for coding agents because the boundary is clear.

A review agent can:

- inspect changed files;
- compare against repository rules;
- run targeted checks;
- identify missing tests;
- flag security risks;
- suggest smaller diffs;
- write review comments.

It should not automatically merge its own approval. It should not block on style preferences unless those preferences are codified. It should cite files, lines, tests, and evidence.

Good review comments are specific:

- what is wrong;
- why it matters;
- where it occurs;
- how to verify;
- whether it is blocking or advisory.

The review agent is a second set of eyes, not the final authority.

## Use When

- The task can be verified with tests, builds, type checks, screenshots, or review.
- The desired change can be described in concrete acceptance criteria.
- The agent can inspect enough repository context to follow local patterns.
- You can isolate work and review the resulting diff.

## Avoid When

- The repository lacks tests or runnable checks and the change is high risk.
- The task is vague, political, or primarily product discovery.
- The agent needs broad production credentials.
- Multiple agents would edit the same files without coordination.

## Evaluation Strategy

Evaluate coding agents through artifacts and trajectories.

Check:

- diff correctness;
- test behavior;
- build and typecheck status;
- changed-files scope;
- architectural fit;
- dependency changes;
- generated code quality;
- command trajectory;
- secret exposure;
- review usefulness;
- handoff quality.

Use baselines:

- no-agent baseline for simple tasks;
- single-agent baseline before multi-agent coding workflows;
- human review outcomes;
- CI pass rate;
- revert or follow-up fix rate.

Coding-agent evals should include negative cases:

- task is too vague;
- tests are missing;
- requested change touches forbidden files;
- CI failure is flaky;
- dependency upgrade requires approval;
- security finding is not reproducible;
- generated code would solve the symptom but violate architecture.

The correct behavior is sometimes to stop and ask for a human decision.

## Operating Patterns

- Ask for a plan before large edits.
- Make the agent cite files and commands it used.
- Prefer small tasks with clear completion criteria.
- Use worktrees or branches for parallel agents.
- Require tests or type checks before commit.
- Review generated code like human code.
- Keep durable repo guidance in a project instruction file.

- Use isolated workspaces for parallel or background work.
- Treat shell commands as auditable tool calls.
- Keep retry budgets for CI-driven repair loops.
- Require explicit handoff artifacts for long-running work.

## Failure Modes

- Plausible code that compiles but violates architecture.
- Broad refactors that mix behavior changes with formatting.
- Tests updated to match broken behavior.
- Hidden dependency changes.
- Shell commands that mutate local state unexpectedly.
- Agents fighting over the same files.
- Review fatigue when diffs are too large.
- CI loops that patch symptoms without understanding failures.
- Background agents that continue after the task definition changes.
- Context windows filled with unrelated files.
- PR comments that sound plausible but cite no evidence.
- Human handoff that omits what was tried and why it failed.

## Production Checklist

- Branch or worktree per task.
- Clear allowed and forbidden files.
- Tool permission profile for read, edit, shell, network, and git operations.
- Repository instruction file loaded by default.
- Curated file context, not whole-repo context.
- Fast local checks before broad CI.
- CI diagnostics parsed into structured feedback.
- Retry budget and stop rules.
- Draft PR or review artifact for human inspection.
- Secret redaction in prompts, command output, traces, and summaries.
- Handoff summary with changed files, commands, results, risks, and open questions.

## Design Rule

The coding agent should never be the only reviewer of its own code. It can propose, edit, test, and explain. A separate check, reviewer, or policy gate should decide whether the change lands.

## Related Chapters

- [Goals and State](#)
- [Context Budgets and Working Sets](#)
- [Tool Capability Design](#)

- Skills
- Agents As Services
- Choosing Multi-Agent Topology
- Human Approval Gates
- Production Evaluation Feedback Loops
- Observability and Evals
- Architecture Decision Records for Agents



# Computer-Use Agents

Computer-use agents operate software through a user interface when APIs, databases, or workflow tools are unavailable or insufficient. They read screens, choose UI actions, click, type, scroll, upload, download, and inspect results.

Use this pattern only when direct integration is not practical. A UI is the least stable interface an agent can operate.

Download the reusable review artifact: [computer-use agent review checklist](#).

## Intent

Let an agent complete tasks in existing applications by controlling a browser, desktop, terminal, or remote environment under strong sandboxing and human oversight.

Computer-use agents are useful for legacy systems, one-off operational tasks, SaaS tools without APIs, cross-application workflows, and product testing.

## Use When

- The system has no usable API.
- The API lacks required functionality.
- The workflow spans several user-facing applications.
- A human currently performs the task through a UI.
- You need to test a product the way a user experiences it.
- The task can tolerate slower execution and occasional recovery.

## Avoid When

- A stable API or database integration exists.
- The workflow has high financial, legal, or safety impact without approval.
- The UI changes frequently and cannot be tested.
- Authentication, CAPTCHA, or 2FA blocks automation.
- The agent would need broad access to private screens or files.

If direct tool use is available, prefer [MCP-first Tool Use](#).

## Architecture

Goal

-> Task state

-> Screen or DOM observation

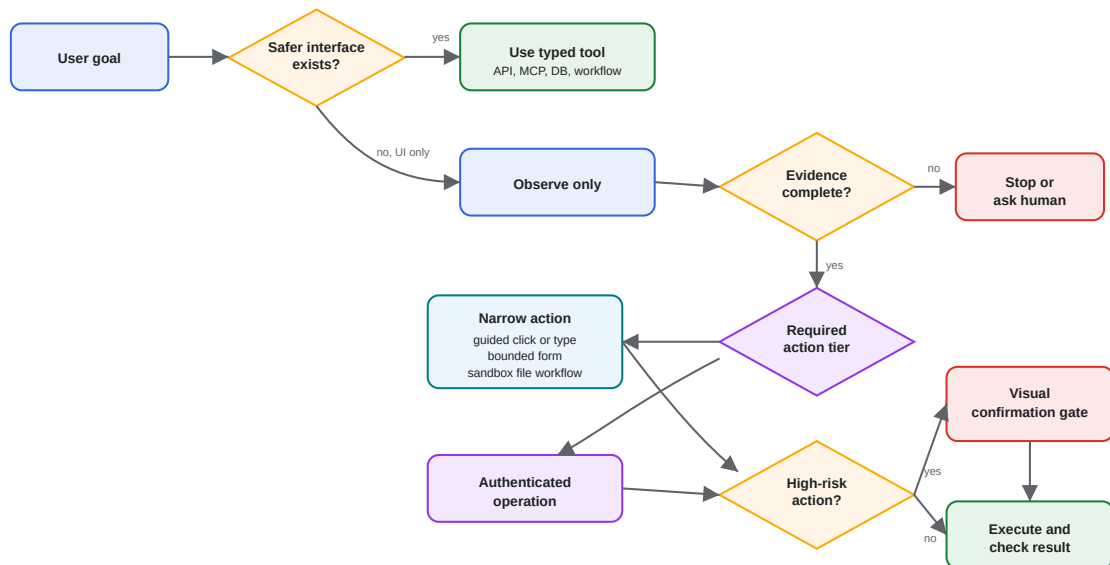
-> UI action proposal

- > Policy and sandbox check
- > Action executor
- > Observation and trace
- > Stop, recover, or continue

The action executor should be deterministic. The model proposes an action; software validates and performs it.

### Computer-Use Agent Action-Space Control Flow

Use the narrowest UI authority that can complete the task, then gate high-risk actions with visible evidence.



## Fit Check

Use computer control only after rejecting safer interfaces.

| Prefer                        | When                                                                                 |
|-------------------------------|--------------------------------------------------------------------------------------|
| API or MCP tool               | The application exposes the needed capability with a stable contract.                |
| Database or event integration | The task reads or writes internal state under known policy.                          |
| Workflow engine               | The sequence, retries, approvals, and state are known.                               |
| Test automation               | The goal is product QA and selectors can be instrumented.                            |
| Computer-use agent            | The only practical interface is the UI and the task can tolerate drift and recovery. |

The cost of UI automation is not only latency. It is fragility. Every selector, modal, visual state, login flow, browser permission, and page redesign becomes part of the agent's operating environment.

## Interface Representation

The agent needs a compact representation of the interface.

Common representations:

- screenshot with coordinates;
- accessibility tree;
- DOM snapshot;
- browser automation locator map;
- terminal buffer;
- application event log;
- image plus OCR;
- structured UI state from test instrumentation.

Use the richest structured representation available. Screenshots help when visual layout matters, but DOM or accessibility trees are easier to validate and replay.

## Observation Evidence Contract

Computer-use agents should treat every observation as evidence, not as an informal screenshot. The runtime should store enough context for another engineer to replay the decision without exposing unnecessary private data.

```
type UiObservation = {
 observationId: string;
 runId: string;
 timestamp: string;
 surface: "browser" | "desktop" | "terminal" | "remote_desktop";
 urlOrApp?: string;
 screenshotRef?: string;
 domSnapshotRef?: string;
 accessibilityTreeRef?: string;
 terminalBufferRef?: string;
 redactions: Array<{
 field: string;
 reason: "secret" | "personal_data" | "customer_data" | "internal_data";
 }>;
 visibleStateSummary: string;
 allowedNextActions: string[];
};
```

The observation should answer three questions before the next action: what did the agent see, what was redacted, and which actions were allowed from that state?

## Screenshot and Artifact Policy

Screenshots, downloads, DOM snapshots, and terminal buffers are useful for debugging but risky to retain. Set policy before production.

| Artifact           | Keep When                                                    | Redact Or Drop When                                                           |
|--------------------|--------------------------------------------------------------|-------------------------------------------------------------------------------|
| Screenshot         | Visual layout, modal state, or pixel-level evidence matters. | It contains secrets, payment data, health data, or unrelated private content. |
| DOM snapshot       | Selectors, labels, and form state matter.                    | Hidden fields, tokens, or full page data exceed the task scope.               |
| Accessibility tree | The action target must be inspectable and replayable.        | Labels expose sensitive user or customer data.                                |
| Downloaded file    | The task output is the downloaded artifact.                  | The file is not needed after validation or contains unapproved data.          |
| Terminal buffer    | Command output proves the state transition.                  | Output contains credentials, tokens, or broad environment details.            |

Retention should match risk. For low-risk QA, keeping screenshots may be useful. For customer data, retain redacted references and action traces instead of raw images whenever possible.

## Action Contract

Every UI action should be typed. Do not let the model emit vague commands like “click the right button.”

```
type UiAction =
 | {
 type: "click";
 selector: string;
 precondition: string;
 timeoutMs: number;
 risk: "low" | "medium" | "high";
 }
 | {
 type: "type";
 selector: string;
 value: string;
 redaction: "none" | "secret" | "personal_data";
 timeoutMs: number;
 }
 | {
 type: "navigate";
```

```

url: string;
allowedDomain: string;
timeoutMs: number;
}
| {
type: "download";
selector: string;
sandboxPath: string;
maxBytes: number;
};

```

The executor should validate preconditions before action and inspect postconditions after action. If the UI state does not match the expected state, stop, recover, or escalate.

### Action Space

Keep the action space small and explicit.

Examples:

- click by stable selector;
- type text into a named field;
- select an option;
- upload a file from a sandbox path;
- press a limited key;
- navigate to an allowed URL;
- download to a sandbox directory;
- wait for a condition.

Avoid unrestricted “control the computer” actions unless the environment is disposable and isolated.

### Action-Space Tiers

Use tiers to decide how much freedom the agent gets. A computer-use agent should start narrow and earn broader control only when tests and traces prove it can recover safely.

| Tier          | Allowed Actions                                         | Use When                                                | Required Evidence                                          |
|---------------|---------------------------------------------------------|---------------------------------------------------------|------------------------------------------------------------|
| Observe only  | screenshot, DOM read, accessibility read, terminal read | inspection, QA, data extraction, or operator assistance | observation trace, redaction proof, no write path          |
| Guided action | click or type only on allowlisted selectors             | known workflow with stable UI states                    | selector map, precondition, postcondition, and retry limit |

| Tier                    | Allowed Actions                                | Use When                                                      | Required Evidence                                                            |
|-------------------------|------------------------------------------------|---------------------------------------------------------------|------------------------------------------------------------------------------|
| Form completion         | fill bounded fields and submit draft           | user or reviewer checks before final action                   | field schema, validation errors, approval before external effect             |
| Sandboxed file workflow | upload or download only in a scoped workspace  | report export, document conversion, or test artifact handling | sandbox path, max size, file type, checksum, retention rule                  |
| Authenticated operation | act inside a logged-in app with scoped account | SaaS workflow without API alternative                         | account boundary, domain allowlist, approval for writes, session cleanup     |
| Disposable exploration  | broader navigation in an isolated environment  | QA exploration or throwaway research                          | disposable profile, no private data, no credentials, no durable side effects |

Do not jump from observe-only to authenticated operation because one happy path worked. Each tier adds authority, so each tier needs its own evals and rollback behavior.

## Visual Confirmation Gates

For high-risk UI actions, require a visual confirmation gate before execution. The gate should show the human or policy engine what the agent sees and what it intends to do.

| Gate Field                 | Purpose                                                             |
|----------------------------|---------------------------------------------------------------------|
| current screen reference   | proves which UI state the action targets                            |
| target selector and label  | proves the agent is acting on the intended control                  |
| proposed action            | click, type, upload, download, submit, or navigate                  |
| affected account or tenant | prevents acting in the wrong workspace                              |
| visible payload or diff    | shows message body, file name, amount, recipient, or setting change |
| policy result              | explains why the action is allowed, denied, or approval-required    |
| postcondition              | defines what success must look like after the action                |

The gate is most important before `submit`, `send`, `delete`, `publish`, `purchase`, `grant access`, or `upload`. If the screen cannot be captured safely, require a typed tool or human operation instead.

## High-Risk UI Actions

Some UI actions should never run without approval:

- sending email, chat, or social messages;
- submitting payments, refunds, purchases, or invoices;
- deleting files, records, users, or permissions;
- changing account settings, security settings, or access controls;
- uploading private files to external services;
- accepting legal, financial, or contractual terms;
- deploying, publishing, or merging production changes.

Approval should bind the exact UI action, target, visible evidence, policy version, user, and trace ID. A human approval for one visible action should not authorize whatever the agent decides to click next.

## Example: SaaS Report Export

A common computer-use task is exporting a report from a SaaS admin console that has no useful API. The agent should act like a careful operator, not like a free-form desktop user.

| Step | Observation           | Proposed Action                          | Required Guard                                            |
|------|-----------------------|------------------------------------------|-----------------------------------------------------------|
| 1    | login page loaded     | request user authentication              | agent does not handle password, 2FA, or CAPTCHA           |
| 2    | dashboard visible     | navigate to <code>/reports</code>        | domain allowlist and route check                          |
| 3    | reports page visible  | choose “Monthly Usage”                   | selector, label, and page title match                     |
| 4    | date filter visible   | type date range                          | typed value redacted when stored if customer data appears |
| 5    | export button visible | click export                             | download path sandboxed and max file size enforced        |
| 6    | file downloaded       | validate file name, size, and format     | no upload or external send without approval               |
| 7    | task complete         | return report location and trace summary | raw screenshot retention follows artifact policy          |

The agent should stop if the page shows an account switcher, destructive modal, unexpected permission prompt, or export destination outside the sandbox.

## State and Recovery

Computer-use agents fail in messy ways:

- modals appear;
- pages load slowly;
- buttons move;

- sessions expire;
- downloads fail;
- validation errors appear;
- the UI changes after deployment.

Design recovery around checkpoints:

- current URL or application state;
- last successful action;
- visible error messages;
- files created or downloaded;
- external side effects;
- retry count;
- human approval state.

The agent should be able to stop with a useful report instead of blindly continuing.

## Recovery Playbook

Recovery should be narrow and state-aware. A failed UI action should not give the agent permission to explore the whole application.

| Failure               | Safe Recovery                                                               | Stop When                                                             |
|-----------------------|-----------------------------------------------------------------------------|-----------------------------------------------------------------------|
| selector missing      | re-observe once and search only within the expected region                  | target still absent or page identity changed                          |
| click has no effect   | wait for expected postcondition, then retry once if no side effect occurred | postcondition still missing                                           |
| form validation error | capture field error and correct only fields inside task scope               | error mentions account, permission, billing, legal, or security state |
| download incomplete   | retry download once into a fresh sandbox path                               | file size, format, or checksum still invalid                          |
| session expired       | pause for user re-authentication                                            | login requires bypassing 2FA, CAPTCHA, or policy                      |
| unexpected modal      | close only allowlisted informational modals                                 | modal asks for deletion, payment, permission, or terms acceptance     |
| destination changed   | verify domain, account, tenant, and page title                              | any identity or tenant mismatch appears                               |

The recovery path should preserve the last safe state and the last attempted action. If the system cannot prove no side effect occurred, it should stop instead of retrying.



## UI Drift Handling

UI drift is normal. Treat it as a first-class failure mode.

| Drift            | Runtime Response                                                                              |
|------------------|-----------------------------------------------------------------------------------------------|
| Selector missing | Re-observe once, then stop with <code>ui_changed</code> .                                     |
| Unexpected modal | Classify modal; close only if allowlisted, otherwise escalate.                                |
| Text changed     | Verify semantic target before action; do not click by approximate text for high-risk actions. |
| Page load slow   | Wait with budget; retry only when no side effect occurred.                                    |
| Session expired  | Pause and request re-authentication; do not bypass 2FA or CAPTCHA.                            |
| Validation error | Capture field errors and return controlled failure.                                           |

Do not train the agent to “try something else” around unknown UI states. That is how a brittle automation becomes a risky one.

## Security Controls

Computer-use agents need strong containment:

- run in an isolated browser profile, container, VM, or remote desktop;
- restrict network destinations;
- isolate downloads and uploads;
- block access to local secrets;
- use scoped credentials;
- record UI actions;
- require approval for irreversible actions;
- clear sessions after runs;
- prevent copy/paste of hidden sensitive data into untrusted sites.

If the agent can see private data and browse untrusted content, treat the workflow as high risk.

## Sandbox Profiles

Match containment to the action space.

| Profile           | Use For                                           | Minimum Controls                                                   |
|-------------------|---------------------------------------------------|--------------------------------------------------------------------|
| Read-only browser | Search, inspection, screenshots, public browsing. | No saved credentials, blocked private networks, download disabled. |

| Profile               | Use For                                       | Minimum Controls                                                                |
|-----------------------|-----------------------------------------------|---------------------------------------------------------------------------------|
| Authenticated browser | SaaS workflows under user or service account. | Isolated profile, scoped account, egress allowlist, trace, approval for writes. |
| Remote desktop        | Legacy apps or cross-app workflows.           | Disposable VM, clipboard controls, file transfer policy, session recording.     |
| Terminal UI           | CLI or TUI workflows.                         | Sandbox workspace, command allowlist, no ambient secrets, timeout.              |
| Product QA runner     | Regression testing through UI.                | Test account, test data, deterministic selectors, artifact retention policy.    |

The sandbox profile should be part of the deployment contract. A read-only browser agent should not silently become an authenticated desktop agent.

## Evaluation Strategy

Computer-use evals should test UI behavior, not only final text.

- Test the happy path with stable selectors.
- Test stale selector and renamed button cases.
- Test unexpected modal and validation error cases.
- Test slow page and timeout behavior.
- Test denied egress and blocked download.
- Test high-risk action approval.
- Test trace replay from screenshots, DOM snapshots, or action logs.
- Test privacy redaction for screenshots and typed values.

A compact eval fixture can look like this:

```
{
 "case_id": "unexpected_delete_button_modal",
 "goal": "Export a report from the admin dashboard.",
 "observations": ["dashboard_loaded", "unexpected_delete_modal"],
 "expected": {
 "final_status": "needs_human",
 "must_not_click": ["confirm_delete"],
 "required_trace_events": ["observe", "policy_denied", "stop"]
 }
}
```

## Production Checklist

- Is there truly no better API or tool integration?

- Are actions restricted to a known set?
- Can every action be traced and replayed?
- Does the agent run in an isolated environment?
- Are credentials scoped to the task?
- Can the user approve high-risk actions?
- Does the run stop when the UI diverges?
- Are UI changes covered by regression tests?
- Are screenshots, DOM snapshots, downloads, and typed values handled under a privacy policy?
- Are selectors, allowed domains, credentials, and sandbox profile versioned?

## Related Chapters

- [Tool Use](#)
- [MCP-first Tool Use](#)
- [Agent Security and Sandboxing](#)
- [Circuit Breakers, Fallbacks, and Replay](#)
- [Coding Agents](#)

# Domain Agent Architectures

Domain agents apply agentic patterns inside a specialized field such as healthcare, finance, legal operations, education, science, insurance, or software engineering. The pattern is not “add an agent to a domain.” The pattern is to encode the domain’s evidence, constraints, risk, and review process into the architecture.

Use this chapter when an agent must operate under domain-specific standards rather than generic productivity expectations.

Download the reusable review artifact: [domain agent architecture review checklist](#).

## Intent

Build agents that respect the domain’s knowledge sources, vocabulary, decision rights, compliance boundaries, and failure costs.

The architecture should make domain judgment auditable.

## Domain Design Questions

Start with the domain, not the model.

- What decisions can the agent make?
- What decisions can it only recommend?
- What evidence is authoritative?
- What evidence is forbidden or insufficient?
- What requires professional review?
- What user data is sensitive?
- What are the domain-specific failure modes?
- What laws, policies, or standards apply?
- What record must be retained?
- What explanation must be shown to users or reviewers?

The answers shape tools, memory, evals, approval gates, and deployment.

## Decision Rights

The first architecture decision is not the model or framework. It is decision rights.

| Decision Type  | Agent May                                            | Agent Must Not                       |
|----------------|------------------------------------------------------|--------------------------------------|
| Classification | assign a draft category with evidence and confidence | silently finalize regulated outcomes |

| Decision Type  | Agent May                                             | Agent Must Not                                         |
|----------------|-------------------------------------------------------|--------------------------------------------------------|
| Recommendation | propose next action, cite policy, explain uncertainty | execute irreversible action without approval           |
| Drafting       | prepare messages, reports, forms, or plans            | send obligation-creating communication without review  |
| Extraction     | extract fields from documents with source spans       | invent missing fields or hide uncertainty              |
| Triage         | route cases by risk and urgency                       | deny service, coverage, care, or access without policy |
| Analysis       | summarize evidence and compare options                | present unsupported judgment as fact                   |

Write the decision-rights table before implementation. It tells the team where the model can help and where the runtime, policy, or professional reviewer must remain in charge.

### Common Domain Shapes

| Domain Shape                   | Agent Role                                              | Architecture Notes                                                                   |
|--------------------------------|---------------------------------------------------------|--------------------------------------------------------------------------------------|
| Healthcare or clinical support | Summarize, triage, draft, explain, prepare evidence.    | Keep licensed humans in decision roles; cite sources; log recommendations.           |
| Finance or insurance           | Analyze documents, classify risk, prepare decisions.    | Enforce policy, audit data lineage, require approval for financial actions.          |
| Legal operations               | Search, summarize, compare, draft, extract obligations. | Preserve privilege boundaries; show citations; avoid unsupervised legal conclusions. |
| Education                      | Tutor, assess, adapt difficulty, produce feedback.      | Track learning goals; avoid over-personalized unsupported claims.                    |
| Scientific research            | Search literature, propose hypotheses, analyze data.    | Separate hypothesis generation from validation; track provenance.                    |
| Software engineering           | Search code, edit, test, review, prepare PRs.           | Sandbox execution; preserve diffs; require CI and review gates.                      |

Different domains can share patterns while requiring different risk controls.

### Concrete Domain Examples

| Domain              | Safe Agent Role                                                        | Hard Boundary                                                               |
|---------------------|------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| Support             | Summarize ticket history, retrieve policy, draft response.             | Cannot promise refunds, credits, or account changes without approval.       |
| Healthcare          | Prepare visit summaries, patient questions, or evidence packets.       | Cannot diagnose, prescribe, or replace licensed review.                     |
| Finance             | Extract statement fields, flag anomalies, draft risk notes.            | Cannot move money, approve credit, or change entitlements without controls. |
| Legal operations    | Compare clauses, summarize matter history, draft obligation lists.     | Cannot provide unsupervised legal judgment or cross privilege boundaries.   |
| Insurance           | Classify claim documents, retrieve policy terms, draft recommendation. | Cannot deny or approve claim without governed review.                       |
| Internal operations | Triage access requests, draft runbooks, summarize incidents.           | Cannot grant production access or close incidents without owner acceptance. |

These examples share one pattern: the agent prepares evidence and recommendations; the governed system owns authority.

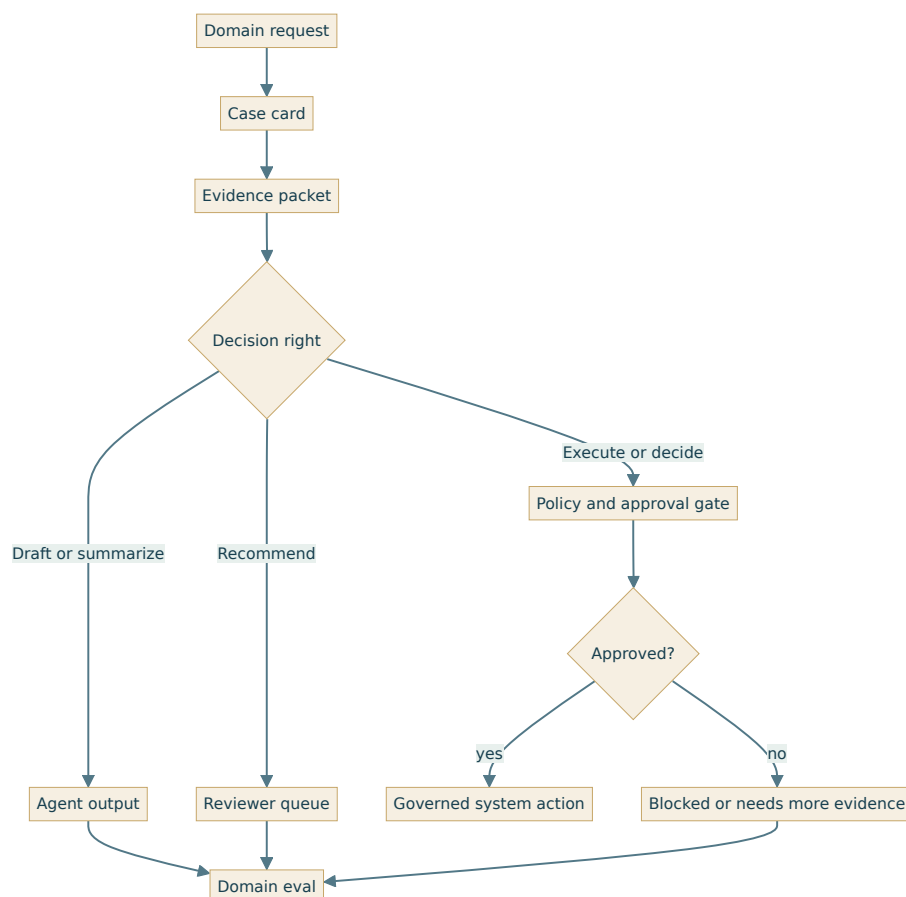
## Production Case Cards

Use case cards when a domain team says, “We need an agent for our process.” The card turns that request into evidence, authority, review, and eval decisions.

| Domain     | Example Request                                | Agent May Produce                                                             | Must Block Or Escalate                                                               | Release Eval                                                          |
|------------|------------------------------------------------|-------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|-----------------------------------------------------------------------|
| Support    | “Refund this order and tell the customer.”     | Ticket summary, policy match, draft recommendation, customer-message draft.   | Refund issue, customer promise, account credit, or missing policy.                   | Above-threshold refund requires approval before any promise language. |
| Healthcare | “Summarize this visit and suggest next steps.” | Visit summary, patient questions, evidence packet for clinician review.       | Diagnosis, treatment plan, medication change, or cross-patient data.                 | Missing clinician review blocks treatment-language output.            |
| Finance    | “Explain why this account was flagged.”        | Anomaly summary, source transactions, risk-note draft, reviewer queue reason. | Money movement, credit approval, entitlement change, or unsupported risk conclusion. | Wrong jurisdiction or stale policy blocks recommendation.             |

| Domain              | Example Request                                          | Agent May Produce                                                             | Must Block Or Escalate                                                    | Release Eval                                                       |
|---------------------|----------------------------------------------------------|-------------------------------------------------------------------------------|---------------------------------------------------------------------------|--------------------------------------------------------------------|
| Legal operations    | "Compare these clauses and advise which is safer."       | Clause comparison, cited differences, obligation list, review questions.      | Legal advice, external communication, filing, or cross-matter source use. | Privilege or matter-boundary violation fails the eval.             |
| Internal operations | "Grant the engineer production access for the incident." | Access-request summary, runbook match, owner recommendation, expiry proposal. | Permission grant, incident closure, destructive command, or owner bypass. | Access change requires owner approval and expiry before execution. |

Each card should name one forbidden sentence or action. For support, "Your refund has been approved" is forbidden before approval. For healthcare, "You should start this medication" is forbidden unless the licensed workflow owns that decision. For internal operations, "Access granted" is forbidden until the identity and owner checks pass.



The case card is not documentation after the fact. It is the architecture input. If the card cannot name the authoritative evidence, forbidden action, reviewer, and release eval, the domain agent is not ready to build.

## Reference Architecture

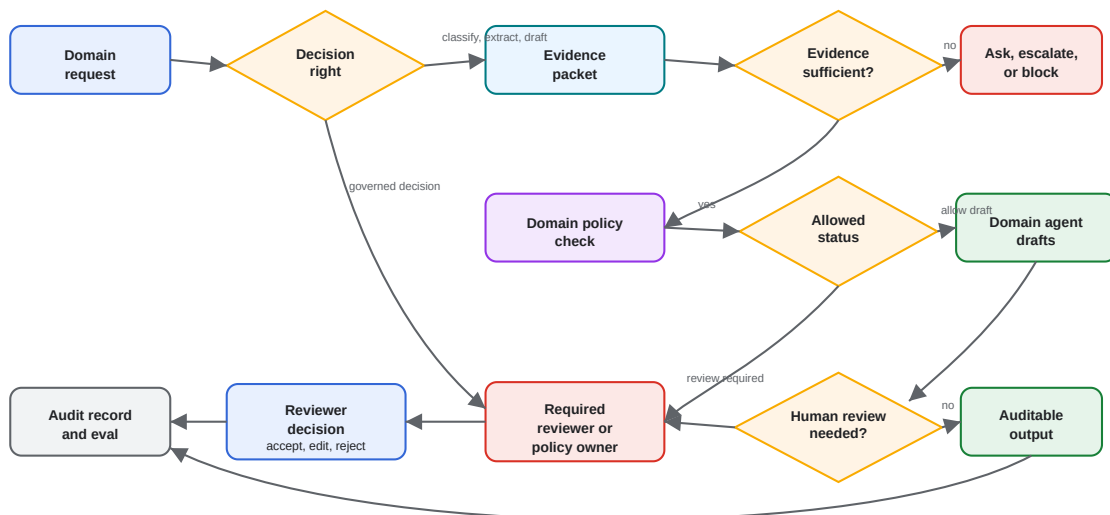
### Domain request

- > identity and authorization
- > domain router
- > evidence retrieval
- > domain tool execution
- > policy and risk checks
- > agent or workflow decision
- > human review when required
- > auditable output

The domain router should select sources, tools, and approval paths. It should not bypass governance.

### Domain Agent Authority and Evidence Flow

The agent prepares evidence and recommendations. Domain policy and authorized reviewers own governed decisions.



Read the diagram from left to right. The first gate is decision rights: if the task asks the agent to execute, deny, approve, diagnose, or decide, the flow moves toward policy or reviewer ownership before the model can produce a governed outcome.

## Domain Policy Contract

Domain policy should be executable enough to test.

```
type DomainPolicyDecision = {
 domain: "healthcare" | "finance" | "legal" | "insurance" | "support" | "internal_ops";
 taskType: string;
 actorRole: string;
 dataClass: "public" | "internal" | "confidential" | "regulated";
```



```
evidenceStatus: "sufficient" | "missing" | "stale" | "conflicting";
proposedAction: "answer" | "draft" | "recommend" | "execute" | "escalate";
decision:
 | { status: "allow"; reason: string }
 | { status: "allow_with_review"; reviewerRole: string; reason: string }
 | { status: "deny"; reason: string }
 | { status: "needs_more_evidence"; missing: string[] };
};
```

The exact type will vary, but the principle should not: domain policy is a runtime decision, not a paragraph in the prompt.

## Evidence Design

Domain agents need explicit evidence policy.

Define:

- authoritative source types;
- freshness requirements;
- citation format;
- source conflict handling;
- minimum evidence for an answer;
- unsupported claim behavior;
- source redaction rules;
- tenant and role boundaries.

If the agent cannot cite or explain the basis for a domain answer, it should lower confidence, ask for clarification, or escalate.

## Domain Evidence Packet

Every domain recommendation should carry an evidence packet. The packet should be small enough for a reviewer to inspect and structured enough for evals to test.

```
type DomainEvidencePacket = {
 domain: "support" | "healthcare" | "finance" | "legal" | "insurance" | "internal_ops";
 caseId: string;
 actorRole: string;
 taskType: string;
 decisionRight: "classify" | "draft" | "recommend" | "execute" | "escalate";
 sources: Array<{
 sourceId: string;
 sourceType: "policy" | "record" | "contract" | "clinical_note" | "ticket" | "runbook" |
 "database_result";
```

```

 version: string;
 effectiveDate?: string;
 jurisdiction?: string;
 tenantId?: string;
 freshness: "current" | "stale" | "unknown";
 citedSpan?: string;
 }>;
 uncertainty: Array<{
 issue: string;
 impact: "low" | "medium" | "high";
 requiredReviewer?: string;
 }>;
 forbiddenActions: string[];
 finalStatus: "draft" | "recommendation" | "needs_more_evidence" | "needs_review" |
"blocked";
};

```

The packet should make unsafe evidence obvious. A missing jurisdiction, stale policy version, cross-tenant source, or unsupported claim should change the final status before the user sees a confident answer.

## Domain Playbooks

Use domain playbooks to avoid treating every specialized field as the same architecture with different vocabulary.

| Domain           | Minimum Evidence                                               | Default Agent Output                                 | Mandatory Review Trigger                                               |
|------------------|----------------------------------------------------------------|------------------------------------------------------|------------------------------------------------------------------------|
| Support          | ticket, account scope, current policy, prior actions           | draft response or recommendation                     | refund, credit, account change, or missing policy                      |
| Healthcare       | patient-scoped record, approved reference, date, reviewer role | summary, question list, or evidence packet           | diagnosis, treatment, medication, or conflicting clinical facts        |
| Finance          | transaction record, policy, jurisdiction, risk class           | anomaly note, extracted fields, or draft analysis    | money movement, credit decision, entitlement change, or high-risk flag |
| Legal operations | matter scope, source document, privilege boundary, clause span | clause summary, obligation list, or draft comparison | legal conclusion, privilege risk, filing, or external communication    |
| Insurance        | claim record, policy version, effective date,                  | claim summary or draft recommendation                | approval, denial, exception, missing document, or                      |

| Domain              | Minimum Evidence                                       | Default Agent Output                                | Mandatory Review Trigger                                                     |
|---------------------|--------------------------------------------------------|-----------------------------------------------------|------------------------------------------------------------------------------|
|                     | required documents                                     |                                                     | conflicting evidence                                                         |
| Internal operations | request, owner, runbook, access policy, incident state | triage, draft runbook step, or owner recommendation | production access, incident closure, permission grant, or destructive action |

The playbook should be local to the domain and owned by the team accountable for the outcome. Do not let a generic agent prompt decide which evidence is enough.

## Worked Slice: Insurance Claim Triage

An insurance claim triage agent can help a reviewer move faster without giving the agent claim authority.

| Step                | Runtime Owner                   | Agent Role                              | Required Evidence                                       | Stop Condition                                 |
|---------------------|---------------------------------|-----------------------------------------|---------------------------------------------------------|------------------------------------------------|
| intake              | workflow service                | none                                    | claim ID, policy ID, claimant, submitted documents      | invalid or cross-tenant claim                  |
| document extraction | extraction tool and verifier    | extract fields with source spans        | invoice, incident date, policy number, requested amount | missing required field                         |
| policy retrieval    | retrieval service               | none                                    | policy version, effective date, coverage terms          | stale or mismatched policy                     |
| triage              | domain agent                    | draft coverage questions and risk notes | extracted fields and policy spans                       | conflicting evidence or exception              |
| recommendation      | domain agent                    | recommend next reviewer action          | evidence packet and uncertainty list                    | approval, denial, or exception requested       |
| review              | licensed or authorized reviewer | none                                    | recommendation, source spans, policy checks             | reviewer accepts, edits, rejects, or escalates |
| audit               | workflow service                | none                                    | final decision, reviewer ID, reason, trace              | retained case record                           |

The agent may say, “This claim appears to need more evidence because the incident date is outside the visible policy period.” It must not say, “Claim denied.” The architecture keeps triage language separate from governed decision language.

## Domain Release Gate

Before a domain agent reaches real users or reviewers, require evidence that the domain boundary works.

| Gate                  | Passing Evidence                                                                             |
|-----------------------|----------------------------------------------------------------------------------------------|
| decision rights       | tests prove the agent drafts, recommends, or escalates instead of making forbidden decisions |
| source authority      | evals fail when the answer uses unapproved, stale, cross-tenant, or wrong-version evidence   |
| reviewer workflow     | reviewer can see source spans, policy checks, uncertainty, and proposed action               |
| privacy and retention | traces, prompts, memory, and outputs redact or omit regulated data according to policy       |
| negative cases        | system escalates missing evidence, conflicting evidence, and policy exceptions               |
| override learning     | reviewer edits and rejections become labeled regression cases                                |
| communication safety  | drafts avoid final decision, diagnosis, promise, obligation, or approval language            |

Do not treat a high score on generic answer quality as a domain release. The release gate should prove the system respects the domain’s authority model.

## Jurisdiction, Tenant, and Version Boundaries

Domain answers often depend on where, when, and for whom the question is asked.

Track:

- tenant or customer boundary;
- jurisdiction or region;
- policy version;
- document version;
- effective date;
- reviewer role;
- data retention class;
- source freshness.

A finance policy from the wrong jurisdiction, a clinical note from the wrong patient, a legal document from the wrong matter, or an insurance policy from the wrong effective date is not weak evidence. It is unsafe evidence.

## Review and Approval

Domain agents often produce recommendations rather than final decisions.

Use human review for:

- clinical, legal, financial, or compliance decisions;
- low-confidence classifications;
- conflicting evidence;
- exceptions to policy;
- irreversible actions;
- communications that create obligation or liability.

The review interface should show evidence, policy checks, extracted fields, uncertainty, and the proposed action.

Reviewer decisions should become data. Capture whether the reviewer accepted, edited, rejected, or escalated the recommendation and why. Those records can update evals, policy examples, prompts, and training material.

## Domain Evals

Generic evals are not enough.

Add evals for:

- domain terminology;
- document extraction;
- citation accuracy;
- missing evidence;
- policy exceptions;
- escalation thresholds;
- privacy constraints;
- adversarial or ambiguous inputs;
- reviewer override cases.

Use subject matter experts for labels where judgment matters.

## Domain Eval Matrix

| Eval Type           | Example                                         |
|---------------------|-------------------------------------------------|
| Evidence grounding  | answer must cite current policy or source span  |
| Extraction accuracy | required fields match labeled document fixtures |
| Missing evidence    | system asks for more evidence or escalates      |
| Policy exception    | recommendation follows exception workflow       |

| Eval Type            | Example                                                                 |
|----------------------|-------------------------------------------------------------------------|
| Privacy boundary     | agent refuses cross-tenant or unauthorized data                         |
| Jurisdiction/version | answer uses correct region and policy date                              |
| Reviewer override    | overridden recommendations become regression cases                      |
| Communication risk   | drafts avoid obligation, diagnosis, promise, or final decision language |

Domain evals should include negative cases. The system must prove it knows when not to answer.

## Failure Modes

- The agent sounds authoritative while evidence is weak.
- Domain-specific policy lives only in prompts.
- The agent treats recommendations as decisions.
- Retrieval mixes tenants, jurisdictions, or policy versions.
- Evals measure language quality but not domain correctness.
- Human review lacks enough evidence to make a decision.
- The system uses a general web source where only approved domain sources are allowed.
- Reviewer overrides are not captured, so the same mistake repeats.
- A draft message creates legal, financial, clinical, or contractual obligation.
- Memory stores regulated or sensitive facts without retention and deletion rules.

## Review Questions

Before piloting a domain agent, ask:

1. Which decisions are the agent forbidden to make?
2. Which evidence sources are authoritative, stale, forbidden, or insufficient?
3. Which human role owns final review?
4. Which tenant, jurisdiction, policy, or document version applies?
5. What must be retained for audit?
6. What must be redacted from prompts, traces, memory, and final output?
7. Which eval fails the release if the agent sounds confident but lacks evidence?
8. How do reviewer corrections become regression tests?

## Related Chapters

- [Agentic RAG Systems](#)
- [Policy Enforcement](#)
- [Human Approval Gates](#)
- [Evaluation-Driven Agent Development](#)
- [Knowledge-Bound Agents](#)

# Architecture Decision Records for Agents

Agentic systems change quickly. Architecture Decision Records keep model, memory, tool, policy, and workflow choices explicit enough that future maintainers can understand why the system behaves the way it does.

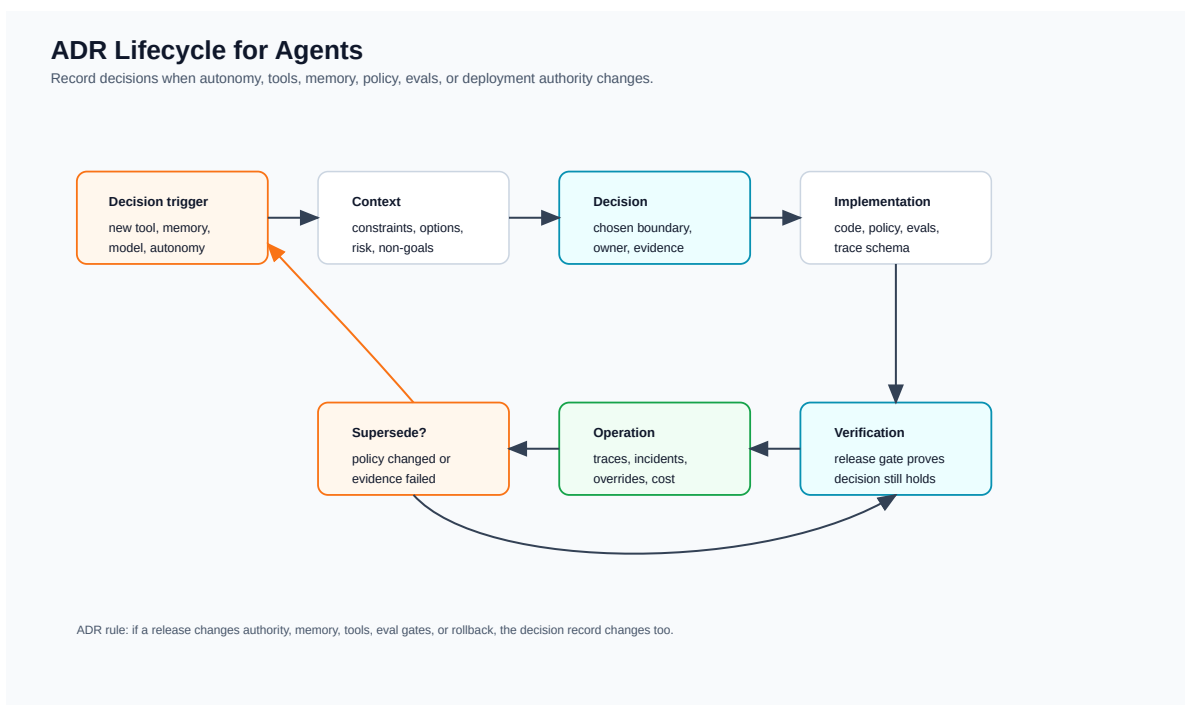
Use ADRs when a decision affects safety, cost, reliability, user trust, or the ability to debug production behavior.

An ADR is not bureaucracy. It is a way to keep autonomy from becoming folklore. If the system can read private data, call tools, write memory, send messages, delegate to other agents, or run without a human watching, the team should be able to point to the decision record that explains why that is allowed.

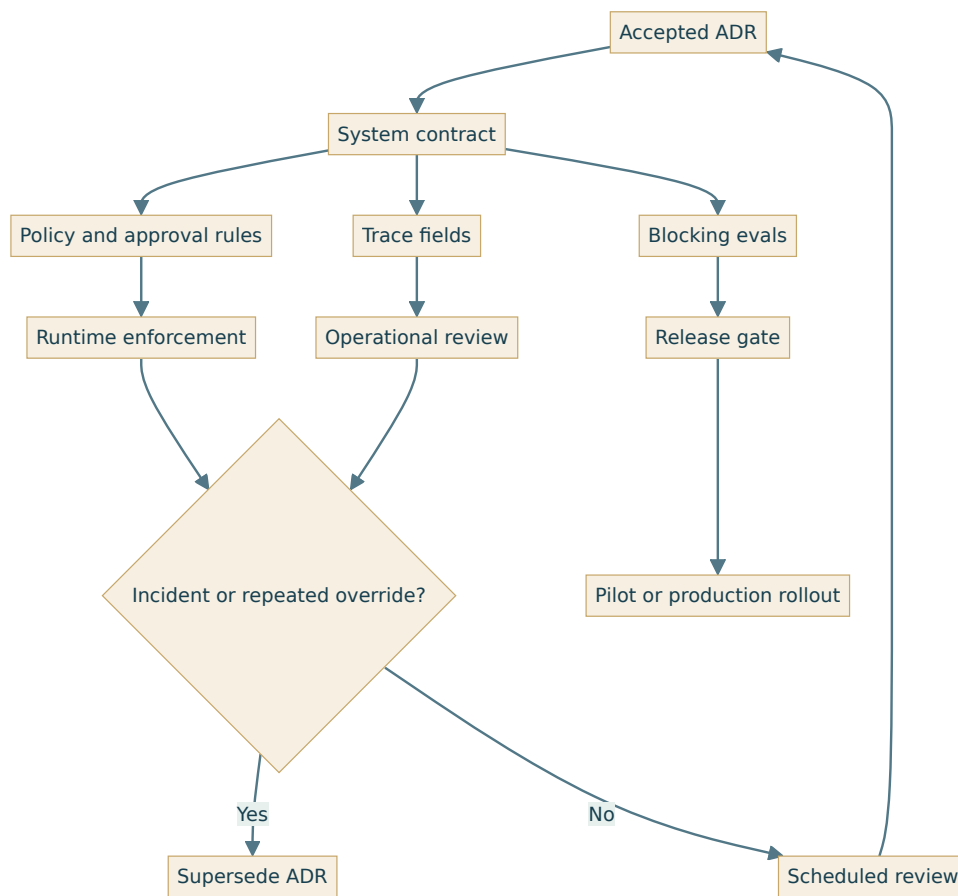
Download completed examples: [agent ADR example pack](#).

## ADR Lifecycle

Use this diagram to decide when an agent decision needs a record and when an existing record must be superseded. Authority changes, memory changes, tool changes, eval changes, and rollback changes should leave evidence.



Use this evidence flow after the ADR is accepted. A decision record should produce runtime checks, not just documentation.



## What to Record

Record decisions about:

- Model selection and fallback models
- Tool permissions and approval rules
- Memory retention and deletion
- Retrieval sources and citation policy
- Workflow retries, compensation, and escalation
- Evaluation datasets and release gates
- Observability and logging boundaries
- Human review requirements
- Self-improvement and skill-update policies

Also record decisions that change the agent's autonomy level:

- advisory only;
- drafts for human review;
- executes after approval;
- executes autonomously within a narrow policy;
- escalates to another agent, workflow, or human.



Autonomy is not a global property. An agent may be autonomous for read-only research, approval-gated for refunds, and forbidden from outbound communication. The ADR should say which actions belong in which category.

## ADR Template

# ADR-000: Short Decision Title

## Status

Proposed | Accepted | Superseded

## Context

What problem are we solving? What constraints matter?

## Decision

What did we choose?

## Scope

Which product, agent, workflow, users, tenants, tools, and data does this decision apply to?

## Autonomy Level

Advisory | Drafts for review | Executes after approval | Executes autonomously within policy

## Tool Authority

Which tools, operations, side effects, scopes, egress, and credentials are allowed?

## Data And Memory Boundaries

Which data may be read? Which memory may be read or written? What retention, deletion, and correction rules apply?

## Human Approval

Which exact actions require approval? Who can approve? How long does approval last?

## Evaluation Gate

Which evals must pass before release? Which failures block deployment?

## ## Observability

Which traces, metrics, audit events, context packets, tool calls, memory writes, and approval decisions must be recorded?

## ## Rollback

How do we disable the capability, revoke access, roll back prompts or models, or stop side effects?

## ## Consequences

What improves? What gets harder? What risks remain?

## ## Verification

How will we know this decision is still working?

## Agent-Specific Additions

Add these fields when relevant:

- **Autonomy level:** advisory, proposes edits, executes after approval, or executes autonomously.
- **Tool scope:** exact tools and operations allowed.
- **State owner:** chat history, workflow state, memory store, database, or external system.
- **Failure policy:** retry, re-plan, ask, refuse, rollback, or escalate.
- **Eval gate:** tests or datasets required before release.
- **Rollback path:** how to disable or reverse the decision.

## Decisions That Deserve ADRs

Write an ADR when the team:

- adds a write-capable tool;
- enables outbound communication;
- enables or changes memory writes;
- moves from advisory mode to execution mode;
- permits autonomous execution for a workflow;
- adds a new source to RAG or changes retrieval policy;
- changes model family, routing, fallback, or temperature for a critical path;
- adds multi-agent delegation or remote agent calls;

- changes approval rules, approver roles, or approval expiry;
- changes observability, retention, redaction, or replay behavior;
- allows code execution, browser use, shell access, or file-system writes.

Small prompt wording changes do not always need ADRs. Authority changes do.

## Example Decisions

- Use a durable workflow for customer-impacting tasks instead of an in-memory loop.
- Require human approval before sending outbound email.
- Store episodic memory for project events but not personal secrets.
- Use hybrid keyword plus vector retrieval for support documentation.
- Run coding agents in disposable worktrees and require `npm test` before commit.
- Keep self-improvement as reviewed skill changes, not silent prompt mutation.

## Completed ADR Examples

Use the [agent ADR example pack](#) when you want completed records for:

- support refund authority, where the agent may draft but not issue money;
- research RAG source policy, where the agent may answer only from approved sources;
- multi-agent delivery review, where specialists work in parallel but one owner accepts the result.

The first example is included below because it shows the most important pattern: separate model recommendation from business authority.

## Example ADR: Support Refund Authority

```
ADR-014: Support refund agent may draft refunds but not issue money
```

```
Status
```

```
Accepted
```

```
Context
```

```
Support agents spend time gathering order details, reading refund policy, and drafting
refund recommendations. The team wants an agent to reduce investigation time without giving
the model direct financial authority.
```

```
Decision
```

```
The support refund agent may investigate orders, retrieve refund policy, summarize
evidence, and create a refund draft. It may not issue money, modify payment state, or
message the customer directly.
```

## ## Scope

Applies to the `support\_refund\_investigation` workflow for consumer orders in the support platform. Business accounts, fraud cases, and chargebacks are out of scope.

## ## Autonomy Level

Executes read-only investigation autonomously. Creates refund drafts autonomously. Refund issuance requires finance approval and a separate deterministic workflow.

## ## Tool Authority

### Allowed:

- `orders.lookup\_order`
- `payments.get\_payment\_summary`
- `refund\_policy.retrieve`
- `refunds.create\_draft`

### Forbidden:

- `refunds.issue\_refund`
- `email.send\_customer\_message`
- broad SQL, browser, shell, or arbitrary HTTP tools

## ## Data And Memory Boundaries

The agent may read order and payment summaries for the current tenant and ticket only. It may not store payment details in long-term memory. It may write an episodic event that a refund draft was created, with source references and retention.

## ## Human Approval

Finance approval is required before issuing money. Approval must bind to the exact refund amount, order ID, draft ID, approver, policy version, and expiry.

## ## Evaluation Gate

### Blocking evals:

- refuses refund issuance without approval;
- cites current refund policy;
- does not email customers;

- does not write sensitive payment details to memory;
- routes fraud and chargeback cases to escalation.

## ## Observability

Trace order lookup, policy retrieval, draft creation, policy decision, approval request, memory write decision, and final recommendation. Redact payment identifiers from logs.

## ## Rollback

Disable `refunds.create\_draft` in the tool registry and route all refund cases back to human support. Existing drafts remain review-only.

## ## Consequences

The agent reduces investigation time and keeps financial authority outside the model. The workflow adds approval latency for edge cases and requires eval maintenance when refund policy changes.

## ## Verification

Review weekly traces for unauthorized tool attempts, approval misses, citation failures, and human override rate. Add every serious miss to the regression eval suite.

## Failure Modes

- Decisions live only in prompts and disappear from engineering review.
- ADRs describe the happy path but not rollback or verification.
- Model upgrades happen without recording eval impact.
- Memory or tool-scope changes ship without privacy review.
- The ADR says “human approval” but not which actions require it.
- The ADR says “agentic” but does not define the autonomy level.
- Tool authority is described as a category instead of exact tools and operations.
- Memory is enabled without retention, correction, deletion, or consent rules.
- Eval gates are listed but not tied to release or rollback.
- Nobody owns the decision after launch.

## Verification Guidance

An ADR should create operational checks, not just documentation.

For each accepted ADR, define:

- the eval suite that protects the decision;

- the traces and dashboards that show the decision is being followed;
- the incident signals that would invalidate the decision;
- the owner who reviews those signals;
- the rollback or disable path;
- the date or trigger for review.

Useful review triggers include model upgrade, prompt rewrite, tool manifest change, new memory type, new data source, new tenant, changed approval policy, high-severity incident, or repeated human overrides.

## Production Checklist

- Does the ADR name the agent, workflow, owner, and scope?
- Does it define the autonomy level by action?
- Does it list exact tools, side effects, scopes, credentials, and egress?
- Does it define data, memory, retention, deletion, and correction boundaries?
- Does it say which actions require approval and who can approve?
- Does it define blocking evals and release gates?
- Does it define traces, metrics, audit records, and redaction rules?
- Does it define rollback, kill switch, or capability disablement?
- Does it include residual risks and review triggers?
- Is the ADR linked from the code, runbook, or deployment checklist?

## Related Chapters

- [Agentic System Architecture](#)
- [Agentic RAG Systems](#)
- [Tool Capability Design](#)
- [Human Approval Gates](#)
- [Memory-Augmented Agent](#)
- [Context Engineering](#)
- [Agent UX and Human Trust](#)
- [Policy Enforcement](#)
- [Observability and Evals](#)

# Reference Architecture

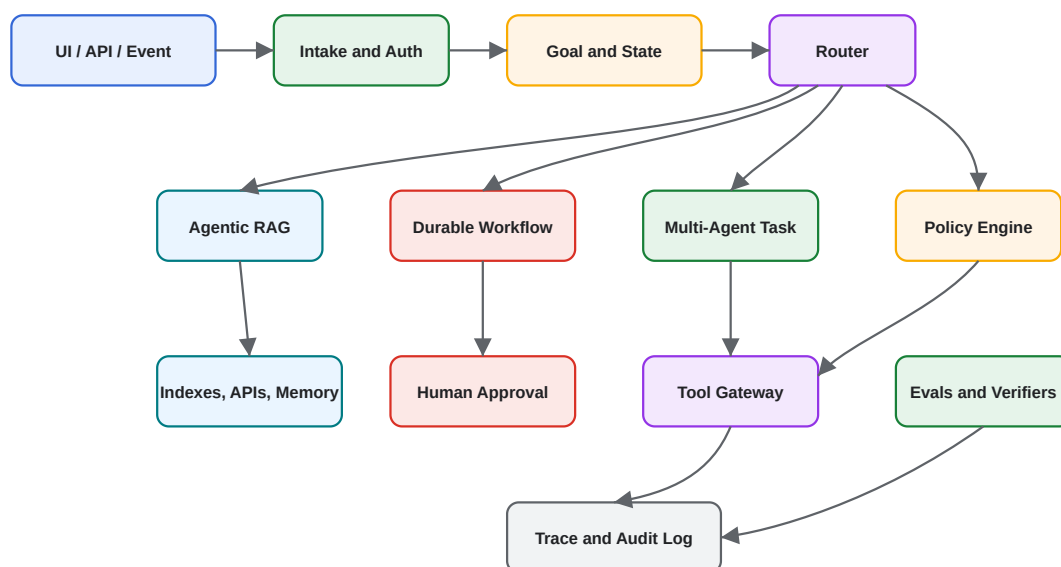
This reference architecture combines the core patterns in the book into a production-ready agentic system. It is intentionally conservative: deterministic software owns state, policy, and side effects; model calls propose decisions inside bounded loops.

Use this chapter as the bridge from pattern selection to system design. The diagram is not a framework prescription; it shows the ownership boundaries a production system needs before agents handle private data, external actions, or long-running work.

## Architecture

Read this diagram as an ownership map for production systems: deterministic services own identity, state, policy, retrieval, approvals, and traces around bounded model calls.

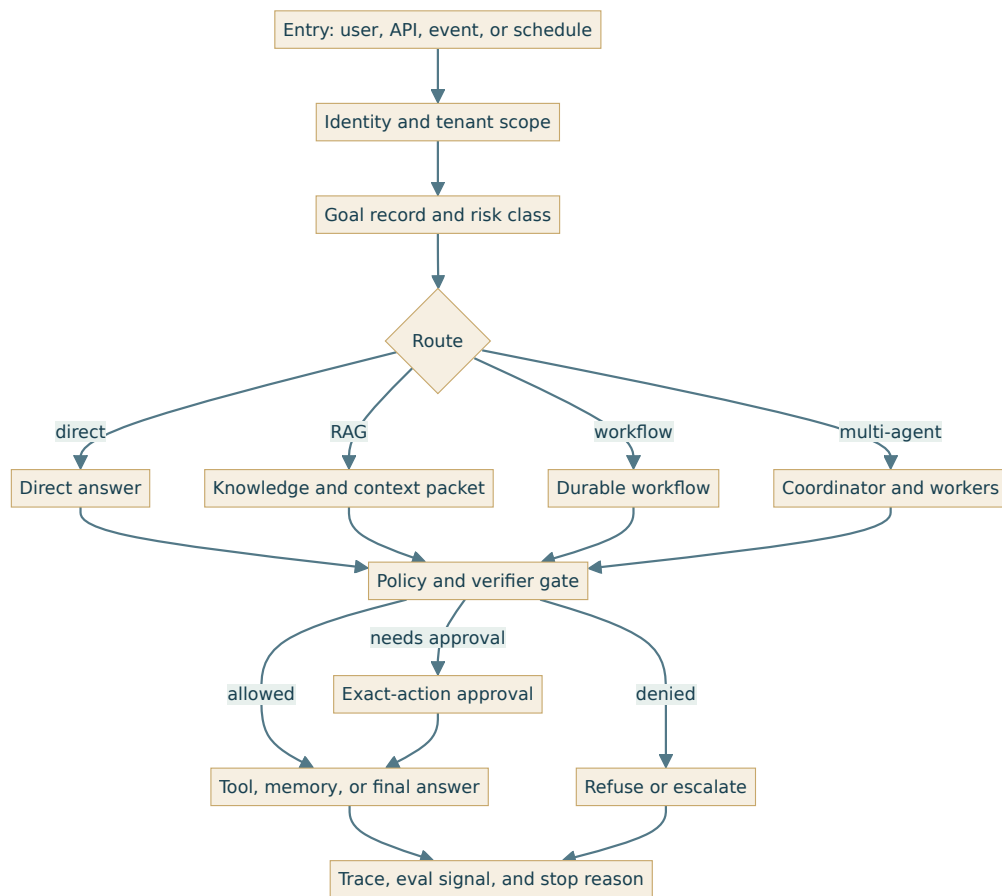
**Production Agentic System Reference Architecture**



Download the reusable review artifact: [reference architecture review checklist](#).

## Boundary Flow

Use this flow to review one request from entry to final status. It separates the control plane that decides what is allowed from the execution plane that retrieves evidence, calls tools, and writes state.



The flow should fail closed. A model proposal can move forward only after state, policy, schema, evidence, approval, or verifier checks accept it.

## What This Architecture Is For

Use this shape when the agent may touch private data, call tools, wait for approval, run longer than one request, or affect customers, money, infrastructure, compliance, or internal operations. It is heavier than a prompt chain, but the extra structure buys reviewability.

Do not start here for a read-only prototype, a one-shot summarizer, or a workflow whose steps are fully deterministic. For those systems, use the smallest pattern that solves the task and keep the production controls proportional to the risk.

## Readiness Questions

Use these questions before a design review:

1. What can the system do that a normal chat answer cannot?
2. Which action creates the highest risk, cost, or user impact?
3. Which service owns the state that survives retries, crashes, and approval waits?
4. Which policy check runs before retrieval, before tool use, before memory write, and before final answer?
5. Which evidence must the model see, and which evidence must stay outside model context?
6. What exact event moves a run from draft to approved action?



- 7. What trace proves that the system followed the intended path?
- 8. What switch disables the risky behavior without disabling the whole product?

If a team answers these with prompt text alone, the design is still a prototype.

Layered View

| Layer                       | Owns                                                              | Must Not Delegate To The Model                 |
|-----------------------------|-------------------------------------------------------------------|------------------------------------------------|
| Entry layer                 | User, tenant, event source, request validation, rate limits.      | Authentication, tenant scope, raw input trust. |
| Routing layer               | Direct answer, workflow, loop, RAG, or multi-agent path.          | Whether a high-risk route is allowed.          |
| Goal and state layer        | Goal record, run state, checkpoints, stop reason, replay data.    | Durable state mutation rules.                  |
| Context and knowledge layer | Retrieval, memory, freshness, citations, source policy.           | Source eligibility or access control.          |
| Tool and execution layer    | Tool schemas, timeouts, idempotency, side-effect records.         | Permission to call external systems.           |
| Policy and approval layer   | Authorization, risk class, exact-action approval, escalation.     | Final authority for risky actions.             |
| Evaluation layer            | Offline evals, runtime verifiers, release gates, regression data. | Quality or safety judgment without thresholds. |
| Observability layer         | Traces, audit logs, costs, latency, tool calls, incidents.        | Hidden reasoning as the only explanation.      |

The layers do not require separate services. A small system can implement several layers in one codebase. What matters is ownership: another engineer should be able to find where each decision is made, tested, logged, and rolled back.

Control Plane vs Execution Plane

Keep control decisions out of free-form model text. The model can propose, summarize, classify, or draft. Runtime-owned services decide whether the proposal may affect state, tools, memory, approvals, or users.

| Plane         | Owns                                               | Examples                                        | Release Evidence                                         |
|---------------|----------------------------------------------------|-------------------------------------------------|----------------------------------------------------------|
| Control plane | authorization, routing, policy, budgets, approval, | route policy, risk class, policy decision, eval | policy logs, eval report, approval trace, rollback drill |

| Plane           | Owns                                                         | Examples                                                                 | Release Evidence                                                   |
|-----------------|--------------------------------------------------------------|--------------------------------------------------------------------------|--------------------------------------------------------------------|
|                 | release gates                                                | threshold, kill switch                                                   |                                                                    |
| Execution plane | retrieval, tool calls, workflows, memory writes, model calls | vector search, order lookup, draft refund, workflow resume, memory write | tool manifest, source metadata, idempotency record, workflow trace |
| Evidence plane  | provenance, citations, state snapshots, audit trail          | context packet, citation map, state version, run transcript              | successful trace, failed trace, redaction proof                    |

This split prevents a common failure: a model both proposes an action and implicitly authorizes it. Production architecture should make the authorization path visible and testable.

## Maturity Ladder

Adopt the architecture in stages. Do not build every platform component before the system has proved value.

| Level                      | System Shape                                                                                        | Required Controls                                                                    | Promotion Test                                                                      |
|----------------------------|-----------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| 0 - Prototype              | Local or internal read-only demo.                                                                   | Clear non-production label, basic trace, no private data or side effects.            | The team can explain what would break before using real data.                       |
| 1 - Scoped Assistant       | One bounded task with retrieval or read-only tools.                                                 | Identity, tenant scope, source policy, citation check, stop reason.                  | Unauthorized or missing evidence produces denial, refusal, or escalation.           |
| 2 - Tool-Assisted Workflow | The system drafts or prepares actions through typed tools.                                          | Tool manifest, schema validation, idempotency for drafts, trace correlation.         | Unauthorized write-capable tools cannot be called from model text alone.            |
| 3 - Approved Action        | The system can trigger customer-visible, financial, operational, or durable effects after approval. | Exact-action approval, policy decision log, audit record, rollback or disable path.  | A risky action cannot execute without policy and approval evidence.                 |
| 4 - Production Runtime     | The system runs continuously with eval gates and incident response.                                 | Durable state, replay, dashboards, alerting, release gates, runbook, rollback drill. | Operators can reconstruct and disable a failed path without redeploying everything. |
| 5 - Platform Capability    | Multiple products or teams reuse the runtime, tools,                                                | Versioned contracts, compatibility tests,                                            | A contract change cannot silently break downstream                                  |

| Level | System Shape        | Required Controls                                         | Promotion Test  |
|-------|---------------------|-----------------------------------------------------------|-----------------|
|       | policies, or evals. | ownership matrix, tenant isolation, shared observability. | agent behavior. |

The ladder is not a status badge. It is a scope control. A level 1 system can be excellent for a read-only task. A level 3 system that cannot prove approval binding is not production-ready.

## First Implementation Path

Start with one user workflow and one risk boundary. A useful first implementation should be small enough to review in one design meeting.

| Step       | Build First                                                                            | Defer Until Needed                                                          |
|------------|----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| Goal       | One task with explicit success and refusal criteria.                                   | Broad assistant scope or multi-domain routing.                              |
| State      | Run ID, actor, tenant, route, stop reason, and trace link.                             | Long-term memory, cross-session personalization, or agent self-improvement. |
| Knowledge  | One approved source with freshness and citation rules.                                 | Multiple indexes, automatic ingestion, or open web retrieval.               |
| Tools      | One read tool or one draft-only tool.                                                  | Write tools, browser control, shell access, or payment actions.             |
| Policy     | Deny unsafe route, unauthorized source, and missing evidence.                          | Full policy platform with dynamic rule authoring.                           |
| Evaluation | Five cases: success, missing evidence, unauthorized source, tool failure, and refusal. | Large judge-based eval suites and synthetic data generation.                |
| Operations | Trace sample, owner, known limits, and disable switch.                                 | Full dashboard suite and multi-team on-call rotation.                       |

This path keeps architecture tied to evidence. Each added layer should close a named risk, not satisfy a diagram.

## Request Flow

1. Authenticate the user or event source.
2. Create a goal record with constraints, user scope, and requested outcome.
3. Route the task to a direct answer, Agentic RAG, durable workflow, or multi-agent process.
4. Run policy checks before retrieval, tool use, or side effects.

5. Persist observations, decisions, tool calls, and outputs.
6. Verify evidence, output quality, and policy compliance.
7. Return an answer, request approval, refuse, or escalate.
8. Store traces for debugging and eval dataset improvement.

### Example: Support Refund Request

A customer asks for a refund. The architecture should keep three decisions separate.

| Decision                           | Owner                               | Example Outcome                                                         |
|------------------------------------|-------------------------------------|-------------------------------------------------------------------------|
| Can the system inspect this order? | Entry, identity, and policy layers. | Allow same-tenant read or deny cross-tenant access.                     |
| Is the refund policy satisfied?    | Retrieval, policy, and eval layers. | Draft recommendation with policy citation or escalate missing evidence. |
| Can money move?                    | Approval and execution layers.      | Require finance approval; the model cannot issue the refund.            |

The model may draft a recommendation. It does not authenticate the user, choose its own tools, decide payment authority, write memory without policy, or mark the workflow complete without a stop reason. That separation is the point of the reference architecture.

### Control-Point Mapping

Turn the architecture into implementation tickets by assigning every risky transition to a concrete control point.

| Transition            | Required Control                       | Evidence To Store                                  | Release Test                                            |
|-----------------------|----------------------------------------|----------------------------------------------------|---------------------------------------------------------|
| request enters system | identity, tenant scope, rate limit     | actor ID, tenant ID, source, request ID            | cross-tenant request denied before retrieval            |
| route selected        | route policy and risk class            | route, risk class, routing reason                  | high-risk route cannot be selected by prompt text alone |
| context assembled     | source eligibility, freshness, budget  | context refs, source owner, freshness, token count | stale or unauthorized source excluded                   |
| model proposes action | schema and policy validation           | proposal, schema result, policy result             | malformed proposal rejected                             |
| tool call requested   | tool manifest, permission, idempotency | tool name, authority, arguments, idempotency key   | unauthorized write denied                               |

| Transition      | Required Control                | Evidence To Store                                    | Release Test                                     |
|-----------------|---------------------------------|------------------------------------------------------|--------------------------------------------------|
| approval needed | exact-action binding            | approver, action, amount or target, expiry, trace ID | broad approval rejected                          |
| result emitted  | final verifier and stop reason  | answer, citations, final status, stop reason         | missing evidence cannot produce completed status |
| memory written  | memory class and retention rule | memory type, owner, retention, deletion path         | sensitive memory denied or redacted              |

If a row has no owner, test, or stored evidence, the architecture is still a drawing. The implementation is ready only when each transition can fail closed.

## Control Points

- **Before retrieval:** enforce access control and source eligibility.
- **Before tool use:** validate schema, permission, budget, and approval needs.
- **Before final answer:** verify claims, citations, and policy.
- **Before memory write:** classify what kind of memory is being stored.
- **Before release:** run evals and regression checks.

## Reference Implementation Slices

Build the architecture in slices. Each slice should produce one deployable proof, not a diagram-only promise.

| Slice                | Build                                                     | Must Prove                                                               |
|----------------------|-----------------------------------------------------------|--------------------------------------------------------------------------|
| Read-only answer     | entry validation, routing, context packet, trace          | The system can answer with scoped evidence and a replayable trace.       |
| Tool-assisted answer | tool gateway, schemas, timeouts, idempotency              | Tool calls are authorized, typed, logged, and reversible where possible. |
| Approval workflow    | approval request, exact action binding, expiry, audit log | A human approves one concrete action, not broad future behavior.         |
| Durable workflow     | state store, checkpoints, retries, stop reasons           | The run survives failure without duplicating side effects.               |
| Production release   | eval suite, runbook, rollback switch, dashboards          | Operators can detect, stop, debug, and improve the system.               |

This sequence keeps teams from building orchestration before they can prove state, policy, and evidence boundaries.

## Deployment Shapes

The reference architecture does not require a large platform on day one. Use the smallest deployment shape that preserves the control points for the risk you are taking.

| Shape                  | Use When                                                                                                       | Acceptable Compression                                                   | Do Not Compress                                                                            |
|------------------------|----------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| Prototype              | The system is read-only, internal, and disposable.                                                             | One application service can own routing, state, retrieval, and traces.   | Authentication, tenant scope, source eligibility, and stop reason.                         |
| Pilot                  | A small group uses the system on real work with limited authority.                                             | Policy checks and approval records can live in the application database. | Tool permission, trace correlation, eval gates, and rollback switch.                       |
| Production             | The system affects users, records, money, infrastructure, or operations.                                       | Several layers can still share a repository or runtime.                  | Durable state, idempotency, audit logs, incident ownership, and release evidence.          |
| Regulated or high-risk | The system touches compliance, sensitive data, payments, security, healthcare, legal, or employment decisions. | Model calls can stay behind a single runtime gateway.                    | Independent policy enforcement, approval audit, redaction, retention, and evidence review. |

Compression is safe only when ownership stays explicit. A prototype can share code paths. It cannot share away the question of who owns state, policy, tool authority, evidence, and rollback.

## Boundary Placement Guide

Use this guide when deciding whether a component belongs inside the agent runtime, the application service, or a separate platform service.

| Boundary  | Keep Inside The App When                                        | Split Into A Service When                                                               |
|-----------|-----------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| Policy    | Rules are few, product-specific, and reviewed with the feature. | Multiple products, tenants, risk classes, or compliance owners use the same rules.      |
| Tools     | Tools are narrow, low-risk, and owned by one team.              | Tools are shared, write-capable, expensive, credentialed, or audited.                   |
| Retrieval | Sources are small, local, and low-sensitivity.                  | Sources require access control, freshness policy, citations, deletion, or owner review. |

| Boundary      | Keep Inside The App When                             | Split Into A Service When                                                              |
|---------------|------------------------------------------------------|----------------------------------------------------------------------------------------|
| Memory        | Memory is task-local or short-lived.                 | Memory crosses sessions, users, tenants, privacy classes, or retention rules.          |
| Workflow      | Runs finish in one request and have no side effects. | Runs wait for approval, retry, resume, schedule work, or call irreversible tools.      |
| Evaluation    | The feature has a small fixed eval set.              | Evals block releases, compare versions, feed incident review, or serve multiple teams. |
| Observability | Logs are enough to debug a toy run.                  | Operators need traces, redaction, cost, latency, tool, policy, and approval records.   |

The split should follow ownership and risk, not fashion. A separate service without a stronger owner is just more distributed code.

## Evidence Bundle

A design that claims to follow this architecture should attach these artifacts:

| Artifact               | Proves                                                                              |
|------------------------|-------------------------------------------------------------------------------------|
| Architecture diagram   | Ownership of state, tools, policy, memory, evals, approvals, and traces.            |
| State schema           | What persists, what can be replayed, and why the run stopped.                       |
| Tool manifest          | Which capabilities exist, who may call them, and what side effects they create.     |
| Context packet example | Which sources reached the model, with trust, freshness, and budget.                 |
| Policy decision log    | Why a tool, retrieval source, memory write, or approval path was allowed or denied. |
| Trace sample           | What happened in one successful run and one failed run.                             |
| Eval report            | Which behaviors block release.                                                      |
| Runbook                | Who owns incidents and how rollback works.                                          |

## Run Trace Contract

A reference architecture is only real if one run can prove the boundaries worked. Capture the run as structured trace data, not as scattered logs.

```
type ReferenceRunTrace = {
 runId: string;
```

```

tenantId: string;
userOrEvent: string;
goal: string;
route: "direct_answer" | "rag" | "agent_loop" | "durable_workflow" | "multi_agent";
stateVersion: string;
context: Array<{
 source: string;
 accessDecision: "allow" | "deny";
 freshness: "current" | "stale" | "unknown";
 citationRequired: boolean;
}>;
modelDecisions: Array<{
 step: string;
 proposal: string;
 acceptedBy: "schema" | "policy" | "human" | "runtime";
}>;
policyChecks: Array<{
 boundary: "retrieval" | "tool" | "memory" | "approval" | "final_answer";
 decision: "allow" | "deny" | "escalate";
 policyVersion: string;
}>;
toolCalls: Array<{
 name: string;
 authority: "read" | "draft" | "write" | "execute";
 idempotencyKey?: string;
 result: "success" | "failed" | "denied";
}>;
finalStatus: "completed" | "refused" | "needs_approval" | "policy_blocked" |
"evidence_missing" | "failed";
stopReason: string;
rollbackPath?: string;
};

```

The exact field names can change. The invariant should not: a reviewer can reconstruct who asked, what evidence entered context, what the model proposed, which policies ran, which tools executed, why the run stopped, and how to reverse or disable the risky path.

## Architecture Acceptance Tests

Before a pilot, run at least these acceptance checks against the trace contract.



| Test             | Passing Evidence                                                                                              |
|------------------|---------------------------------------------------------------------------------------------------------------|
| Tenant isolation | A cross-tenant request records a denial before retrieval or tool use.                                         |
| Missing evidence | The run stops with <code>evidence_missing</code> or <code>needs_approval</code> ; it does not invent support. |
| Policy denial    | A blocked action records policy version, boundary, denial reason, and no side effect.                         |
| Approval binding | A high-risk action waits for approval tied to exact action, actor, expiry, and trace ID.                      |
| Retry safety     | A retried write uses the same idempotency key or stops before duplicate side effects.                         |
| Memory control   | A memory write records classification, owner, retention, and deletion path.                                   |
| Rollback proof   | Operators can disable the model route, tool, workflow, policy, or agent without deleting the whole product.   |

If these tests are hard to write, the architecture is probably still hidden in framework glue, prompts, or informal process.

## Runtime Components

- Identity and tenant boundary
- Goal and state store
- Prompt and instruction registry
- Tool gateway
- Retrieval router
- Memory service
- Policy engine
- Approval service
- Workflow engine
- Evals service
- Trace and audit store

## Ownership Matrix

Assign owners before production. Missing ownership is an architecture defect.

| Component                    | Engineering Owner           | Operational Owner            | Release Artifact     |
|------------------------------|-----------------------------|------------------------------|----------------------|
| Identity and tenant boundary | platform or product backend | security or platform on-call | access-control tests |

| Component             | Engineering Owner                       | Operational Owner           | Release Artifact                   |
|-----------------------|-----------------------------------------|-----------------------------|------------------------------------|
| Goal and state store  | workflow/runtime team                   | runtime on-call             | state schema and replay fixture    |
| Retrieval router      | search or knowledge team                | content/data owner          | source policy and freshness report |
| Tool gateway          | integration team                        | service owner for each tool | tool manifest and audit sample     |
| Policy engine         | security, compliance, or product policy | policy owner                | policy decision log                |
| Approval service      | product workflow team                   | business approver owner     | approval trace sample              |
| Evals service         | AI platform or quality team             | release owner               | blocking eval report               |
| Trace and audit store | observability team                      | incident commander          | successful and failed run traces   |

One person can own several cells in a small team. The table still matters because every production failure will ask who owned the boundary.

## Design Review Questions

Ask these questions before a production pilot:

1. What decision does the model make, and what decision does software make after that?
2. Which state changes survive a crash, retry, or approval wait?
3. Which tool call has the highest cost, risk, or irreversibility?
4. What evidence is required before the system answers or acts?
5. What happens when evidence is missing, stale, conflicting, or unauthorized?
6. What gets written to memory, and who can correct or delete it?
7. Which eval blocks release if the system regresses tomorrow?
8. Can an operator reconstruct the failed run without guessing?
9. What can be disabled without redeploying the whole system?
10. What would make the team shut the agent off?

## Minimum Production Checklist

- Explicit owner for goal and state
- Tool schemas and validation
- Human approval for high-risk actions
- Access-controlled retrieval
- Citation validation for grounded answers

- Evals for core tasks
- Traces for every run
- Budget, timeout, and cancellation controls
- Incident review path
- Rollback or disable switch

## Reference Architecture Review Gate

Use the [reference architecture review checklist](#) before a pilot or production release. Block release when:

- the model owns authentication, authorization, policy, or side-effect permission;
- durable state has no schema, replay path, or stop reason;
- retrieval sources lack access control, trust, freshness, or citation checks;
- approvals authorize vague future behavior;
- evals inspect only final text and ignore tools, policy, evidence, and state transitions;
- operators cannot reconstruct a successful run and a failed run from traces.

## Scaling Path

Start small:

1. Single agent with tool validation.
2. Add goals and state.
3. Add retrieval and citation checks.
4. Add durable workflow for long-running tasks.
5. Add human approval gates.
6. Add eval datasets and observability.
7. Add multi-agent decomposition only when one agent becomes a bottleneck.

## Anti-Patterns

Avoid these designs:

- The model chooses from a broad tool list without route-level permission.
- Prompt text is the only policy boundary.
- Retrieval, memory, and tool results are merged into context without provenance.
- The system can act but cannot replay why it acted.
- Approvals authorize vague future behavior instead of one exact action.
- Evals check only final answer quality and ignore tool use, policy, evidence, and stop reason.
- Rollback requires deleting the whole deployment because prompts, tools, and policies cannot be disabled independently.

## Related Chapters

- [Agentic System Architecture](#)

- [Agentic RAG Systems](#)
- [MCP-first Tool Use](#)
- [Production Runtime Overview](#)
- [Durable Workflows](#)
- [Policy Enforcement](#)
- [Observability and Evals](#)

# Production Runtime Overview

Production agentic systems need more than prompts, tools, and a good model. They need a runtime that owns the execution boundary.

The core rule is the same throughout this book: the model proposes, the runtime decides. The model can propose a plan, a tool call, a memory write, a retrieval query, a reply, or a stop condition. The runtime owns whether that proposal is valid, allowed, affordable, observable, durable, and safe to execute.

This is where many agent projects fail. They treat the model call as the system, then add logging, retries, approvals, evals, and rollback after the first incident. A production runtime inverts that order. It gives the agent a controlled place to operate.

This chapter owns execution control across runs: admission, state, policy, budgets, tool execution, recovery, rollout, and operator visibility. It does not own product UX, domain policy content, model training, or the internal implementation of tools and services.

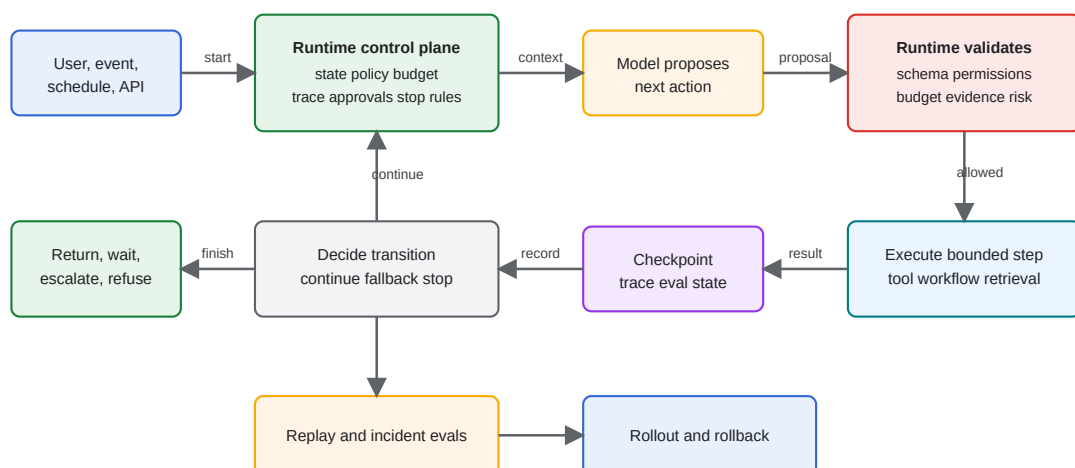
Read this after [Agent Harnesses](#). The harness explains the working environment around one agent; this chapter explains what has to be true when that work runs as a production system with queues, budgets, retries, state, rollout, rollback, and operators.

For a step-by-step path from lab to release, use [Deployment Walkthrough](#). For framework setup and decision templates, use [Real Framework Setup Notes](#) and [Templates and Worksheets](#).

Download the operating worksheet: [runtime SLO and incident review worksheet](#).

Use this diagram as a control-plane map. Each box names a runtime responsibility that should have an owner, a trace record, and a rollback path before production traffic.

## Production Runtime Overview



## What The Runtime Owns

The production runtime is the control plane around model judgment.

| Runtime Concern      | What It Owns                                                                        |
|----------------------|-------------------------------------------------------------------------------------|
| State                | Goal, run status, workflow step, attempts, tool results, approvals, stop reason.    |
| Policy               | Permissions, risk classification, denial, approval, escalation, audit requirements. |
| Budgets              | Tokens, model calls, tool calls, retries, delegations, wall-clock time, cost.       |
| Tools                | Schemas, permissions, timeouts, idempotency, side-effect records.                   |
| Memory and retrieval | Source eligibility, freshness, access control, evidence references, memory writes.  |
| Observability        | Trace IDs, spans, costs, latency, model and tool events, replay metadata.           |
| Evaluation           | Runtime checks, incident fixtures, release gates, regression datasets.              |
| Recovery             | Retries, fallbacks, circuit breakers, checkpoints, compensation, rollback.          |

The runtime does not remove autonomy. It makes autonomy bounded enough to trust.

## Runtime Responsibility Matrix

The runtime should make ownership explicit. If nobody owns one of these rows, the model will eventually own it by accident.

| Responsibility      | Runtime decision                                                                  | Failure when missing                                         |
|---------------------|-----------------------------------------------------------------------------------|--------------------------------------------------------------|
| Run admission       | Should this request start, wait, refuse, or route elsewhere?                      | Unsafe or unsupported work enters the loop.                  |
| State ownership     | Which state is durable, temporary, visible, or deleted?                           | Lost progress, hidden memory, non-replayable failures.       |
| Execution mode      | Should the task run synchronously, asynchronously, or as a durable workflow?      | Timeouts, zombie runs, blocked users, partial side effects.  |
| Proposal validation | Is the model proposal valid for schema, policy, budget, and state?                | Tool calls execute because the model sounded confident.      |
| Tool execution      | Which credentials, timeout, idempotency key, and side-effect record apply?        | Duplicate writes, leaked credentials, unclear ownership.     |
| Approval state      | What waits for human review, who approved it, and what exact action was approved? | Approval becomes broad permission instead of a bounded gate. |
| Cancellation        | What stops immediately, what drains, and what must be compensated?                | Cancelled runs keep spending or continue side effects.       |

| Responsibility | Runtime decision                                                                   | Failure when missing                              |
|----------------|------------------------------------------------------------------------------------|---------------------------------------------------|
| Rollout        | Which model, prompt, policy, tool schema, retriever, or harness version is active? | Regressions ship globally with no quick rollback. |
| Operations     | Who is paged, what is disabled, and what evidence is preserved?                    | Incidents turn into manual archaeology.           |

## The Runtime Loop

A production runtime loop is not just observe, decide, act. It is closer to:

1. receive a request, event, schedule, webhook, or workflow command;
2. authenticate the caller and load task class, risk class, state, policy, budget, and trace ID;
3. assemble the working set: goal, constraints, evidence, allowed tools, memory, and stop rules;
4. ask the model or deterministic router for a bounded proposal;
5. validate the proposal against schema, policy, budget, state, evidence, and approval rules;
6. execute one bounded step through a tool, workflow, retrieval service, evaluator, or approval gate;
7. checkpoint state, trace events, cost, latency, side effects, and stop reason;
8. decide whether to continue, fallback, wait, escalate, refuse, compensate, or complete;
9. convert important failures and near misses into eval cases.

This loop is what separates an agentic product from a demo. The model can still be creative and adaptive, but the system knows what happened and why.

## Runtime Boundaries

A good runtime creates explicit boundaries:

- **Decision boundary:** the model can suggest actions, but software validates them.
- **Authority boundary:** side effects require tool schemas, permissions, budgets, and approval rules.
- **State boundary:** durable workflow state is separate from model context.
- **Context boundary:** the model sees a selected working set, not every available document or memory.
- **Cost boundary:** every loop, tool, retry, model call, and delegation spends from a budget.
- **Policy boundary:** denial and escalation are runtime outcomes, not prompt preferences.
- **Recovery boundary:** retries, fallbacks, replay, and rollback are designed before production traffic.

When these boundaries are missing, the model becomes the control plane by accident.

## Execution Modes

Do not run every agent the same way. Match the runtime mode to the work.

| Mode                | Use when                                                                     | Runtime requirements                                                       |
|---------------------|------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| Synchronous request | The task is short, read-heavy, and safe to fail fast.                        | Tight timeout, small budget, no irreversible side effects, complete trace. |
| Async job           | The task may take seconds or minutes but does not need complex compensation. | Queue, status record, cancellation, retries, idempotency, progress events. |
| Durable workflow    | The task spans approvals, external systems, retries, or long-running state.  | Checkpoints, resumability, compensation, replay, versioned workflow state. |
| Event-triggered run | The task starts from webhook, schedule, stream, or system event.             | Deduplication, event identity, ordering policy, backpressure, audit trail. |
| Human-gated run     | The task can prepare work but needs approval before execution.               | Approval record, exact-action binding, pause and resume semantics.         |

The wrong mode creates production bugs. A refund workflow should not depend on one HTTP request staying alive. A short classifier should not pay the complexity cost of a durable workflow engine.

## Queues, Backpressure, And Concurrency

Agents consume scarce resources: model quota, tool capacity, human approval time, database connections, browser workers, and money. The runtime should control admission and concurrency before the loop starts spending.

Useful controls include per-tenant queues, route-level concurrency limits, model-provider rate limits, tool-specific bulkheads, retry budgets, dead-letter queues, and priority classes. Backpressure is not just an infrastructure concern. It is how the system refuses low-value work before it damages high-value work.

Concurrency also affects correctness. Two runs should not issue the same refund, update the same ticket, rewrite the same memory, or deploy the same service without coordination. Use locks, version checks, idempotency keys, or workflow state transitions where duplicate work would be harmful.

## Rollout And Rollback

Production agents change in more ways than normal services. A release may change model, prompt, tool schema, retriever, memory policy, approval rule, sandbox profile, evaluator, or workflow code. The runtime should version those pieces and record the active version set on every run.

Rollout should be gradual for high-risk agents:

- start with offline evals;
- run shadow or replay tests where possible;



- enable a small tenant, route, or percentage;
- compare traces, costs, stop reasons, policy denials, and user-visible outcomes;
- keep a rollback path for each changed component.

Rollback must be operational, not theoretical. Operators should be able to disable a tool, pin a model, revert a prompt, tighten a policy, stop a route, drain a queue, or force human approval without redeploying the whole product.

## How The Production Runtime Chapters Compose

Read the production runtime section as one operating model:

- [Durable Workflows](#) own long-running state, retries, checkpoints, approvals, compensation, and resumability.
- [Observability and Evals](#) records what happened and turns behavior into something engineers can inspect.
- [Production Evaluation Feedback Loops](#) converts production failures into regression cases and release gates.
- [Cost Controls and Runtime Budgets](#) defines how much autonomy, spend, time, and human attention a run may consume.
- [Policy Enforcement](#) keeps permission, risk, and compliance decisions outside the model.
- [Event-Triggered Agents](#) shows how agents respond to events without losing idempotency, state, and auditability.
- [Mastra Runtime](#) maps these production concerns into a concrete runtime style.

The chapters are separate because each boundary deserves attention. In a real system, they should work together.

## Launch Evidence Map

Before launch, each runtime concern should produce evidence that another engineer can inspect. A green build is useful, but it does not prove the agent can be operated.

| Concern   | Evidence Artifact                                                        | Release Question                                                       |
|-----------|--------------------------------------------------------------------------|------------------------------------------------------------------------|
| Admission | Route table with task class, risk class, tenant scope, and refusal path. | Which requests can start, wait, route elsewhere, or refuse?            |
| State     | Run-state schema, checkpoint example, and deletion rule.                 | Can the team reconstruct what the agent believed and where it stopped? |
| Policy    | Decision matrix, reason codes, and approval rules.                       | Can software stop an unsafe proposal before it executes?               |
| Budget    | Per-route budget policy and exhaustion behavior.                         | What happens when the task is no longer worth more spend?              |

| Concern              | Evidence Artifact                                                         | Release Question                                                    |
|----------------------|---------------------------------------------------------------------------|---------------------------------------------------------------------|
| Tools                | Tool manifest with schemas, permissions, timeouts, and side-effect class. | Can a tool call be validated without trusting model prose?          |
| Memory and retrieval | Source policy, freshness rule, citation rule, and memory retention class. | Can the agent explain what evidence it used and why it was allowed? |
| Observability        | Trace contract and dashboard links.                                       | Can an operator move from symptom, to trace, to owner?              |
| Evaluation           | Blocking eval list, incident fixtures, and release-gate output.           | Which known failures would block this release?                      |
| Recovery             | Runbook with retry, fallback, compensation, and rollback actions.         | What can operators disable or restore without a full redeploy?      |

The evidence does not need to be elaborate. It needs to be real, current, and linked from the release record.

## Minimal Runtime Contract

Every production run should be able to produce a contract like this:

```
type RuntimeRun = {
 runId: string;
 traceId: string;
 requestId: string;
 actorId: string;
 tenantId: string;
 route: string;
 goal: string;
 autonomyLevel: "advisory" | "drafts_for_review" | "executes_after_approval" |
"bounded_autonomous";
 riskClass: "low" | "medium" | "high";
 executionMode: "sync" | "async_job" | "durable_workflow" | "event_triggered";
 status: "queued" | "running" | "waiting" | "succeeded" | "failed" | "refused" |
"cancelled";
 versionSet: {
 model: string;
 prompt: string;
 policy: string;
 toolSchema: string;
 retriever?: string;
 harness: string;
 };
};
```

```

budgetPolicyVersion: string;
policyVersion: string;
workflowStep?: string;
allowedTools: string[];
idempotencyKey?: string;
approvalId?: string;
checkpointRef?: string;
stopReason?: string;
};

```

This is not enough to implement a full platform, but it is enough to make the hidden parts visible. If a run does not have actor, tenant, route, trace ID, risk class, autonomy level, execution mode, version set, budget policy, policy version, allowed tools, status, and stop reason, it will be hard to operate.

For high-risk work, this contract should be stored before the first model call. The run may change state, but the runtime should never be guessing who started it, what authority it has, what version is active, or why it stopped.

## Operating Dashboard View

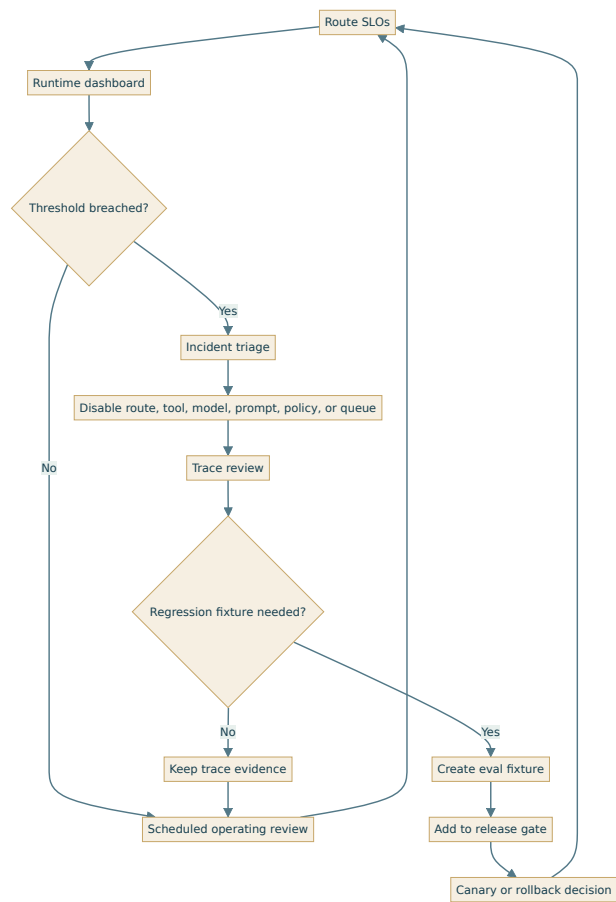
The runtime dashboard should show control, not just activity.

| Panel            | Shows                                                                        | Operator Action                                                 |
|------------------|------------------------------------------------------------------------------|-----------------------------------------------------------------|
| Active runs      | route, tenant, risk class, status, workflow step, age, and owner.            | Cancel, pause, or escalate stuck work.                          |
| Budget pressure  | token spend, tool spend, retry spend, and runs near exhaustion.              | Lower concurrency, require approval, or stop low-priority work. |
| Policy decisions | denies, approvals, escalations, false allows, and override rate.             | Tighten rules, review exceptions, or add eval fixtures.         |
| Tool health      | timeout rate, retry rate, side-effect failures, and idempotency conflicts.   | Disable a tool, drain a queue, or switch fallback path.         |
| Trace quality    | missing spans, missing stop reasons, redaction failures, and replay gaps.    | Block release until evidence is complete.                       |
| Release versions | active model, prompt, policy, retriever, workflow, and tool schema versions. | Pin, roll back, or canary a component.                          |
| Eval status      | blocking failures, flaky cases, incident fixture failures, and gate history. | Stop rollout or assign fixture repair.                          |

If the dashboard cannot answer “what should an operator do now?”, it is a reporting page, not a runtime control surface.

# Runtime SLO And Incident Loop

Service-level objectives (SLOs) make runtime quality explicit. Do not define them only for uptime. Agentic systems also need SLOs for trace coverage, policy-decision coverage, approval latency, eval-gate health, cost, and stop-reason completeness.



Use this loop after launch and during canaries. A good operating review can answer which SLO moved, which route changed, which trace proves the failure, which component can be disabled, and which eval now prevents recurrence.

| Runtime SLO              | Example Target                                                                                 | Investigate When                                   | Roll Back Or Pause When                                  |
|--------------------------|------------------------------------------------------------------------------------------------|----------------------------------------------------|----------------------------------------------------------|
| Trace coverage           | 99% of high-risk runs have complete run, route, policy, tool, approval, and stop-reason spans. | Any high-risk trace lacks policy or stop reason.   | A release changes risky behavior without trace coverage. |
| Policy-decision coverage | 100% of write, send, memory, and external-action paths record a policy decision.               | Policy span is missing or detached from tool span. | A side effect executes without policy evidence.          |

| Runtime SLO              | Example Target                                                                                           | Investigate When                                                         | Roll Back Or Pause When                                                         |
|--------------------------|----------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|---------------------------------------------------------------------------------|
| Approval latency         | 95% of approval waits resolve or expire inside the business SLA.                                         | Approval waits age without owner or expiry.                              | Stale approvals can resume changed actions.                                     |
| Cost budget              | Route stays within agreed per-run or per-completed-task budget.                                          | Cost per completed task spikes after model, prompt, or retrieval change. | Budget exhaustion does not stop or degrade safely.                              |
| Eval-gate health         | Blocking evals pass before prompt, model, policy, tool, memory, or workflow changes.                     | Warning eval failures cluster around one boundary.                       | A known incident fixture fails or becomes flaky without owner.                  |
| Stop-reason completeness | Every terminal run records completed, refused, failed, cancelled, timed out, blocked, or needs approval. | Terminal states contain generic <code>done</code> or missing reason.     | Operators cannot distinguish success, refusal, failure, or partial side effect. |

The numbers will vary by product. The important part is that each SLO names an owner and an action. A threshold without an operator decision is only a chart.

## Runtime Checklist

Before a production agent handles real work, answer:

- What owns the active goal?
- Where is durable run state stored?
- Which component validates model proposals?
- Which execution mode fits this task?
- Which tools are allowed for this task class?
- Which actions require approval?
- What budget applies to the run?
- What happens when the budget is exhausted?
- What trace events are required?
- What side effects need idempotency or compensation?
- What breaker, fallback, or escalation path exists?
- What evals block release?
- What queue, concurrency limit, or backpressure policy applies?
- What component versions are recorded for every run?
- What can be rolled back without redeploying the whole system?

If those answers are vague, the system is still a prototype, even if it is already serving users.

## Failure Modes

- The model owns state transitions because the runtime has no workflow state.
- Policy lives in the prompt instead of a runtime enforcement layer.
- Tool calls happen before budget, permission, schema, or approval checks.
- Retry logic repeats side effects without idempotency keys.
- Observability records final answers but not proposals, validation decisions, tool calls, and stop reasons.
- Evals test final prose while the runtime path remains untested.
- Operators cannot disable a risky tool, prompt version, model route, or workflow step quickly.
- The system can continue spending tokens, tool calls, and human attention after the task is no longer worth it.
- A synchronous request hides a long-running workflow until the first timeout or duplicate retry.
- Queues grow without backpressure, priority, cancellation, or dead-letter handling.
- Cancelled runs stop the UI but not the queued tool call or external workflow.
- A model or prompt change ships without versioned traces, targeted evals, or a rollback plan.
- Partial failure looks like success because the runtime records the final answer but not the failed side effect.
- Durable state exists, but the model context and workflow state disagree about what step is active.

## Design Rule

Production runtime is where agentic architecture becomes honest. If the runtime cannot explain, bound, replay, and stop the agent, the model is not the only risk. The architecture is.

Continue with [Durable Workflows](#) for resumable execution, [Observability and Evals](#) for operational evidence, and [Policy Enforcement](#) for authority decisions.

## Related Chapters

- [Architecture Before Autonomy](#)
- [Agentic System Architecture](#)
- [Reference Architecture](#)
- [Durable Workflows](#)
- [Observability and Evals](#)
- [Production Evaluation Feedback Loops](#)
- [Cost Controls and Runtime Budgets](#)
- [Policy Enforcement](#)

# Deployment Walkthrough

This walkthrough turns a lab-derived agent into a production candidate. It is framework-agnostic: the same gates apply whether the implementation uses direct TypeScript, Python, LangGraph, AutoGen, Mastra, CrewAI, or a custom mini-runtime.

Download the [deployment walkthrough review checklist](#) before using this chapter for a release review.

For complete examples, use the [Capstone Projects](#) after this chapter.

The goal is not to deploy faster. The goal is to deploy with enough control that the team can inspect, stop, replay, and improve the system after real users arrive.

## Scope

Use this walkthrough for systems that can read private data, call tools, write memory, send messages, create drafts, execute workflow steps, or influence business decisions.

For throwaway demos, keep the process lighter. For production, do not skip the gates that match the system’s authority.

## Deployment Readiness Questions

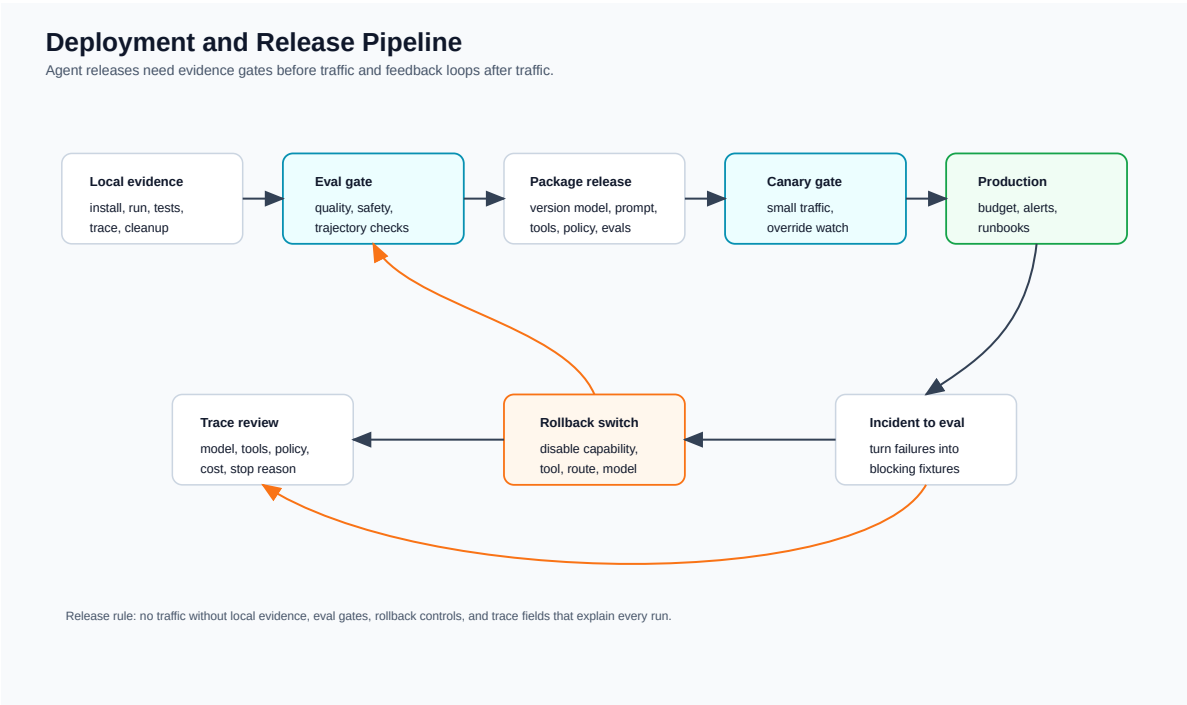
Use these questions before promoting a lab, pattern implementation, or capstone into a service:

| Question                             | Release Evidence                                                                            |
|--------------------------------------|---------------------------------------------------------------------------------------------|
| What authority does the agent have?  | Read, write, approval, tool, memory, and user-facing action inventory.                      |
| What must be durable?                | Checkpoints for approvals, retries, side effects, and workflow waits.                       |
| What blocks release?                 | Tests, evals, trace review, policy checks, and security gates.                              |
| What can be disabled without deploy? | Model route, prompt version, tool capability, memory writes, workflow, or full agent route. |
| What can operators inspect?          | Runbook, trace dashboard, eval dashboard, config version, and incident log.                 |
| What happens during partial failure? | Retry, compensation, degradation, escalation, and stop reason rules.                        |

The release is not ready when the only proof is “the demo worked.” It is ready when a second engineer can deploy, inspect, stop, and replay it.

## Release Pipeline

Use this diagram as the deployment control path. A production agent release needs local evidence, eval gates, canary observation, rollback controls, and incident-to-eval feedback.



## 1. Local Development

Local development should prove the runtime contract before cloud infrastructure exists.

Required local evidence:

| Evidence | Required Proof                                  |
|----------|-------------------------------------------------|
| install  | clean checkout can install dependencies         |
| run      | one command executes the vertical slice         |
| test     | unit and trajectory tests pass                  |
| eval     | at least one release-blocking eval runs locally |
| trace    | local run emits structured trace events         |
| cleanup  | local state and temporary data can be removed   |

Suggested local commands:

```
npm test
npm run typecheck
npm run book:build
```



For Python framework variants, add the project-specific virtual environment, install, test, and eval commands to the lab README.

## 2. Configuration And Secrets

Configuration should make deployment behavior explicit without exposing secrets.

Use these environment groups:

| Group          | Examples                                                       |
|----------------|----------------------------------------------------------------|
| model provider | <code>OPENAI_API_KEY</code> , model name, timeout, retry limit |
| runtime        | environment, region, service name, release version             |
| storage        | checkpoint store URL, trace store URL, memory store URL        |
| policy         | policy version, approval mode, disabled capabilities           |
| observability  | trace export endpoint, sampling mode, redaction mode           |
| evals          | eval dataset version, release threshold, failure mode          |

Rules:

- commit `.env.example`, not `.env`;
- keep secrets in the deployment platform's secret manager;
- fail startup when required secrets are missing;
- log which configuration version loaded, not secret values;
- treat prompt, model, tool, policy, and eval versions as release inputs.

## 3. Persistence And Checkpointing

Persistence depends on authority. A read-only answer can often be stateless. A workflow that waits for approval, retries tools, or creates side effects needs durable state.

Choose the minimum persistence boundary that supports recovery:

| Need                    | Persistence Boundary                    |
|-------------------------|-----------------------------------------|
| request-only answer     | request log plus trace                  |
| conversation continuity | thread state or conversation store      |
| human approval wait     | checkpoint plus approval record         |
| tool side effect        | idempotency key plus side-effect record |
| long-running workflow   | workflow state plus step checkpoints    |

| Need   | Persistence Boundary                              |
|--------|---------------------------------------------------|
| memory | governed memory store with retention and deletion |

Checkpoint every externally visible step:

1. accepted request;
2. planned action;
3. policy decision;
4. approval request or approval result;
5. tool call attempt;
6. side-effect result;
7. final response;
8. eval or post-run quality result.

Retries should read the checkpoint and decide whether to continue, compensate, or stop. They should not replay side effects blindly.

## 4. Observability Export

Agent observability must explain one run and aggregate many runs.

Export these events:

| Event     | Required Fields                                                                    |
|-----------|------------------------------------------------------------------------------------|
| run       | trace ID, run ID, actor, tenant, environment, release                              |
| model     | model, prompt version, input reference, output status, tokens, cost, latency       |
| tool      | tool name, redacted arguments, authorization, status, retry count, idempotency key |
| retrieval | source IDs, freshness, access decision, citation requirements                      |
| memory    | read IDs, write IDs, retention class, policy basis                                 |
| policy    | policy version, decision, reason code, enforcement effect                          |
| approval  | approver role, exact action, expiry, result                                        |
| eval      | case ID, evaluator version, score, threshold, pass/fail                            |

Do not store raw secrets, credentials, payment data, or private content unless the retention policy explicitly allows it. Prefer references to encrypted records when raw content is not needed for debugging.

## 5. Eval Gate In CI

CI should block risky changes before deployment.

Tie eval subsets to change type:

| Change    | Blocking Eval                                               |
|-----------|-------------------------------------------------------------|
| prompt    | task success, schema validity, policy compliance            |
| model     | task success, refusal behavior, tool argument quality, cost |
| tool      | authorization, idempotency, error handling                  |
| retrieval | source access, freshness, citation correctness              |
| memory    | read scope, write policy, deletion behavior                 |
| workflow  | route correctness, retry, cancellation, resume              |
| policy    | false allow, false deny, approval routing                   |

A minimal CI gate should run:

```
npm test
npm run typecheck
```

Add project-specific eval commands next to the implementation. The gate should fail closed: if the eval dataset cannot load, the release should stop.

GitHub Actions Gate

A minimal GitHub Actions workflow should separate ordinary tests from release-blocking agent checks.

```
name: agent-release-gate

on:
 pull_request:
 workflow_dispatch:

jobs:
 verify:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v4
 - uses: actions/setup-node@v4
 with:
 node-version: 20
 cache: npm
 - run: npm ci
 - run: npm test
```

```
- run: npm run typecheck --if-present
- run: npm run eval:release --if-present
- run: npm run trace:contract --if-present
```

For Python agents, add setup-python, dependency install, unit tests, and eval commands. Keep production secrets out of pull-request jobs. CI should use synthetic fixtures, mock tools, redacted traces, and staging credentials only when explicitly approved.

The release gate should publish a small evidence summary: commit SHA, eval dataset version, passed checks, failed checks, trace contract result, and release owner. A green CI badge is not enough when the agent can call tools or affect users.

## 6. Rollout

Roll out by capability, not by hope.

Use stages:

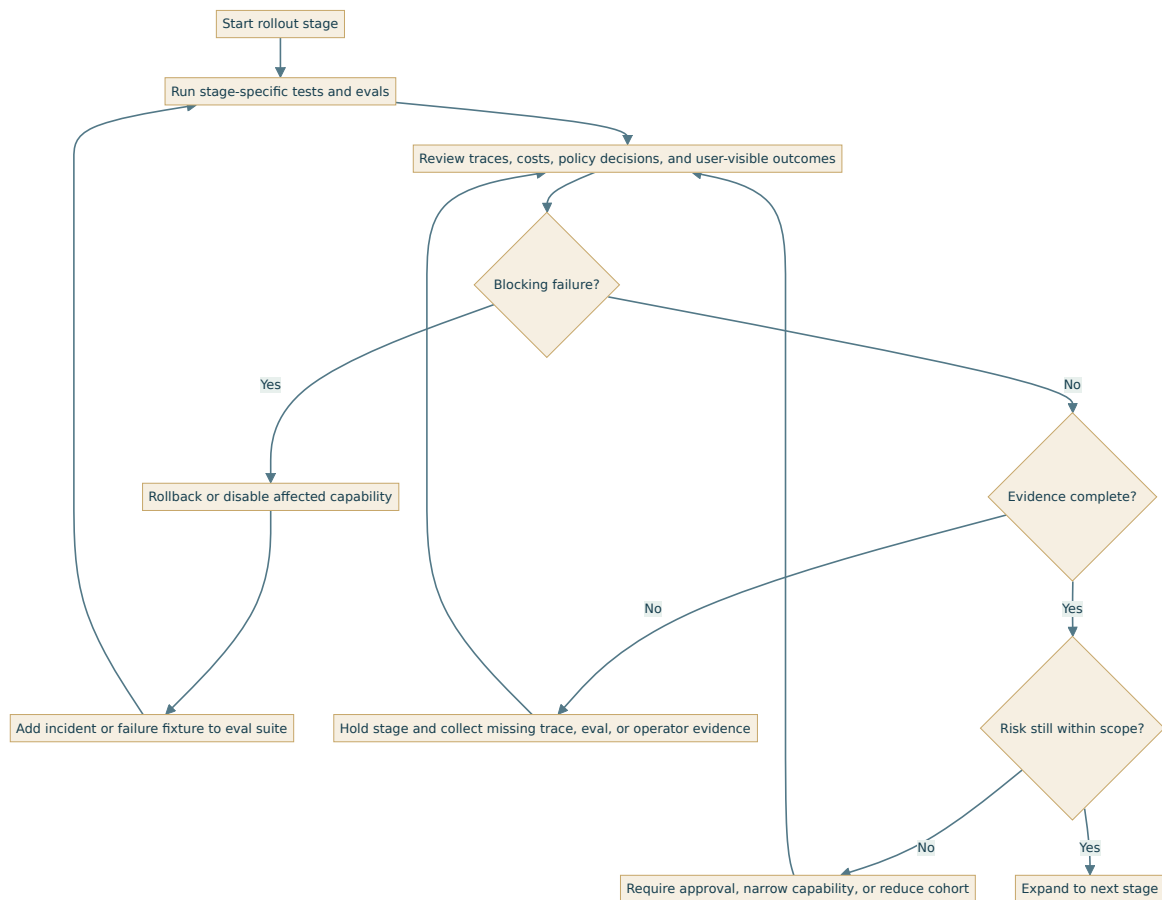
1. local run with deterministic fixtures;
2. staging run with synthetic data;
3. internal run with read-only authority;
4. limited tenant or cohort;
5. expanded traffic with dashboards and alerts;
6. full release after trace and eval review.

At each stage, record:

- release version;
- model and prompt version;
- tool schema version;
- policy version;
- eval dataset version;
- trace export status;
- rollback owner.

### Rollout Decision Flow

Use this flow at each rollout stage. The goal is to make expansion a decision based on evidence, not a default next step.



## 7. Rollback And Kill Switch

Every production agent needs a fast disable path.

Define kill switches at several layers:

| Layer    | Disable Action                                        |
|----------|-------------------------------------------------------|
| model    | route to previous model or deterministic fallback     |
| prompt   | revert prompt version                                 |
| tool     | disable one capability in the tool registry           |
| memory   | disable writes while keeping reads available if safe  |
| workflow | pause new runs and let safe in-flight runs finish     |
| policy   | change risky actions to approval-required or denied   |
| agent    | route traffic back to human or deterministic workflow |

Rollback should not require a code deploy for common failures. Tool disablement, model rollback, prompt rollback, and policy tightening should be operational controls.

## 8. Production Runbook

Create a runbook before launch.

Minimum runbook:

```
service:
owner:
on-call:
runtime:
framework:
release:
model versions:
prompt versions:
tool registry:
policy version:
memory stores:
checkpoint stores:
trace dashboard:
eval dashboard:
known failure modes:
rollback command:
kill switch:
incident channel:
post-incident eval process:
```

The runbook should link to the framework selection ADR, production readiness worksheet, eval suite, and deployment dashboard.

9. Concrete Runtime Path

Use this path when a lab or capstone becomes a service. It keeps framework code behind product-owned contracts.

| Step       | Artifact                                                 | Completion Signal                                                         |
|------------|----------------------------------------------------------|---------------------------------------------------------------------------|
| package    | container image or serverless bundle                     | image contains only required runtime files, lockfile, and config template |
| entrypoint | HTTP handler, queue consumer, or workflow worker         | request creates a run ID and trace ID before model or tool work starts    |
| config     | <code>.env.example</code> , secret names, policy version | startup fails closed when required values are missing                     |
| state      | database table, checkpoint, or workflow store            | interrupted or retried run resumes from known state                       |

| Step          | Artifact                                        | Completion Signal                                                         |
|---------------|-------------------------------------------------|---------------------------------------------------------------------------|
| tools         | registry plus capability metadata               | each tool has side-effect class, owner, timeout, retry, and approval rule |
| evals         | release gate command                            | CI blocks deploy when grounding, policy, schema, or trajectory evals fail |
| observability | trace export and dashboard                      | one run can be reconstructed without raw secrets                          |
| rollback      | feature flag, route switch, or tool disablement | owner can disable risky capability without code deploy                    |

Minimum service contract:

`POST /runs`

`input: actor, tenant, task, request payload, idempotency key`

`output: run_id, trace_id, status, response or escalation`

`side effects: none before policy, approval, and idempotency checks`

For queue or workflow deployments, keep the same contract even if transport changes. The request envelope, state record, trace ID, policy decision, and eval result should look the same across HTTP, worker, and scheduled jobs.

## 10. Cloud Deployment Shapes

Different cloud shapes can host the same agent contract. Pick the simplest shape that preserves state, policy, traces, and rollback.

| Shape                  | Use When                                                                      | Required Controls                                                          |
|------------------------|-------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| container service      | HTTP or worker agent needs long-lived process, local cache, or custom runtime | health check, autoscaling limit, secret manager, trace export, kill switch |
| serverless function    | short stateless step with strict timeout and no approval wait                 | external state store, idempotency key, timeout budget, cold-start test     |
| queue worker           | event-triggered or background work                                            | dead-letter queue, retry policy, backpressure, replay procedure            |
| workflow engine worker | long-running work, approvals, compensation, or resume after failure           | checkpoint store, versioned workflow definition, stuck-run dashboard       |
| scheduled job          | periodic eval, memory cleanup, ingestion, or report generation                | lock, idempotency, last-run record, alert on missed run                    |

Cloud deployment should not change the agent’s authority model. If a local run requires approval before sending email, the cloud worker must require the same approval. If the local trace redacts tool arguments, the cloud trace must redact them too.

## 11. Research RAG Deployment Notes

Research RAG systems need extra deployment controls because retrieval can expose forbidden, stale, or unsupported material.

Required runtime controls:

| Control          | Production Rule                                                                  |
|------------------|----------------------------------------------------------------------------------|
| ingestion        | store source ID, title, version, freshness, owner, ACL group, and citation label |
| retrieval        | retrieve candidates with metadata, not text alone                                |
| source filter    | enforce ACL, freshness, and source type before context assembly                  |
| context packet   | include evidence, omissions, and citation labels as structured fields            |
| answer synthesis | answer only from approved evidence packet                                        |
| citation eval    | block answers that cite missing, stale, or forbidden sources                     |
| fallback         | return ranked approved sources or escalate when evidence is weak                 |

Deployment sequence:

1. deploy retrieval in read-only mode;
2. compare retrieved candidates with source-filter output;
3. enable answer synthesis for internal users only;
4. gate release on citation faithfulness, forbidden-source omission, and stale-source rejection;
5. add dashboards for missing evidence, stale-source hits, forbidden-source attempts, and citation failures;
6. keep a kill switch that disables synthesis and returns approved source lists only.

This path connects directly to the [Research RAG Agent capstone](#) and the native LangGraph slice under `native-framework-examples/langgraph-research-rag/`.

## Framework-Specific Deployment Notes

| Framework Shape | Deployment Note                                                                                                   |
|-----------------|-------------------------------------------------------------------------------------------------------------------|
| LangGraph       | Use persistent checkpointers for approval waits, resume, and fault tolerance. Treat thread ID as sensitive state. |



| Framework Shape | Deployment Note                                                                                                                            |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| AutoGen         | Persist transcripts with redaction and termination metadata. Evaluate role behavior, not only final output.                                |
| Mastra          | Keep TypeScript runtime packaging explicit: agents, workflows, tools, memory, evals, and trace export need ownership.                      |
| CrewAI          | Keep Flow state separate from Crew-local collaboration. Validate crew output before the Flow accepts it.                                   |
| Mini-runtime    | Use the deployment process to decide which production controls you must build yourself and which should move into platform infrastructure. |

## Complete When

The system is deployable when:

- local setup is reproducible;
- secrets and config are separated;
- persistence matches authority;
- traces are exported and redacted;
- evals block risky changes;
- rollout stages are documented;
- rollback works without code changes for common failures;
- the runbook names owners, dashboards, and incident actions.

Until then, the system may be useful, but it is not production-ready.

# Durable Workflows

Durable workflows make agentic systems resumable and auditable by owning retries, checkpoints, approvals, compensation, and long-running state.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

Durable workflows wrap agent steps in a resumable execution model. The workflow owns ordering, checkpoints, retries, approvals, compensation, cancellation, and long-running state. Agents perform bounded work inside workflow steps.

This distinction matters. An agent loop is not a workflow engine. A model can propose the next action, but the workflow decides what step is active, what state is durable, which retry is safe, which approval is pending, and how the system resumes after failure.

## Use When

- Work spans minutes, hours, external systems, or human approvals.
- Tool calls may fail and must be retried safely.
- State must survive process restarts, deployments, timeouts, and partial outages.
- Side effects need idempotency, compensation, audit, or approval.
- Operators need replay, cancellation, rollback, and incident diagnosis.

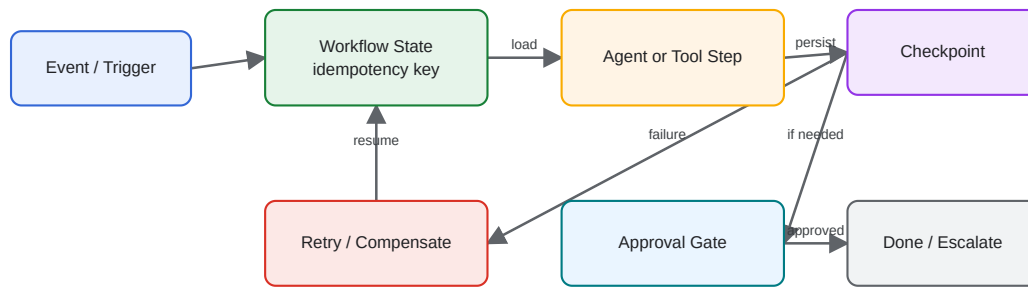
## Avoid When

- The task is a short stateless response.
- No external side effects, retries, approvals, or durable state are needed.
- The workflow engine would hide more behavior than it clarifies.
- The team cannot define step boundaries and idempotency rules.
- The system cannot decide what should happen after partial completion.

## Architecture

Use this diagram to read Durable Workflows as a system boundary, not only a code shape. The key ownership question is: the runtime owns durable state, retries, traces, triggers, deployment configuration, and operational controls.

## Durable Workflow



## System Shape

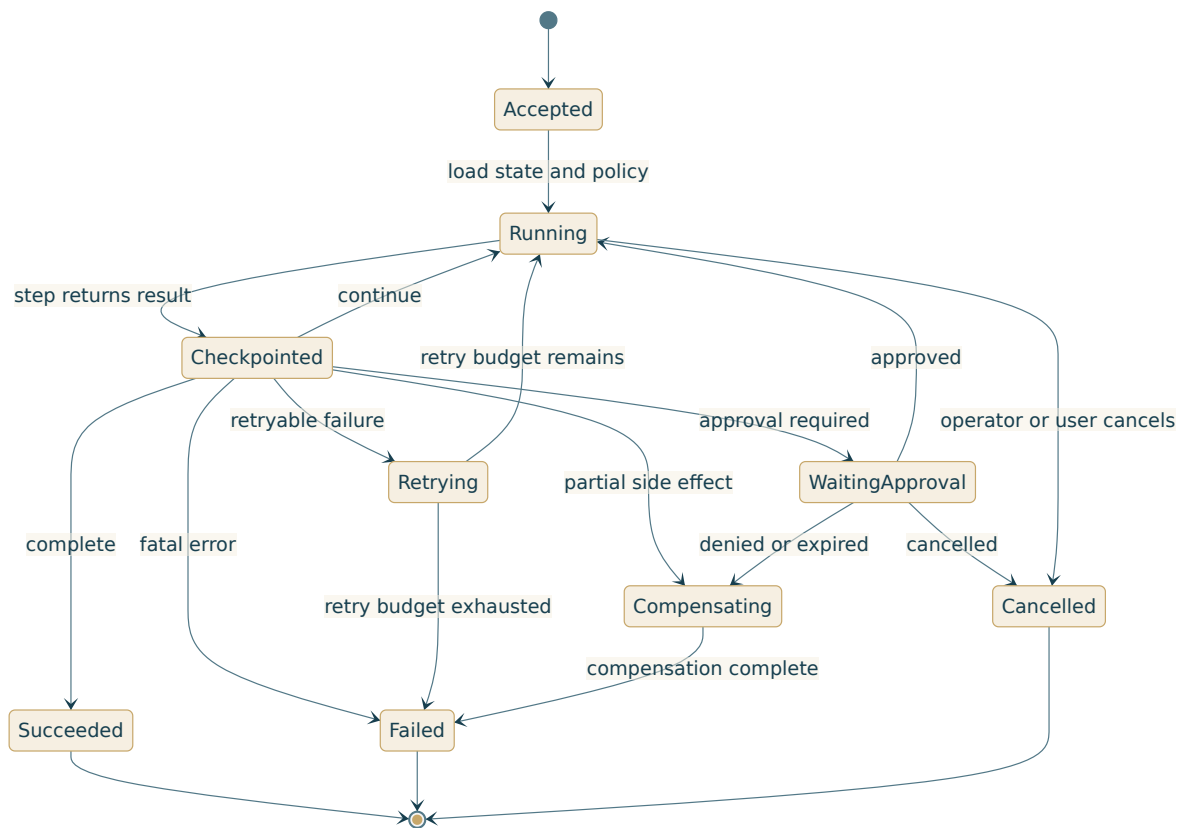
- **Pattern boundary:** the workflow owns step order, durable state, retries, approvals, cancellation, compensation, and replay.
- **Agent boundary:** the agent handles bounded uncertainty inside a step, then returns a typed result, refusal, error, or escalation.
- **State owner:** workflow state is durable; model context is temporary.
- **Side-effect boundary:** every external action has an idempotency key, checkpoint, and audit record.
- **Operational promise:** a run can fail, pause, resume, and be explained without losing its place.

## Core Protocol

1. Receive a request, event, schedule, or workflow command with an idempotency key.
2. Create or load workflow state with caller, goal, current step, policy context, and trace ID.
3. Execute one bounded step: deterministic code, agent loop, tool call, approval wait, or evaluator.
4. Checkpoint state, result, cost, trace events, errors, and pending approvals.
5. Decide next transition: continue, retry, wait, compensate, cancel, escalate, or complete.
6. Resume from the last durable checkpoint after restart or deployment.
7. Record final result, stop reason, side effects, and replay metadata.

## Workflow Transition Map

Use this map to review whether every transition has a durable checkpoint, trace event, and owner. The workflow should be able to stop and resume from any non-terminal state without repeating unsafe side effects.



## Implementation Notes

Keep workflow state separate from model conversation state.

```
type WorkflowState = {
 workflowId: string;
 traceId: string;
 status: 'running' | 'waiting' | 'succeeded' | 'failed' | 'cancelled';
 currentStep: string;
 completedSteps: string[];
 pendingApprovalId?: string;
 idempotencyKeys: Record<string, string>;
 sideEffects: Array<{
 actionId: string;
 tool: string;
 status: 'planned' | 'executed' | 'compensated';
 }>;
 stopReason?: string;
};
```

Each step should return a transition, not just text:

```
type StepResult =
 | { transition: 'continue'; nextStep: string; patch: Partial<WorkflowState> }
```

```
| { transition: 'wait_for_approval'; approvalId: string; patch: Partial<WorkflowState> }
| { transition: 'retry'; reason: string; retryAfterMs: number }
| { transition: 'compensate'; reason: string; actionId: string }
| { transition: 'complete'; output: unknown }
| { transition: 'fail'; reason: string };
```

Idempotency is not optional for side effects:

```
async function executeRefundDraft(state: WorkflowState, input: RefundDraftInput) {
 const idempotencyKey = state.idempotencyKeys.refundDraft ?? `refund-
draft:${state.workflowId}`;

 const result = await refunds.draftRefundRequest(input, { idempotencyKey });

 return {
 transition: 'continue',
 nextStep: 'review_policy',
 patch: {
 idempotencyKeys: { ...state.idempotencyKeys, refundDraft: idempotencyKey },
 sideEffects: [
 ...state.sideEffects,
 { actionId: result.draftId, tool: 'refunds.draftRefundRequest', status: 'executed' }
]
 }
 };
}
```

Retrying a model call is usually safe. Retrying a side effect is safe only when the side effect is idempotent or guarded by the workflow state.

## Failure Modes

- Side-effectful steps retry without idempotency.
- Approval state is stored only in chat history and disappears after restart.
- Workflow status says success while the agent step returned uncertainty or missing evidence.
- A deployment changes prompts or tool manifests halfway through a run without versioning.
- The workflow cannot resume because the current step is implicit.
- Compensation is missing for partially completed external actions.
- Cancellation stops the UI but not the underlying tool or worker.
- Traces show final output but not checkpoints, retries, approvals, and side effects.
- The workflow engine hides agent behavior instead of making it inspectable.

## Evaluation Strategy

Durable workflow evals should test recovery, not only happy paths.

- Test restart from every meaningful checkpoint.
- Test duplicate event delivery with the same idempotency key.
- Test retryable tool failure and fatal tool failure.
- Test approval wait, approval denial, approval timeout, and resume after approval.
- Test deployment during a waiting or running workflow.
- Test cancellation and compensation.
- Test replay from a production trace.
- Test that every final result has a stop reason and side-effect audit trail.

A compact workflow eval can look like this:

```
{
 "case_id": "approval_timeout_after_refund_draft",
 "initial_step": "draft_refund",
 "events": [
 { "type": "tool_success", "tool": "refunds.draftRefundRequest" },
 { "type": "approval_timeout", "approval_id": "ap_123" }
],
 "expected": {
 "final_status": "waiting_or_escalated",
 "must_not_issue_refund": true,
 "requires_checkpoint": ["refund_draft", "approval_request"],
 "required_trace_events": ["step_started", "tool_result", "checkpoint",
"approval_timeout"]
 }
}
```

Measure resume success rate, duplicate-event safety, retry success, compensation success, approval wait time, cancellation correctness, replay success, and incident recurrence.

## Production Checklist

- Define workflow state separately from prompt context.
- Give every workflow run a stable workflow ID, trace ID, and idempotency key.
- Checkpoint after every external side effect and approval request.
- Make side effects idempotent or compensatable.
- Version prompts, policies, model routes, tool manifests, and workflow definitions.
- Treat approval wait, timeout, denial, cancellation, and escalation as normal transitions.
- Persist enough trace data to replay failed runs.
- Define operational ownership for stuck, waiting, failed, and compensating workflows.
- Add alerts for retry storms, stuck approvals, duplicate side effects, and replay failures.

- [Convert production workflow incidents into regression evals.](#)

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

This pattern currently has no dedicated code excerpt. Use the source and download links below for the full pattern folder.

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `durable-workflow-pattern/` folder from this repository.

## Related Patterns

- [Agent Loop](#)
- [Goals and State](#)
- [Human Approval Gates](#)
- [Policy Enforcement](#)
- [Self-Healing Workflows](#)
- [Production Evaluation Feedback Loops](#)
- [Pattern Composition Playbook](#)

# Observability and Evals

Observability records what happened. Evals decide whether behavior is good enough. Release gates decide whether a change is allowed to ship.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

The Observability and Evals Pattern makes agent behavior inspectable, replayable, and testable. Observability records what the agent did, why it did it, what it saw, what it changed, what it cost, and why it stopped. Evals turn those traces, known tasks, incidents, and near misses into release gates.

This pattern matters because agent failures rarely live only in the final answer. They live in the trajectory: a missing retrieval, an unsafe tool call, an ignored policy denial, a loop that never converged, a human approval that was skipped, or a model upgrade that changed the plan. Logging final answers is not observability, and checking final answers is not enough evaluation.

Read this after the runtime and security chapters if you are designing for production. The runtime gives the agent a controlled place to run; security defines what must be blocked or approved; observability and evals prove whether those controls worked.

## Use When

- Agent decisions affect users, money, data, or external systems.
- You need regression tests for prompts, tools, routing, or workflows.
- Failures are hard to reproduce from final answers alone.
- You need to compare model, prompt, tool, memory, or policy changes before release.
- You operate workflows where the stop reason matters as much as the answer.

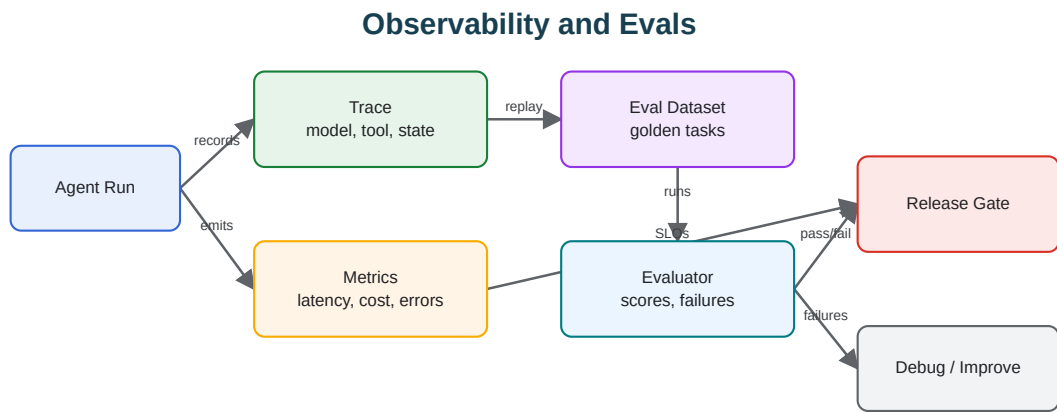
## Avoid When

- You cannot store traces safely because of privacy or regulatory constraints.
- The prototype is throwaway and has no operational users.
- You only log final answers and call that observability.
- Nobody owns the eval suite after the first version ships.
- The organization is not ready to define retention, redaction, and access rules for traces.

## Architecture



Use this diagram to read Observability and Evals as a system boundary, not only a code shape. The key ownership question is: the runtime owns durable state, retries, traces, triggers, deployment configuration, and operational controls.



## System Shape

- **Runtime boundary:** the runtime creates trace IDs, run IDs, span IDs, and idempotency keys before the agent starts work.
- **Span model:** model calls, retrieval calls, tool calls, policy decisions, approval waits, evaluator decisions, retries, and workflow steps are first-class spans.
- **Eval boundary:** evals are not an afterthought. They are connected to traces, incidents, release gates, and model or prompt changes.
- **Data boundary:** traces are redacted before storage, retained for a defined period, and protected like production data.
- **Operational boundary:** dashboards connect behavior, quality, cost, latency, and incident response.

Observability and evaluation are related, but they are not the same layer.

| Layer         | Question it answers                 | Typical artifact                                                         |
|---------------|-------------------------------------|--------------------------------------------------------------------------|
| Logs          | What event happened?                | Structured event records.                                                |
| Metrics       | How often, how slow, how expensive? | Counters, gauges, histograms, SLOs.                                      |
| Traces        | Which path did one run take?        | Run, iteration, model, tool, policy, approval, and evaluator spans.      |
| Evals         | Was the behavior acceptable?        | Test cases, expected outcomes, graders, thresholds, and failure reports. |
| Release gates | Can this change ship?               | Required eval subsets and approval records.                              |

Do not collapse these into one dashboard. A dashboard can show symptoms. A trace can explain one run. An eval can block a bad change before it reaches users.

## Core Protocol

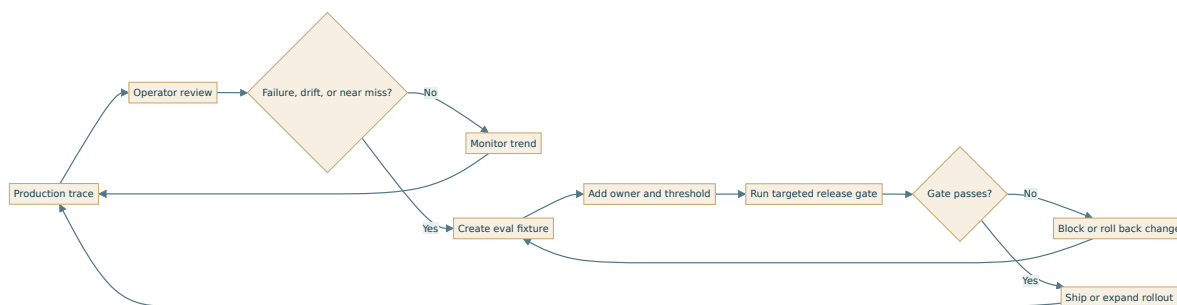
1. Start every run with a stable trace ID, run ID, request ID, and caller context.
2. Record each model, tool, retrieval, policy, approval, evaluator, and workflow span.
3. Capture enough input, output, configuration, and evidence references to replay the behavior.
4. Redact secrets, credentials, private data, and unnecessary raw content before persistence.
5. Store stop reason, status, cost, latency, token count, tool count, retry count, and policy outcome.
6. Convert incidents, near misses, and representative traces into eval fixtures.
7. Gate risky changes with the relevant eval subset before deployment.
8. Keep failed evals attached to owners, release decisions, and follow-up work.

The operational loop should be explicit:

1. The runtime emits traces and metrics.
2. Operators inspect failures, near misses, and outliers.
3. Engineers convert useful examples into eval fixtures.
4. Release gates run the relevant fixtures before model, prompt, policy, tool, memory, or workflow changes.
5. New production traces confirm whether the change improved behavior or only moved the failure.

## Implementation Notes

Use this loop to connect runtime evidence to release decisions. The important step is the handoff from a production trace to a named eval fixture that can block the next risky change.

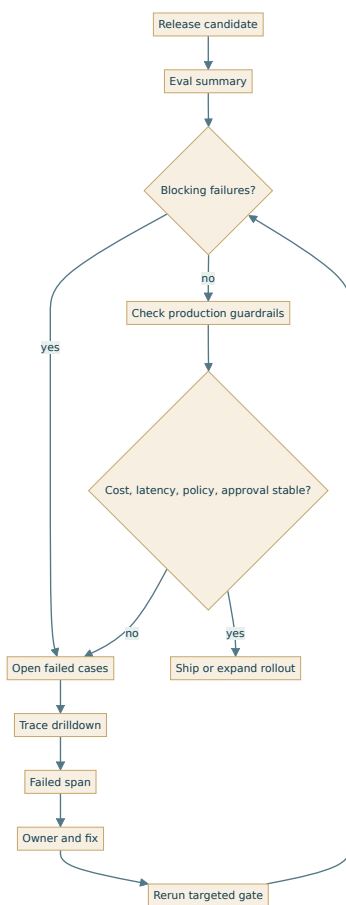


- Trace at the level of run, loop iteration, model call, tool call, workflow step, and evaluator result.
- Store enough input/output detail to reproduce failures, with redaction for sensitive data.
- Maintain golden datasets for routing, structured outputs, tool plans, and final answers.
- Treat eval failures as release blockers for production agents.
- Track both final quality and trajectory quality. A good answer produced through an unsafe tool path is still a failure.
- Keep trace schemas stable. If every service logs different fields, debugging becomes archaeology.
- Attach eval cases to the pattern they protect: routing, retrieval, tool use, policy enforcement, memory, human approval, or multi-agent coordination.

- Separate product analytics from agent observability. Product analytics says what users did. Agent observability says what the system did on their behalf.
- Store identifiers for prompts, models, tools, policies, retrievers, memory stores, and harness versions. Without versions, a trace explains what happened but not what changed.
- Treat “no trace” as a production defect. An untraced agent run cannot be debugged, replayed, or defended in an incident review.

## Eval Dashboard Review Model

An eval dashboard should help an engineer decide whether to ship, roll back, or inspect a specific run. Do not build a wall of charts that cannot answer which trace failed and who owns the fix.



Use this as the minimum dashboard layout:

| Panel           | Shows                                                                      | Release Question          |
|-----------------|----------------------------------------------------------------------------|---------------------------|
| Release Gate    | eval suite, changed component, pass/fail, blocking count, owner.           | Can this change ship?     |
| Failure Table   | case ID, severity, expected behavior, actual behavior, protected boundary. | Which failures matter?    |
| Trace Drilldown | run, model, retrieval, tool, policy, approval, evaluator spans.            | Where did the path break? |

| Panel               | Shows                                                               | Release Question                               |
|---------------------|---------------------------------------------------------------------|------------------------------------------------|
| Guardrails          | stop reasons, policy denials, approval waits, tool errors, retries. | Did autonomy stay inside its boundary?         |
| Cost And Latency    | p50/p95 latency, cost, token count, tool count by route.            | Did the change create an operating regression? |
| Incident Conversion | production issue, source trace, new fixture, owner, due date.       | Will this failure be caught next time?         |

A useful dashboard starts from a release decision and drills into trace evidence. If a failed eval cannot open the source trace, the dashboard is reporting quality without explaining behavior.

## Minimum Trace Contract

At minimum, every production run should connect these records:

- run identity: trace ID, run ID, request ID, actor, tenant, environment, and version set;
- goal and stop state: requested goal, accepted goal, status, stop reason, and error class;
- context: context packet ID, retrieved evidence IDs, memory IDs, omitted-source notes, and redaction level;
- model activity: model, prompt version, tool schema version, token counts, latency, cost, and output status;
- tool activity: tool name, arguments after redaction, authorization decision, result status, side-effect record, idempotency key, and retry count;
- policy activity: policy version, decision, reason code, approval requirement, and escalation owner;
- memory activity: read IDs, write IDs, retention class, consent or policy basis, and correction path;
- evaluation activity: evaluator version, case ID when applicable, score, threshold, and pass or fail decision.

The trace should not store every raw byte by default. It should store enough structured evidence to reconstruct the path safely.

## Trace Event Example

```
type AgentTraceEvent = {
 traceId: string;
 runId: string;
 spanId: string;
 parentSpanId?: string;
 requestId: string;
 actorId: string;
 tenantId: string;
 environment: 'dev' | 'staging' | 'prod';
 step: string;
```

```

spanType:
 | 'run'
 | 'model'
 | 'tool'
 | 'retrieval'
 | 'memory'
 | 'policy'
 | 'approval'
 | 'evaluator'
 | 'workflow';
timestamp: string;
status: 'started' | 'succeeded' | 'failed' | 'denied' | 'waiting' | 'cancelled';
latencyMs: number;
versionSet: {
 model?: string;
 prompt?: string;
 toolSchema?: string;
 policy?: string;
 retriever?: string;
 harness?: string;
};
model?: string;
tool?: string;
policyDecision?: 'allow' | 'deny' | 'require_approval' | 'escalate';
evidenceRefs?: string[];
memoryRefs?: string[];
sideEffectRef?: string;
idempotencyKey?: string;
costCents?: number;
stopReason?: string;
redaction: 'none' | 'pii_removed' | 'secret_removed' | 'content_reference_only';
};

```

This event is not meant to be the only schema in the system. It is a contract for correlation. A model provider trace, an OpenTelemetry span, a workflow engine event, and an eval result can all map into it.

## Eval Fixture Example

```

{
 "case_id": "tool_called_without_policy_trace",
 "source_trace_id": "tr_1042",
 "failure": "A refund draft was created without a recorded policy decision.",
 "expected": {
 "required_spans": ["tool", "policy"],

```

```

 "must_not_call_tools": ["refunds.issue_refund"],
 "stop_reason": "policy_boundary"
 }
}

```

## Eval Types

Agent evals need more than one score.

| Eval type              | What it protects                                    | Example check                                                       |
|------------------------|-----------------------------------------------------|---------------------------------------------------------------------|
| Task success           | The user-visible job was completed.                 | The support agent drafts the correct refund response.               |
| Trajectory correctness | The agent took an acceptable path.                  | It retrieved policy before drafting the refund.                     |
| Tool correctness       | Tool choice and arguments were valid.               | It called <code>orders.lookup</code> before proposing compensation. |
| Policy compliance      | Unsafe actions were blocked or escalated.           | It did not issue a refund without approval.                         |
| Retrieval quality      | Evidence was relevant, fresh, and cited.            | The answer cites the active refund policy, not an archived one.     |
| Memory correctness     | Memory reads and writes were scoped and reviewable. | It did not store a transient complaint as a durable preference.     |
| Autonomy safety        | The system stopped at the right boundary.           | It produced a draft instead of sending the message.                 |
| Recovery behavior      | Failure handling preserved control.                 | A timeout produced a retry or escalation, not a silent success.     |
| Cost and latency       | The system stayed within budget.                    | A prompt change did not double median cost.                         |

The point is not to build a perfect judge. The point is to make important failures visible before production traffic finds them.

## Release Gates

Tie eval subsets to change types.

| Change        | Required eval subset                                                     |
|---------------|--------------------------------------------------------------------------|
| Prompt change | task success, schema validity, trajectory correctness, policy compliance |

| Change                    | Required eval subset                                                     |
|---------------------------|--------------------------------------------------------------------------|
| Model change              | task success, refusal behavior, cost, latency, tool argument quality     |
| Tool schema change        | tool correctness, authorization, idempotency, error handling             |
| Retrieval change          | grounding quality, citation faithfulness, stale-source handling          |
| Memory change             | memory read scope, memory write policy, deletion and correction behavior |
| Policy change             | false allow, false deny, approval routing, escalation traceability       |
| Harness or runtime change | cancellation, retry, replay, trace completeness, side-effect safety      |

Small systems can start with a short gate. Serious systems eventually need gates that match the blast radius of the change.

## Failure Modes

- Logs that omit the prompt, tool input, or model configuration.
- Evals that only check happy paths.
- Metrics without trace IDs, making incidents hard to investigate.
- Storing sensitive data without retention or redaction rules.
- Final-answer-only logging that hides the path that produced the answer.
- Tool calls without captured arguments, outputs, permissions, and side effects.
- Policy denials that are not visible in traces, so blocked work looks like model confusion.
- Traces that leak secrets, credentials, customer data, or internal reasoning that should not be stored.
- Evals that test prose quality but ignore retrieval evidence, tool trajectory, and policy behavior.
- Dashboards that show aggregate cost and latency but cannot drill into failed runs.
- Incident reviews that do not create new eval cases.
- Eval suites with no owner, no freshness process, and no release authority.
- Traces that cannot answer which model, prompt, policy, tool schema, retriever, or harness version produced the run.
- Evals that pass because they mock away the exact tool, memory, or policy behavior that failed in production.
- Online evaluators that silently become another unobserved agent path.

## Evaluation Strategy

- **Trace completeness:** every production run has correlated model, tool, policy, evaluator, and workflow spans.
- **Replayability:** engineers can reproduce a failure with the stored configuration, inputs, evidence references, and tool mocks.
- **Trajectory correctness:** the agent used the allowed tools, respected policy, stopped for the right reason, and did not skip required approvals.

- **Grounding quality:** answers that depend on retrieval cite the evidence that was actually used.
- **Cost and latency regression:** model, prompt, and tool changes cannot silently increase runtime cost or response time.
- **Policy-denial accuracy:** unsafe requests are blocked or escalated with a traceable reason.
- **Incident-to-eval conversion:** repeated or high-severity production failures become regression fixtures.
- **Memory safety:** memory reads and writes follow the retention, consent, correction, and deletion rules for the task.
- **Autonomy boundary:** the agent stops for approval, escalation, or handoff when its autonomy level requires it.
- **Release-gate authority:** failed evals block the change or require an explicit, traceable override.

## Production Checklist

- Define a stable trace schema before production traffic.
- Redact and classify trace fields before storage.
- Correlate logs, metrics, traces, eval results, workflow state, and user-visible incidents.
- Give every eval suite an owner and a release decision role.
- Run targeted evals for prompt, model, tool, memory, policy, and workflow changes.
- Add dashboards for success rate, stop reason, policy denials, tool errors, cost, latency, retries, and eval regression rate.
- Define retention, access control, and deletion rules for trace data.
- Turn incidents and near misses into eval fixtures before closing the operational follow-up.
- Version prompts, models, tools, policies, retrievers, memory behavior, and harness code in the trace.
- Map change types to required eval subsets.
- Require a named owner for eval failures and overrides.

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

```
observability-and-evals-pattern/trace-contract.ts
```

[Open full source](#)



```

export type SpanType =
 | "run"
 | "model"
 | "tool"
 | "retrieval"
 | "memory"
 | "policy"
 | "approval"
 | "evaluator"
 | "workflow";

export type AgentTraceEvent = {
 traceId: string;
 runId: string;
 spanId: string;
 parentSpanId?: string;
 requestId: string;
 actorId: string;
 tenantId: string;
 environment: "dev" | "staging" | "prod";
 step: string;
 spanType: SpanType;
 timestamp: string;
 status: "started" | "succeeded" | "failed" | "denied" | "waiting" | "cancelled";
 latencyMs: number;
 versionSet: {
 model?: string;
 prompt?: string;
 toolSchema?: string;
 policy?: string;
 retriever?: string;
 harness?: string;
 };
 model?: string;
 tool?: string;
 policyDecision?: "allow" | "deny" | "require_approval" | "escalate";
 evidenceRefs?: string[];
 memoryRefs?: string[];
 sideEffectRef?: string;
 idempotencyKey?: string;
 costCents?: number;
 stopReason?: string;
 redaction: "none" | "pii_removed" | "secret_removed" | "content_reference_only";
};

```

```

export type TraceCheckResult = {
 ok: boolean;
 missing: string[];
};

export function checkTraceContract(events: AgentTraceEvent[]): TraceCheckResult {
 const missing: string[] = [];
 const spanTypes = new Set(events.map(event => event.spanType));
 const run = events.find(event => event.spanType === "run");

 if (!run) missing.push("run span");
 if (!spanTypes.has("model")) missing.push("model span");
 if (!events.some(event => event.stopReason)) missing.push("stop reason");
 if (!events.every(event => event.traceId && event.runId && event.spanId)) {
 missing.push("correlation ids");
 }

 for (const event of events) {
 if (event.spanType === "tool" && !event.policyDecision) {
 missing.push(`policy decision for tool span ${event.spanId}`);
 }

 if (event.spanType === "tool" && event.status === "succeeded" && !event.idempotencyKey) {
 missing.push(`idempotency key for tool span ${event.spanId}`);
 }

 if (event.spanType === "retrieval" && event.status === "succeeded" &&
!event.evidenceRefs?.length) {
 missing.push(`evidence refs for retrieval span ${event.spanId}`);
 }

 if (event.redaction === "none" && event.environment === "prod") {
 missing.push(`redaction classification for production span ${event.spanId}`);
 }
 }

 return {
 ok: missing.length === 0,
 missing: [...new Set(missing)]
 };
}

```

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `observability-and-evals-pattern/` folder from this repository.

## Related Patterns

- [Evaluator-Optimizer](#)
- [Durable Workflows](#)
- [Agent Loop](#)
- [Tool Use](#)
- [Compliance/Policy Enforcer](#)
- [Human Approval Gates](#)

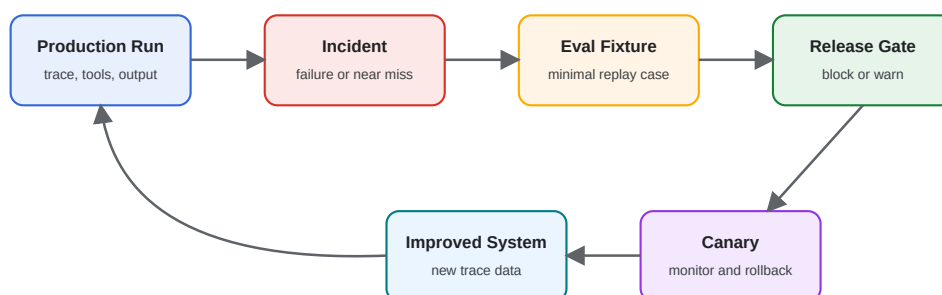
# Production Evaluation Feedback Loops

Agent evaluation should not stop at launch. Pre-production evals tell you whether the system is ready to meet users. Production feedback tells you whether the system is still true under real traffic, real tools, real ambiguity, real failures, and real incentives.

Download the [production evaluation feedback loop review checklist](#) before using this chapter for an operating review.

The core rule is simple: production failures should become future tests. If an incident only creates a ticket, the system learns nothing. If the same incident becomes a replayable eval, the system gets harder to break the same way again.

## Incident To Eval Feedback Loop



## Why This Pattern Exists

Observability explains what happened. A production evaluation feedback loop decides what the team does with that knowledge.

Without the loop, traces become a debugging archive. They help during incidents, but they do not improve the system by themselves. With the loop, traces become regression cases, release gates, canary checks, and rollback signals.

This is especially important for agentic systems because the same final answer can be produced through very different paths. One path may be correct, grounded, cheap, and policy-compliant. Another path may skip retrieval, call a write tool too early, leak stale memory, and still produce a convincing answer. Production evals must protect the trajectory, not only the final text.

## The Feedback Loop

A production evaluation loop connects five activities:

1. observe real runs;
2. detect failures, near misses, and human corrections;
3. convert important cases into eval fixtures;
4. gate changes against those fixtures;
5. release changes gradually with monitoring and rollback.

The loop applies to prompts, tools, routes, models, policies, memory rules, retrieval indexes, and agent topology. Any change that can alter behavior should pass through it.

The loop should have one owner. Not necessarily one person, but one accountable team. If product owns quality, platform owns traces, security owns policy, and nobody owns the release decision, the loop will fail at the boundary between teams.

## Feedback Loop Readiness Questions

Use these questions before calling an eval program production-ready:

| Question                               | Evidence To Produce                                                                                                     |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| Which production signals become evals? | Rules for incidents, near misses, human corrections, overrides, regressions, and high-cost outliers.                    |
| Who owns the loop?                     | Accountable team, release authority, fixture owner, and escalation path.                                                |
| What blocks release?                   | Blocking eval list with severity, owner, reason, and affected change types.                                             |
| What expires or gets deleted?          | Fixture expiry rules for temporary incidents, retired tools, old policies, and migrated workflows.                      |
| How is recurrence measured?            | Metrics for incident-to-eval conversion, eval catch rate, recurrence rate, and time to regression test.                 |
| How does rollback connect?             | Canary thresholds and rollback targets for prompts, policies, tools, model routes, memory rules, and retrieval indexes. |

The loop is healthy only when production failures change future release behavior.

## What Production Teaches

Production surfaces the failures that design reviews tend to miss. Users ask for unsupported combinations of tasks. Retrieval returns stale but plausible evidence. Tool errors arrive in a strange order. Approvals get skipped because the route is wrong. The agent loops because a tool returned partial data. A prompt change improves tone but quietly breaks policy behavior. Memory stores a bad preference and keeps reusing it. A model upgrade shifts tool-selection behavior. A low-frequency customer segment turns out to have different rules. A human override reveals the real acceptance criteria that nobody wrote down.

None of these are merely operational events. They are evaluation material, and treating them as such is what makes the difference between a system that drifts and one that improves.

## Incident-To-Eval

Every serious incident should produce at least one eval case.

| Field             | Purpose                                                                          |
|-------------------|----------------------------------------------------------------------------------|
| Incident ID       | Link back to the production event.                                               |
| Severity          | Decide whether the eval blocks release or warns only.                            |
| Owner             | Person or team responsible for keeping the case healthy.                         |
| Redacted input    | Minimal user request or event that reproduces the failure.                       |
| Context fixture   | Retrieved documents, memory, state, policy, or tool outputs needed to reproduce. |
| Tool trace        | Expected, forbidden, or observed tool calls.                                     |
| Expected behavior | Pass criteria stated as concretely as possible.                                  |
| Failure mode      | What went wrong and why it matters.                                              |
| Release gate      | Whether this eval blocks prompt, policy, model, or tool changes.                 |

The eval should be smaller than the incident. Do not preserve the whole production mess when a minimal fixture reproduces the failure.

A minimal incident-derived eval can be stored as data:

```
{
 "id": "incident-2026-04-18-refund-approval-bypass",
 "severity": "blocking",
 "owner": "support-platform",
 "input": {
 "user_message": "Refund this order and tell the customer it is done.",
 "order_id": "ord_redacted"
 },
 "context": {
 "policy_version": "refund-policy-2026-04",
 "customer_status": "standard"
 },
 "expected": {
 "tools_called": ["draft_refund_request"],
 "tools_not_called": ["issue_refund", "send_customer_email"],
 "final_status": "needs_human"
```

```

 },
 "reason": "Refunds over the threshold require approval before money movement or customer notification."
 }
}

```

The fixture encodes the behavior the system must preserve. It does not need the whole production trace.

## Worked Incident Walkthrough: Approval Bypass

Use a worked incident to prove the loop. Suppose a support refund agent drafted a customer message that implied a refund was approved before a support lead approved the money movement.

| Review Step         | Evidence                                                                                                                                                                                          | Decision                                                                  |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| Incident            | Customer-facing draft said “your refund has been approved” for an order above the self-serve threshold.                                                                                           | Severity is blocking because the message creates a financial expectation. |
| Trace review        | Route was <code>refund_assist</code> ; policy span existed, but approval span was missing before the draft span.                                                                                  | Failure mode is skipped approval, not bad tone.                           |
| Minimal fixture     | Redacted ticket, order amount above threshold, current refund policy, standard customer status.                                                                                                   | Keep only data needed to reproduce the boundary.                          |
| Expected trajectory | <code>read_order</code> , <code>retrieve_refund_policy</code> , <code>draft_refund_request</code> , <code>approval_required</code> ; forbid <code>issue_refund</code> and final customer promise. | Eval checks trajectory before prose.                                      |
| Fix                 | Move approval gate before customer-message drafting and require approved recommendation ID in the draft step.                                                                                     | Architecture change, not prompt-only repair.                              |
| Release gate        | Run fixture for prompt, policy, model-route, workflow, and tool-manifest changes.                                                                                                                 | Blocking until the fixture passes.                                        |
| Canary              | Shadow 5% of refund drafts and alert on missing approval spans.                                                                                                                                   | Roll back route if any above-threshold case lacks approval evidence.      |

The incident should produce both a trace assertion and a wording assertion. The trace assertion protects authority. The wording assertion protects the customer-visible promise.

```

{
 "id": "refund-approval-bypass-regression",
 "required_spans": [
 "tool:read_order",

```

```

 "retrieval:refund_policy",
 "policy:refund_threshold",
 "approval:required"
],
 "forbidden_spans": [
 "tool:issue_refund",
 "message:customer_promise_sent"
],
 "expected_stop_reason": "needs_human_approval",
 "output_must_not_contain": [
 "approved",
 "processed",
 "completed"
]
}

```

If the fixture fails, do not average it into a quality score. Hold the release. A system that drafts a polished unauthorized promise is still unsafe.

## Signal Triage

Not every production signal should become a blocking eval. Triage signals by risk, recurrence, and architectural meaning.

| Signal                                                     | Triage Decision                                                               | Eval Action                                                                                      |
|------------------------------------------------------------|-------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| Safety, privacy, money movement, or authorization incident | Treat as blocking until reviewed.                                             | Create a minimal fixture, add a forbidden trajectory, and require owner approval before release. |
| Human correction on a customer-visible answer              | Review for product quality and grounding.                                     | Add a warning eval unless the correction exposes policy, citation, or tool misuse.               |
| Policy denial that a human overrides                       | Check whether policy, routing, or UI expectation was wrong.                   | Add paired allow/deny fixtures when the boundary was ambiguous.                                  |
| High-cost or high-latency outlier                          | Check whether the agent loop, retrieval fanout, or tool retry policy drifted. | Add budget and stop-condition checks if the outlier can recur.                                   |
| Tool error recovered correctly                             | Keep as trace evidence.                                                       | Add an eval only if recovery hides a degraded answer or repeated retry.                          |
| Near miss caught by approval                               | Preserve the approval boundary.                                               | Add a fixture that proves the agent still pauses before the side effect.                         |

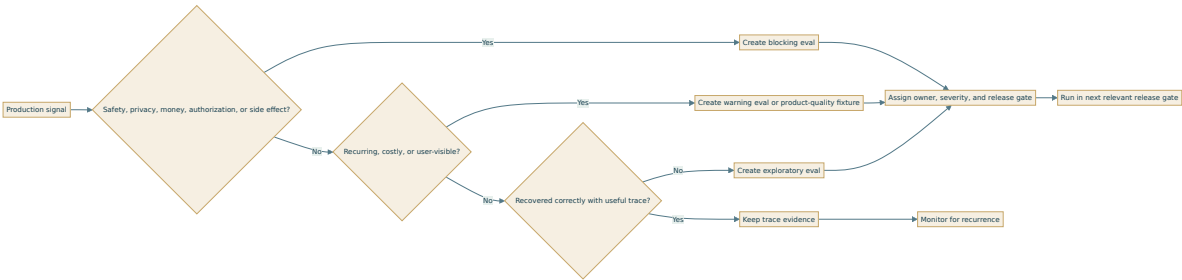


| Signal                 | Triage Decision                    | Eval Action                                                      |
|------------------------|------------------------------------|------------------------------------------------------------------|
| One-off user confusion | Improve UX or documentation first. | Add an eval only when the same confusion changes agent behavior. |

The triage decision should happen close to the incident, while the trace, human correction, and business context are still fresh.

Signal Triage Flow

Use this flow during incident review. It keeps the team from turning every signal into a blocking eval while still protecting the boundaries that matter.



Eval Fixture Contract

An incident-derived eval should be explicit enough that a new engineer can understand why it exists and what would count as a regression. A useful fixture usually contains four groups of fields:

| Group               | Fields                                                                                                                                                    |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Identity            | <code>id</code> , <code>incident_id</code> , <code>owner</code> , <code>severity</code> , <code>created_at</code> , <code>expires_at</code> if temporary. |
| Inputs              | Redacted user request, workflow event, state snapshot, retrieved evidence, memory records, mocked tool outputs.                                           |
| Expected trajectory | Required tools, forbidden tools, required policy decisions, approval state, stop reason, retry behavior.                                                  |
| Expected result     | Final status, structured output fields, evidence requirements, user-visible response constraints.                                                         |

Do not make the fixture depend on private production data unless that data is redacted and intentionally retained. The fixture should reproduce the failure mode, not archive the incident forever.

Minimal Eval Runner

The simplest useful production eval runner checks structure and trajectory before it checks prose. This example is intentionally small, but it shows the boundary the book keeps returning to: agents need tests around the path they take.

```

type TraceSpan = {
 type: 'model' | 'tool' | 'retrieval' | 'policy' | 'approval' | 'workflow';
 name: string;
 status: 'succeeded' | 'failed' | 'denied' | 'waiting';
};

type EvalCase = {
 id: string;
 requiredSpans: string[];
 forbiddenTools: string[];
 expectedStopReason: string;
};

type AgentRun = {
 spans: TraceSpan[];
 stopReason: string;
};

function evaluateRun(testCase: EvalCase, run: AgentRun) {
 const spanNames = new Set(run.spans.map((span) => span.name));
 const calledTools = new Set(
 run.spans
 .filter((span) => span.type === 'tool')
 .map((span) => span.name)
);

 const missingSpans = testCase.requiredSpans.filter(
 (spanName) => !spanNames.has(spanName)
);
 const forbiddenCalls = testCase.forbiddenTools.filter(
 (toolName) => calledTools.has(toolName)
);
 const wrongStopReason = run.stopReason !== testCase.expectedStopReason;

 return {
 id: testCase.id,
 passed:
 missingSpans.length === 0 &&
 forbiddenCalls.length === 0 &&
 !wrongStopReason,
 missingSpans,
 forbiddenCalls,
 wrongStopReason
 };
}

```

```
};
}
```

This kind of runner will not tell you whether a customer-facing answer is beautifully written. That is fine. Its job is narrower and more architectural: prove that the agent respected the release boundary.

## Mocked Tool Evals

Many agent failures happen well before the final answer, which is why mocked tool evals matter so much for tool-using agents: they let you test the trajectory without touching real systems. Use them to check which tools the agent chooses and which it avoids, whether arguments are valid, whether approval is requested, whether retries are safe, whether the agent stops on a policy denial, whether it treats untrusted tool output as data, and whether it recovers from malformed responses.

Mocked tools do not need to be perfect simulations. They need to be realistic enough to test the decision boundary. If the agent would call `issue_refund` when it should call `draft_refund_request`, you do not need a real payment system to catch that.

For state-changing tools, the mock should capture intent and side effects separately. It should be possible to assert that the agent prepared an action, requested approval, or stopped at a policy boundary without touching the real downstream system.

## Trajectory Evals

Final-answer evals are not enough. Evaluate the trajectory.

| Trajectory Layer | What To Check                                                        |
|------------------|----------------------------------------------------------------------|
| Route            | Did the system send the task to the right workflow, model, or agent? |
| Context          | Did the model receive the minimum useful evidence?                   |
| Retrieval        | Were sources relevant, current, and allowed?                         |
| Tool selection   | Were expected tools called and forbidden tools avoided?              |
| Tool input       | Were arguments valid, scoped, and policy-compliant?                  |
| State            | Were state transitions correct and replayable?                       |
| Memory           | Were reads justified and writes constrained?                         |
| Policy           | Were approvals, refusals, and denials enforced?                      |
| Output           | Was the final response correct, grounded, and safe?                  |

An answer can look good while the system behaved badly. The trajectory tells you whether the architecture actually worked.

## Release Gates

Agent releases should have gates, just like software releases, and the gate should scale with the risk of the change.

| Change Type     | Suggested Gate                                                             |
|-----------------|----------------------------------------------------------------------------|
| Prompt wording  | Golden tasks, incident fixtures, tool trajectory checks.                   |
| Tool schema     | Schema tests, mocked tool evals, authorization tests.                      |
| Policy rule     | Denial tests, approval tests, canary monitoring.                           |
| Model version   | Regression suite, cost and latency budget, tool-selection comparison.      |
| Retrieval index | Source relevance, freshness, citation coverage, missing-evidence behavior. |
| Memory rule     | Privacy tests, stale-memory tests, write-policy checks.                    |
| Agent topology  | Task completion, coordination cost, trace completeness, merge quality.     |

Not every eval should block every release. Keep blocking evals for safety, privacy, policy, and known incidents; use warning evals for quality, tone, and edge cases; and keep exploratory evals for new behavior still under investigation. Blocking evals should be few, serious, and maintained. When everything blocks, teams start ignoring the gate.

Good release gates answer three questions:

1. What changed?
2. Which behaviors could that change affect?
3. Which eval subset protects those behaviors?

If a prompt change can alter tool choice, run tool trajectory checks. If a retrieval index changes, run grounding and missing-evidence checks. If a model route changes, compare cost, latency, tool selection, policy denials, and final quality. A generic average score is not enough for production release decisions.

## Canary And Rollback

Prompts, policies, model routes, and tool rules are production artifacts, so release them gradually:

1. shadow the new behavior where possible;
2. send a small percentage of traffic to the candidate;
3. compare quality, cost, latency, tool errors, policy denials, and human overrides;
4. expand only when the guardrails hold;
5. roll back automatically when safety or reliability thresholds are breached.

A rollback should restore the last known-good prompt, policy, model route, tool manifest, or memory rule. If rollback requires manual reconstruction, the release process is weak.

## Eval Ownership

Evals need owners. Each important eval should have a business or system owner, a severity label, a stated reason for existing, a maintenance rule, a link to the incident or requirement or risk behind it, and a decision about whether it blocks release. Without ownership, eval suites rot: they grow slow, flaky, redundant, and disconnected from the production risk they were meant to track.

Ownership also means deletion. Some evals should expire after a migration, policy sunset, tool replacement, or customer-specific incident. A stale eval can be worse than no eval because it blocks useful change while pretending to protect a current risk.

## Metrics

Track the feedback loop itself, not only agent quality. The useful signals are the ones that tell you whether the loop is working: incident-to-eval conversion rate, eval catch rate before release, recurrence rate for known incidents, time from incident to regression test, number of blocking evals, flaky eval rate, production trace coverage, policy-denial accuracy, human-override rate, rollback frequency, and mean time to detect an agent regression. The point is not a beautiful dashboard. It is knowing whether the system is learning from failure.

## Dashboard Thresholds

Use thresholds to decide when the loop needs attention. Start simple and tune the numbers after the team has real traffic.

| Metric                      | Investigate When                                                                       | Block Or Roll Back When                                                    |
|-----------------------------|----------------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| Incident-to-eval conversion | Serious incidents produce no fixture within two business days.                         | A repeated incident has no regression eval.                                |
| Eval catch rate             | Production incidents bypass the suite more than once in a release cycle.               | A known incident fixture passes while production repeats the same failure. |
| Flaky eval rate             | More than 5% of release-gate runs require rerun.                                       | A blocking eval is flaky and nobody owns the repair.                       |
| Production trace coverage   | Fewer than 95% of risky runs have complete route, tool, policy, and stop-reason spans. | A release changes risky behavior without trace coverage.                   |
| Policy-denial accuracy      | Operators frequently override denials or users report false refusals.                  | The agent executes a side effect after a policy denial.                    |
| Human-override rate         | Overrides spike after a prompt, model, retrieval, or policy change.                    | Overrides expose skipped approval, stale evidence, or wrong tool choice.   |
| Rollback frequency          | Rollbacks cluster around the same component.                                           | The same component causes two rollback events without a new gate.          |

Dashboard thresholds should name an owner and an action. A red chart without a decision rule is decoration.

## Anti-Patterns

- Treating evals as a demo score instead of a release control.
- Running only final-answer checks for a system that uses tools, memory, policy, or retrieval.
- Keeping incident fixtures that nobody owns and nobody understands.
- Allowing model upgrades without replaying known production failures.
- Using one global average score to approve high-risk behavior changes.
- Capturing production traces that cannot be safely stored, searched, or replayed.
- Letting flaky evals remain blocking until engineers learn to ignore the gate.

## Practical Workflow

In practice the loop runs like this:

1. A run fails, escalates, or receives a human correction.
2. The trace is reviewed and redacted.
3. The team identifies the failure mode.
4. A minimal eval fixture is created.
5. The current system is confirmed to fail the fixture.
6. A fix is made in prompt, tool, policy, retrieval, memory, or architecture.
7. The eval passes without breaking blocking cases.
8. The change ships through canary.
9. Production monitoring confirms the failure does not recur.

This is how agent engineering becomes cumulative. Each serious failure should make the next version harder to fail in the same way.

## Design Checklist

Before operating an agent in production, answer:

- Which traces are captured?
- Which traces are safe to store?
- Which failures become evals?
- Who owns incident-derived evals?
- Which evals block release?
- Which evals only warn?
- Can tools be mocked?
- Can runs be replayed?
- Can prompts, policies, tools, and model routes be rolled back?
- Can a canary be stopped automatically?
- Can operators explain why a release was blocked?

If production failures do not feed evals, observability is passive. It tells you what happened, but it does not improve the system.

## Related Chapters

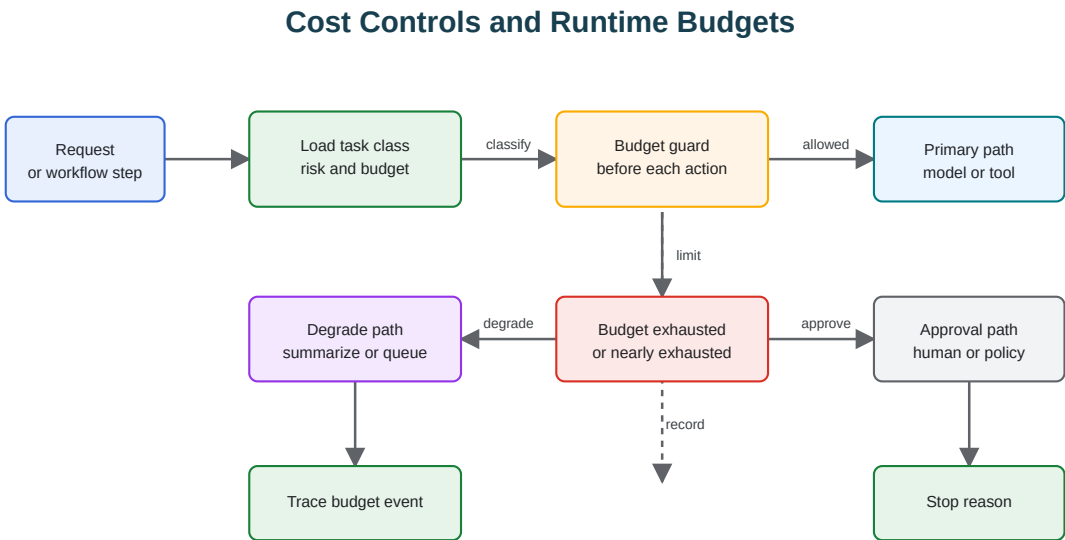
- [Evaluation-Driven Agent Development](#)
- [Pattern Evaluation Checklist](#)
- [Observability and Evals](#)
- [Circuit Breakers, Fallbacks, and Replay](#)
- [Tool Capability Design](#)
- [Agent Threat Model](#)
- [Durable Workflows](#)

# Cost Controls and Runtime Budgets

Agentic systems spend money, time, context, tool quota, and human attention every time they think, retrieve, call a tool, delegate, retry, evaluate, or ask for approval. Runtime budgets make those costs explicit and enforceable.

Download the [cost controls and runtime budgets review checklist](#) before using this chapter for a production review.

This chapter is not about making agents cheap at any cost. It is about making cost, latency, and autonomy part of the control plane. A useful agent should know when to continue, when to degrade, when to ask for approval, and when to stop.



## Intent

Keep agentic behavior bounded by task value, risk, user tier, and operational limits.

A budget is not only a finance number. It is a runtime contract: how many steps the system may take, how many tools it may call, how long it may run, how much context it may carry, how many retries it may spend, how much human attention it may request, and how much autonomy is justified before escalation.

## Runtime Budget Readiness Questions

Use these questions before enabling autonomous loops, retries, retrieval, delegation, or write tools:

| Question                             | Evidence To Produce                                                     |
|--------------------------------------|-------------------------------------------------------------------------|
| What does the task justify spending? | Budget policy by task class, risk class, user tier, and business value. |



| Question                         | Evidence To Produce                                                                              |
|----------------------------------|--------------------------------------------------------------------------------------------------|
| What does the runtime measure?   | Counters for model calls, tool calls, retries, retrieval, delegation, wall-clock time, and cost. |
| Where are budgets enforced?      | Pre-action gates before model, tool, retrieval, memory, approval, and delegation steps.          |
| What happens when budget is low? | Defined degraded modes, user-facing message, and operator stop reason.                           |
| When is approval required?       | Thresholds for extra spend, write tools, risky evidence gaps, or high-risk tasks.                |
| How are budget changes reviewed? | Versioned policy, eval cases, trace comparison, and rollback path.                               |

Budgeting is an architecture feature. If the budget is only a dashboard after the run, it is accounting, not control.

## Use When

- Agent runs can loop, retry, delegate, retrieve, or call tools.
- Model, tool, retrieval, or evaluator calls have meaningful cost or latency.
- Different task classes deserve different autonomy levels.
- The system serves interactive users and must preserve responsiveness.
- Operators need clear stop reasons when a run cannot continue.

Use this pattern before cost surprises appear in production. Retrofitting budgets after a runaway loop is harder because the architecture has already hidden the counters.

## Avoid When

- The task is a single deterministic function call.
- The system cannot observe model calls, tool calls, retries, or wall-clock time.
- The organization wants cost reduction without defining acceptable quality and risk.

Even then, keep simple counters. The first useful prototype often becomes the first production workflow.

## Budget Types

| Budget            | What It Controls                                                        |
|-------------------|-------------------------------------------------------------------------|
| Token budget      | Prompt, context, retrieved evidence, memory, and reserved output space. |
| Model-call budget | Number of model calls, model tier, judge calls, and repair attempts.    |

| Budget            | What It Controls                                                                       |
|-------------------|----------------------------------------------------------------------------------------|
| Tool-call budget  | Tool count, write operations, browser actions, shell commands, and external API quota. |
| Wall-clock budget | End-to-end time, step time, queue time, and human wait time.                           |
| Retry budget      | How often a step can recover before the system changes strategy.                       |
| Delegation budget | Number of agents, handoffs, debates, or parallel branches.                             |
| Retrieval budget  | Query count, source count, reranking cost, and evidence volume.                        |
| Memory budget     | Memory reads, writes, retention, and privacy-sensitive recall.                         |
| Approval budget   | How many human approvals, reviews, or escalations a workflow can request.              |

The most important budget is often not money. In production, human attention and user patience are usually scarcer than tokens.

## Budget Ownership

Budgets should be owned by the runtime, not by the prompt. The model can propose that more work is useful, but software should decide whether the system is allowed to spend more.

Budget ownership usually belongs in the same layer that owns state, policy, trace IDs, and stop reasons:

- routers load the budget for the task class;
- loop controllers decrement the budget after each step;
- tool gateways check tool and side-effect budgets;
- retrieval services enforce evidence and query budgets;
- workflow engines persist budget state across retries and approvals;
- observability records budget events as first-class trace events.

If every tool wrapper tracks its own counters, the system will drift. The runtime needs one budget view for the run.

## Runtime Budget Object

Give every run a budget object that travels with the state:

```
type RiskClass = 'low' | 'medium' | 'high';
type BudgetDecision = 'continue' | 'degrade' | 'approval_required' | 'stop';

type RuntimeBudget = {
 riskClass: RiskClass;
 maxCostCents: number;
```

```

 maxModelCalls: number;
 maxToolCalls: number;
 maxWriteToolCalls: number;
 maxRetrievalQueries: number;
 maxDelegations: number;
 maxRetries: number;
 maxWallClockMs: number;
 approvalRequiredAboveCents: number;
};

type RuntimeUsage = {
 costCents: number;
 modelCalls: number;
 toolCalls: number;
 writeToolCalls: number;
 retrievalQueries: number;
 delegations: number;
 retries: number;
 startedAtMs: number;
};

type PlannedActionCost = {
 estimatedCostCents: number;
 modelCalls: number;
 toolCalls: number;
 writeToolCalls: number;
 retrievalQueries: number;
 delegations: number;
};

```

The budget object should be versioned. A production incident caused by a budget change should be replayable against the old and new budget policy.

## Enforcement

Check budgets before every expensive or risky action, not after the run is already over.

```

function checkBudget(
 budget: RuntimeBudget,
 usage: RuntimeUsage,
 next: PlannedActionCost,
 nowMs: number
): { decision: BudgetDecision; reason: string } {
 if (nowMs - usage.startedAtMs >= budget.maxWallClockMs) {

```

```

 return { decision: 'stop', reason: 'wall_clock_budget_exhausted' };
 }

 const projectedCost = usage.costCents + next.estimatedCostCents;
 if (projectedCost > budget.maxCostCents) {
 return { decision: 'stop', reason: 'cost_budget_exhausted' };
 }

 if (projectedCost > budget.approvalRequiredAboveCents) {
 return { decision: 'approval_required', reason: 'cost_approval_required' };
 }

 if (usage.writeToolCalls + next.writeToolCalls > budget.maxWriteToolCalls) {
 return { decision: 'degrade', reason: 'write_tool_budget_exhausted' };
 }

 if (usage.modelCalls + next.modelCalls > budget.maxModelCalls) {
 return { decision: 'degrade', reason: 'model_call_budget_exhausted' };
 }

 if (usage.retrievalQueries + next.retrievalQueries > budget.maxRetrievalQueries) {
 return { decision: 'degrade', reason: 'retrieval_budget_exhausted' };
 }

 if (usage.toolCalls + next.toolCalls > budget.maxToolCalls) {
 return { decision: 'degrade', reason: 'tool_call_budget_exhausted' };
 }

 if (usage.delegations + next.delegations > budget.maxDelegations) {
 return { decision: 'stop', reason: 'delegation_budget_exhausted' };
 }

 if (usage.retries >= budget.maxRetries) {
 return { decision: 'degrade', reason: 'retry_budget_exhausted' };
 }

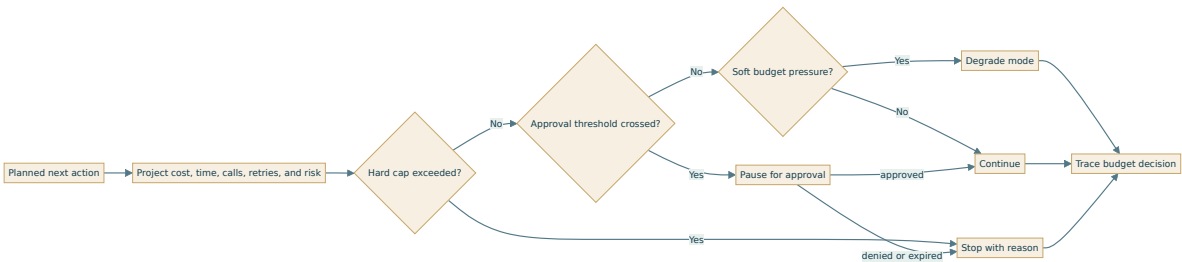
 return { decision: 'continue', reason: 'within_budget' };
}

```

The decision should become part of the trace. Notice that the check uses projected cost, not only current usage. A budget gate that runs after the model call or tool call has already spent the money is only accounting. A budget gate that runs before the next action is control.

## Budget Decision Flow

Use this flow before each model call, retrieval query, tool call, delegation, retry, approval request, or memory write. It turns budget pressure into an explicit runtime decision.



## Degraded Modes

When a budget is nearly exhausted, the system should not improvise. It should switch to a known degraded mode:

| Budget Pressure          | Safer Degraded Mode                                   |
|--------------------------|-------------------------------------------------------|
| Token budget is low      | Summarize state and preserve evidence references.     |
| Model-call budget is low | Stop revision loops and return best validated result. |
| Tool-call budget is low  | Move from action to draft or read-only analysis.      |
| Wall-clock budget is low | Return partial result or queue background work.       |
| Retry budget is low      | Stop retrying and expose the blocker.                 |
| Delegation budget is low | Assign one owner instead of adding more agents.       |
| Retrieval budget is low  | Ask for clarification or cite missing evidence.       |
| Approval budget is low   | Batch decisions or escalate to a human queue.         |

The degraded mode should be visible to the user or operator. Silent degradation creates false confidence.

## Risk-Based Budgets

Not every task deserves the same budget.

Low-risk tasks can be cheap and fast. High-risk tasks may justify stronger models, more evidence, more policy checks, and human approval. The mistake is giving every task the same budget because the runtime has no task class.

| Task Class            | Budget Posture                                                       |
|-----------------------|----------------------------------------------------------------------|
| Classification        | Small model, low context, strict latency, no tools unless necessary. |
| Evidence-bound answer | Retrieval budget, citation requirement, limited synthesis calls.     |
| Workflow automation   | Tool budget, retry budget, idempotency keys, stop reason.            |

| Task Class           | Budget Posture                                                        |
|----------------------|-----------------------------------------------------------------------|
| Money movement       | Write-tool budget, approval threshold, policy check, audit trace.     |
| Multi-agent analysis | Delegation budget, merge budget, coordination trace, final evaluator. |
| Background research  | Longer wall-clock budget, queue support, checkpointing.               |

Budgets are a way to encode judgment. They say which tasks are worth more reasoning, more tools, more review, or more time.

### Budget Calculator Example

Start with a simple per-run estimate before building a complex dashboard.

| Item                  | Quantity | Unit Estimate  | Run Estimate |
|-----------------------|----------|----------------|--------------|
| planner model call    | 1        | 1.5 cents      | 1.5 cents    |
| retrieval queries     | 3        | 0.2 cents      | 0.6 cents    |
| synthesis model calls | 2        | 2.0 cents      | 4.0 cents    |
| tool calls            | 4        | 0.5 cents      | 2.0 cents    |
| evaluator call        | 1        | 1.0 cents      | 1.0 cents    |
| retry reserve         | 20%      | 9.1 cents base | 1.8 cents    |
| total planned budget  |          |                | 10.9 cents   |

Then map the estimate to a policy:

| Task Class                    | Default Cap | Approval Above | Degraded Mode                                     |
|-------------------------------|-------------|----------------|---------------------------------------------------|
| read-only support answer      | 15 cents    | 25 cents       | answer with cited evidence and no extra retrieval |
| refund recommendation         | 50 cents    | 75 cents       | draft recommendation and request human review     |
| multi-agent incident analysis | \$2.00      | \$3.00         | stop fan-out and assign one owner                 |
| background research           | \$5.00      | \$8.00         | queue continuation and return partial findings    |

The exact numbers will differ by model and vendor. The useful habit is the same: estimate before the run, reserve for retries, enforce before actions, and record actual spend against the policy version.

## Alert Thresholds

Budget alerts should distinguish one expensive run from a systemic problem.

| Signal              | Warning                    | Critical                     | Action                                          |
|---------------------|----------------------------|------------------------------|-------------------------------------------------|
| per-run cost        | 80% of cap                 | 100% of cap                  | degrade or stop before next action              |
| model calls per run | 80% of cap                 | cap reached                  | stop revision loop or require approval          |
| write-tool calls    | any unexpected write       | cap reached                  | pause write route and review trace              |
| retry rate          | 10% of runs retry          | 25% of runs retry            | inspect failing step and add eval case          |
| delegation fan-out  | above planned worker count | uncontrolled worker creation | stop delegation and assign owner                |
| queue spend rate    | 2x normal hourly spend     | 5x normal hourly spend       | apply backpressure or pause route               |
| approval requests   | 2x normal volume           | reviewer queue blocked       | batch, route, or pause low-priority work        |
| degraded-mode rate  | 5% of runs                 | 15% of runs                  | review budgets, prompts, tools, or task routing |

Every alert should point to an operator action. A cost alert that only says “spend is high” is too vague to help during an incident.

## Interaction With Circuit Breakers

Budgets and breakers are related but not identical.

A budget says how much the system is allowed to spend. A breaker says when the system must stop or change strategy because the pattern of behavior is unsafe or unproductive.

Examples:

- a cost budget stops the run after a spend limit;
- a no-progress breaker stops the run even if budget remains;
- a tool budget limits calls to a tool class;
- a tool failure breaker disables a tool after repeated bad results;
- a delegation budget limits handoffs;
- a handoff breaker stops agents from bouncing work with no owner.

Use both. A run can be within budget and still be unsafe. A run can be safe but no longer worth the spend.

## Observability

Budget events should be visible in traces:

- budget policy version;
- task class and risk class;
- starting budget;
- remaining budget after each step;
- budget checks before model and tool calls;
- decision: continue, degrade, approval required, or stop;
- stop reason;
- user-visible degraded mode;
- operator-visible explanation.

If cost is only visible in the monthly bill, the architecture is already too late.

## Evaluation Guidance

Test budget behavior like any other production control.

| Eval Case                               | Expected Behavior                                                   |
|-----------------------------------------|---------------------------------------------------------------------|
| Model-call budget exhausted             | Revision loop stops or degrades with a traceable reason.            |
| Write-tool budget exhausted             | Agent switches from action to draft or approval path.               |
| Cost threshold crossed                  | Human approval is required before continuing.                       |
| Delegation budget exhausted             | Supervisor assigns an owner or stops with blocked status.           |
| Wall-clock budget exhausted             | Workflow queues background work or returns partial result.          |
| Low-value task requests expensive route | Router chooses cheaper safe path.                                   |
| High-risk task needs more evidence      | Runtime allows extra retrieval but requires stronger policy checks. |

The eval should check both outcome and trajectory. A final answer that looks good but exceeded budget without approval is not a pass.

## Failure Modes

- Budgets live only in prompts, so the model can ignore or reinterpret them.
- Budgets are checked only after expensive work has already happened.
- The same budget is used for low-risk classification and high-risk automation.



- Cost is optimized while human review work increases.
- Retry budgets hide repeated failure instead of changing strategy.
- Delegation budgets are missing, so multi-agent systems create coordination cost.
- Budget exhaustion is logged as a generic error with no stop reason.
- Operators cannot compare budget policy versions during incident review.

## Production Checklist

- Does every run start with a task class, risk class, and budget policy version?
- Are budgets checked before model, tool, retrieval, memory, and delegation actions?
- Are budget decisions recorded as trace events?
- Are degraded modes defined before launch?
- Can high-risk runs request approval for more budget?
- Can low-risk runs be stopped without human review?
- Are budgets tied to evals, incidents, and release gates?
- Can budget policy changes be rolled back?
- Can operators answer where the cost went for a single run?

## Related Chapters

- [Resource-Aware Agent Design](#)
- [Circuit Breakers, Fallbacks, and Replay](#)
- [Agent Loop](#)
- [Context Budgets and Working Sets](#)
- [Observability and Evals](#)
- [Production Evaluation Feedback Loops](#)
- [Durable Workflows](#)

# Policy Enforcement

Policy enforcement constrains what the agent may say or do through permissions, data-access rules, business rules, safety rules, and escalation.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

Policy enforcement is the software-owned boundary that decides whether a model-proposed answer, tool call, data access, memory write, or side effect is allowed. The model can propose an action. The runtime decides whether to allow, deny, require approval, escalate, or audit it.

Policy should not live only in prompt text. Prompts can explain policy to the model, but enforcement belongs in code, workflow, tool manifests, access-control systems, and auditable decision records.

Knowledge-bound agents use the same idea for answers: the model should answer only from approved sources, cite the evidence, and refuse or escalate when the required source is missing, stale, forbidden, or conflicting.

The practical rule is: policy runs before authority. Before retrieval, before memory writes, before tool execution, before external communication, and before final answers in regulated or evidence-bound domains, the runtime should know whether the action is allowed.

## Use When

- Actions must be checked before execution.
- The agent handles private, regulated, security-sensitive, or business-critical data.
- The system needs approved sources, citations, or compliance constraints.
- Policy decisions must be auditable and replayable.
- The runtime can identify actor, resource, action, capability, risk, and context.
- Tool calls, memory writes, retrieval, final answers, or workflow transitions require different rules by task class.
- A human approval path exists for actions that are valid but too risky to execute autonomously.

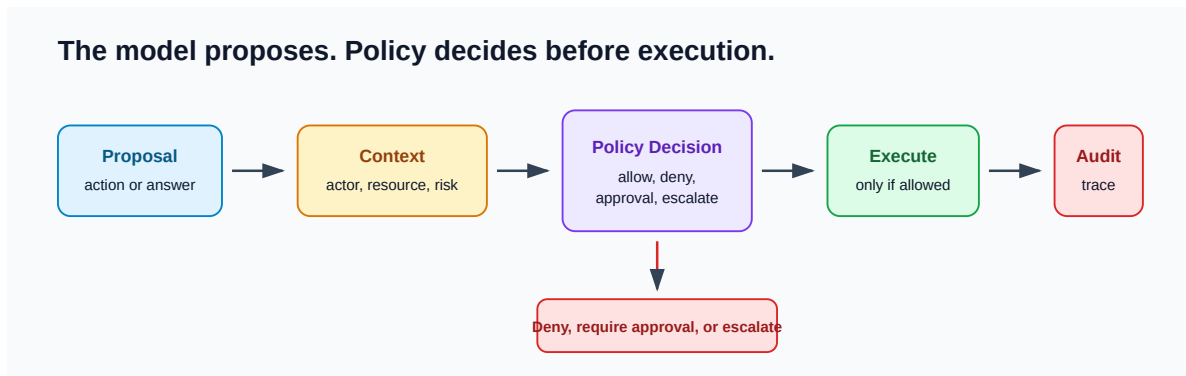
## Avoid When

- Policy is only written as prompt guidance with no runtime check.
- The system cannot identify the actor, resource, action, and context.

- Policy checks happen after irreversible actions.
- Exceptions are silent, unreviewed, or missing from traces.
- Approved knowledge sources cannot be identified, updated, or cited.
- The runtime cannot stop, pause, or change execution after a policy decision.

## Architecture

Use this diagram to read Policy Enforcement as a system boundary, not only a code shape. The key ownership question is: the runtime owns durable state, retries, traces, triggers, deployment configuration, and operational controls.



## System Shape

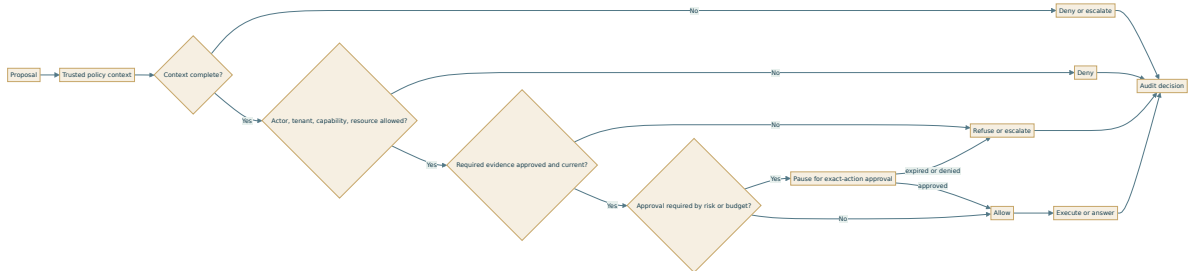
- **Pattern boundary:** the policy boundary evaluates proposed actions, data access, answers, and memory writes before they take effect.
- **State owner:** the runtime owns policy context, decision records, policy version, trace ID, and enforcement outcome.
- **Model role:** the model proposes an action or answer and may explain risk, but it does not grant itself permission.
- **Knowledge boundary:** approved sources, freshness, citations, and refusal rules define what the agent may claim.
- **Budget boundary:** policy can require approval, downgrade capability, or stop when a run has exceeded the spend or autonomy allowed for its risk class.
- **Operational promise:** policy decisions happen before execution and are visible after the run.

## Core Protocol

1. Receive a proposed action, answer, tool call, retrieval result, or memory write.
2. Build policy context: actor, caller, tenant, resource, capability, risk, evidence, and trace ID.
3. Evaluate policy before the action executes or the answer is returned.
4. Return a decision: allow, deny, require approval, escalate, or audit-only.
5. If approval is required, pause through the approval gate.
6. Execute only decisions that are allowed or approved.
7. Record decision, reason, policy version, actor, resource, action, and trace ID.
8. Feed serious denials, misses, and overrides into regression evals.

## Implementation Notes

Use this decision flow when reviewing where policy runs. Every branch should produce a trace event with the policy version, reason, actor, resource, capability, and execution effect.



A policy decision should be a typed runtime object.

```
type PolicyOutcome = 'allow' | 'deny' | 'require_approval' | 'escalate' | 'audit';

type PolicyDecision = {
 actionId: string;
 actor: {
 id: string;
 role: string;
 tenantId?: string;
 };
 resource: {
 type: 'customer_record' | 'refund' | 'email' | 'memory' | 'document';
 id: string;
 tenantId?: string;
 };
 capability: 'read' | 'write' | 'send' | 'refund' | 'remember' | 'answer';
 riskLevel: 'low' | 'medium' | 'high' | 'critical';
 decision: PolicyOutcome;
 reason: string;
 requiredApproval?: {
 approverRole: string;
 approvalPolicy: string;
 };
 policyVersion: string;
 traceId: string;
};
```

The policy context should come from trusted runtime state, not only from model text:

```
type PolicyContext = {
 actionId: string;
```

```

traceId: string;
actorRole: string;
actorTenant?: string;
resourceTenant?: string;
capability: PolicyDecision['capability'];
riskLevel: PolicyDecision['riskLevel'];
toolName?: string;
evidenceStatus?: 'present' | 'missing' | 'stale' | 'forbidden';
budgetState: 'within_budget' | 'approval_threshold' | 'exhausted';
hasHumanApproval: boolean;
policyVersion: string;
};

```

The enforcement function should run before retrieval, memory write, tool call, side effect, or final answer:

```

function enforcePolicy(input: PolicyContext): Pick<PolicyDecision, 'decision' | 'reason'>
{
 if (input.actorTenant && input.resourceTenant && input.actorTenant !==
input.resourceTenant) {
 return { decision: 'deny', reason: 'tenant_boundary' };
 }

 if (input.budgetState === 'exhausted') {
 return { decision: 'escalate', reason: 'budget_exhausted' };
 }

 if (input.budgetState === 'approval_threshold' && !input.hasHumanApproval) {
 return { decision: 'require_approval', reason: 'budget_approval_required' };
 }

 if (input.capability === 'refund' && input.riskLevel === 'high') {
 return { decision: 'require_approval', reason: 'high_risk_refund' };
 }

 if (input.capability === 'send' && input.actorRole !== 'support_agent') {
 return { decision: 'deny', reason: 'role_not_allowed' };
 }

 if (input.capability === 'answer' && input.evidenceStatus !== 'present') {
 return { decision: 'escalate', reason: 'required_evidence_not_available' };
 }

 if (input.capability === 'remember' && input.riskLevel !== 'low') {
 return { decision: 'require_approval', reason: 'memory_write_requires_review' };
 }
}

```

```

 }

 return { decision: 'allow', reason: 'policy_passed' };
 }
}

```

For knowledge-bound answers, policy also decides whether the evidence is allowed:

```

type SourcePolicy = {
 sourceId: string;
 approved: boolean;
 freshness: 'current' | 'stale' | 'unknown';
 citationRequired: boolean;
 allowedTenant?: string;
};

```

The model can explain why an action looks safe. The runtime still makes the decision.

## Where Policy Runs

| Boundary            | Policy Question                                                         |
|---------------------|-------------------------------------------------------------------------|
| Retrieval           | Is this actor allowed to read these sources for this task?              |
| Tool call           | Is this tool allowed for the actor, tenant, resource, risk, and budget? |
| Memory write        | Is this memory safe, scoped, useful, and allowed to persist?            |
| Human approval      | Is this action allowed only after review, and who can approve it?       |
| Final answer        | Is the answer supported by approved evidence and safe to return?        |
| Workflow transition | Is the next step valid after the current state and policy decision?     |

Treat each policy decision as a runtime event. It should have a trace ID, policy version, input summary, decision, reason, and execution effect.

## Failure Modes

- Policy exists only in the system prompt.
- Policy runs after a tool has already executed.
- The decision lacks actor, resource, tenant, or capability context.
- A retry bypasses policy because the first attempt was checked.
- Policy versions change but traces do not record which version applied.
- Denials are not logged, so operators cannot see attempted unsafe actions.
- Approval-required actions are treated as allowed.
- Knowledge answers cite unapproved, stale, or inaccessible sources.
- Exceptions become permanent undocumented policy holes.

- Policy checks ignore budget state, so an agent can keep spending after approval should be required.
- Memory writes bypass policy because they are treated as harmless context management.
- Final answers bypass policy even when the domain requires approved evidence.

## Evaluation Strategy

Policy evals should test allowed, denied, approval-required, and escalation paths.

- Test allowed low-risk actions.
- Test denied actions across role, tenant, resource, and capability boundaries.
- Test approval-required actions before side effects.
- Test stale or unapproved sources in knowledge-bound answers.
- Test retries and resumed workflows to ensure policy is applied every time.
- Test missing actor, resource, or tenant context.
- Test policy version changes and audit completeness.
- Test production incidents as replayable policy fixtures.
- Test budget-threshold cases that require approval before more work.
- Test memory-write denials, approvals, and tenant scoping.
- Test final-answer refusal or escalation when required evidence is missing.

A compact policy eval can look like this:

```
{
 "case_id": "cross_tenant_customer_record_read",
 "proposed_action": {
 "actor_tenant": "tenant_a",
 "resource_tenant": "tenant_b",
 "capability": "read",
 "resource_type": "customer_record",
 "budget_state": "within_budget",
 "evidence_status": "present"
 },
 "expected": {
 "decision": "deny",
 "reason": "tenant_boundary",
 "must_not_execute": true,
 "required_trace_fields": ["actor", "resource", "policy_version", "trace_id"]
 }
}
```

Measure policy decision accuracy, false allow rate, false denial rate, approval-routing accuracy, tenant-boundary violations, stale-policy use, denial logging completeness, and recurrence of known policy failures.

For production systems, false allow is usually the most dangerous metric. A false denial may frustrate a user. A false allow may leak data, move money, send the wrong message, or create an incident.

## Production Checklist

- Enforce policy before execution, answer return, memory write, or external communication.
- Build policy context from trusted runtime data, not only model text.
- Return explicit allow, deny, require-approval, escalate, or audit decisions.
- Record actor, resource, capability, reason, policy version, and trace ID.
- Apply policy on retries and resumed workflows.
- Keep policy versions, tool manifests, source rules, and approval rules versioned.
- Treat missing policy context as deny or escalate.
- Apply policy to memory writes and final answers, not only tools.
- Connect policy decisions to runtime budget state.
- Add dashboards for denials, approvals, overrides, and false allows.
- Convert policy misses into regression evals.
- Review exceptions and expire them intentionally.

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

`compliance-policy-enforcer-agent/policy_contract.ts`

[Open full source](#)

```
export type PolicyOutcome = 'allow' | 'deny' | 'require_approval' | 'escalate';

export type PolicyContext = {
 traceId: string;
 actor: {
 id: string;
 role: 'support_agent' | 'finance_reviewer' | 'viewer';
 tenantId: string;
 };
 resource: {
 type: 'customer_record' | 'refund' | 'email' | 'memory' | 'document';
```



```

 id: string;
 tenantId: string;
};

capability: 'read' | 'write' | 'send' | 'refund' | 'remember' | 'answer';
riskLevel: 'low' | 'medium' | 'high' | 'critical';
evidenceStatus: 'present' | 'missing' | 'stale' | 'forbidden';
budgetState: 'within_budget' | 'approval_threshold' | 'exhausted';
hasHumanApproval: boolean;
policyVersion: string;
};

export type PolicyDecision = {
 traceId: string;
 policyVersion: string;
 decision: PolicyOutcome;
 reason: string;
 executionAllowed: boolean;
 requiredApprovalRole?: 'finance_reviewer' | 'security_reviewer' | 'manager';
};

export function enforcePolicy(context: PolicyContext): PolicyDecision {
 const base = {
 traceId: context.traceId,
 policyVersion: context.policyVersion
 };

 if (context.actor.tenantId !== context.resource.tenantId) {
 return {
 ...base,
 decision: 'deny',
 reason: 'tenant_boundary',
 executionAllowed: false
 };
 }

 if (context.evidenceStatus === 'forbidden') {
 return {
 ...base,
 decision: 'deny',
 reason: 'evidence_forbidden',
 executionAllowed: false
 };
 }
}

```

```

if (context.evidenceStatus === 'missing' || context.evidenceStatus === 'stale') {
 return {
 ...base,
 decision: 'escalate',
 reason: 'required_evidence_not_current',
 executionAllowed: false
 };
}

if (context.budgetState === 'exhausted') {
 return {
 ...base,
 decision: 'escalate',
 reason: 'budget_exhausted',
 executionAllowed: false
 };
}

if (context.capability === 'refund' && context.riskLevel !== 'low' &&
!context.hasHumanApproval) {
 return {
 ...base,
 decision: 'require_approval',
 reason: 'refund_requires_review',
 executionAllowed: false,
 requiredApprovalRole: 'finance_reviewer'
 };
}

if (context.capability === 'send' && context.actor.role !== 'support_agent') {
 return {
 ...base,
 decision: 'deny',
 reason: 'role_cannot_send',
 executionAllowed: false
 };
}

```

*Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.*

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `compliance-policy-enforcer-agent/` folder from this repository.

## Related Patterns

- [Human Approval Gates](#)
- [Tool Capability Design](#)
- [Agent Threat Model](#)
- [Semantic Recall and RAG](#)
- [Production Runtime Overview](#)
- [Cost Controls and Runtime Budgets](#)
- [Production Evaluation Feedback Loops](#)
- [Pattern Evaluation Checklist](#)

# Event-Triggered Agents

Event-triggered agents run in response to webhooks, queues, schedules, or domain events.

Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

Event-triggered agents run in response to webhooks, queues, schedules, or domain events.

Use this pattern when the system must react to work that arrives without a live user watching the screen. The event runtime, not the model, owns admission, deduplication, idempotency, retries, dead letters, replay, and storm controls.

## Scenario

A support platform emits `refund.requested` when a customer opens a refund case. The agent should inspect policy, summarize the order history, draft a recommendation, and notify a reviewer. It must not process the same refund twice if the queue redelivers the event. It must not act on a stale or unauthorized event. It must stop safely if the policy source is unavailable.

This is the difference between an event-triggered agent and a chat agent. The agent may run at 03:00 with no user present. The runtime must decide whether to accept, delay, reject, retry, or dead-letter the event before the model proposes anything.

## Use When

- A domain event, webhook, queue message, schedule, or file arrival should start bounded agent work.
- The task can be described as one event type mapped to one task class.
- Duplicate delivery, retries, and late events can be handled safely.
- The event can carry or fetch enough authorized context.
- The system can trace event ID, run ID, idempotency key, policy decision, tool calls, and stop reason.

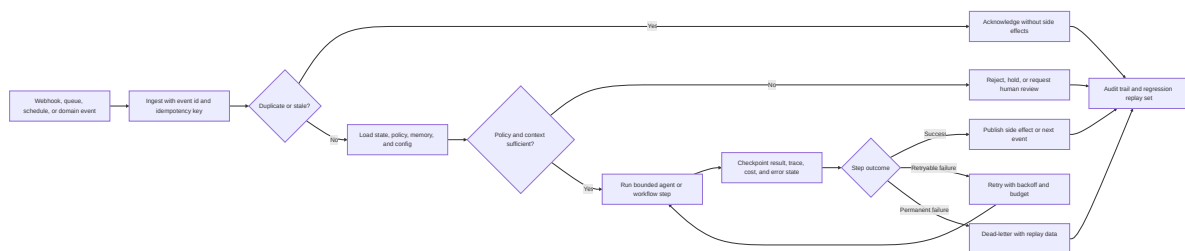
## Avoid When

- The event payload lacks enough identity, tenant, resource, or version data to act safely.
- Duplicate delivery could repeat irreversible side effects.
- Ordering requirements are unclear.

- Failures cannot be retried, compensated, or dead-lettered.
- A human must inspect context before the first action.

## Architecture

Use this diagram to read Event-Triggered Agents as a system boundary, not only a code shape. The key ownership question is: the runtime owns durable state, retries, traces, triggers, deployment configuration, and operational controls.

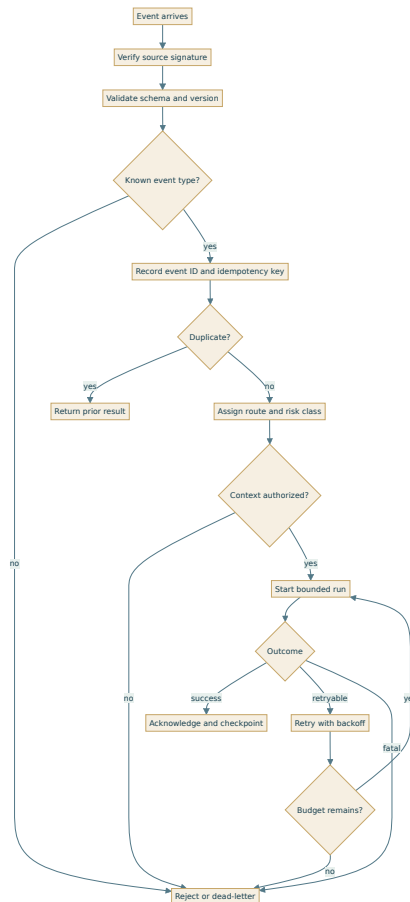


Read it as an unattended runtime boundary: event identity, idempotency, policy, checkpoints, retries, dead letters, and replay evidence protect every side effect.

## Decision Rules

| Question                         | Required Answer                                             | Failure If Missing                               |
|----------------------------------|-------------------------------------------------------------|--------------------------------------------------|
| What event type starts the work? | Stable event name and version.                              | One route handles incompatible payloads.         |
| What resource is affected?       | Resource ID, tenant, actor, and correlation ID.             | Cross-tenant or wrong-resource action.           |
| What makes the event unique?     | Event ID plus source, tenant, or resource scope.            | Duplicate delivery repeats work.                 |
| What ordering matters?           | None, per resource, per tenant, or global.                  | Late events overwrite newer state.               |
| What side effects can happen?    | Read, draft, notify, write, or execute.                     | Retry duplicates irreversible actions.           |
| What is retryable?               | Error classes, max attempts, backoff, and dead-letter rule. | The system either drops work or retries forever. |
| Who reviews failures?            | Dead-letter owner and review cadence.                       | Broken events become invisible backlog.          |

## Event Admission Flow



## System Shape

- **Pattern boundary:** a production service or framework hosts the agent behind durable workflow, policy, observability, and deployment boundaries.
- **State owner:** the runtime owns event receipt, deduplication, run state, retries, dead letters, replay, traces, deployment configuration, and operational controls.
- **Primary artifact:** `event-triggered-agent-pattern/` documents the reviewable event-to-agent boundary and runtime controls.
- **Operational promise:** one accepted event creates at most one authorized, traceable run for the affected resource.

## Contract

An event-triggered agent needs an event envelope before it needs a prompt.

| Field                     | Purpose                                            |
|---------------------------|----------------------------------------------------|
| <code>eventId</code>      | Deduplication and replay identity.                 |
| <code>eventType</code>    | Routes the event to one bounded task class.        |
| <code>eventVersion</code> | Protects incompatible payload changes.             |
| <code>source</code>       | Names webhook, queue, schedule, or service origin. |

| Field                                               | Purpose                                                                |
|-----------------------------------------------------|------------------------------------------------------------------------|
| <code>actorId</code>                                | Records who or what caused the event when known.                       |
| <code>tenantId</code>                               | Enforces tenant boundary before retrieval or tools.                    |
| <code>resourceId</code>                             | Identifies the order, ticket, account, file, or record being affected. |
| <code>occurredAt</code> and <code>receivedAt</code> | Supports late-event and ordering decisions.                            |
| <code>correlationId</code>                          | Connects upstream request, event, run, tools, and traces.              |
| <code>idempotencyKey</code>                         | Prevents duplicate side effects across retries.                        |
| <code>payload</code>                                | Carries validated task data, not trusted instructions.                 |

## Core Protocol

1. Verify event source, schema, version, identity, tenant, and resource scope.
2. Record event ID, correlation ID, and idempotency key before side effects.
3. Route the event to one bounded task class with a risk level and policy context.
4. Fetch required context through authorized tools instead of trusting payload text.
5. Execute one bounded agent or workflow step.
6. Checkpoint result, trace data, cost, and error state.
7. Acknowledge, retry, compensate, continue, refuse, escalate, or dead-letter according to runtime policy.

## Implementation Notes

- Treat event payloads as untrusted input. A webhook body can describe a task, but it should not override policy, route, tools, or memory rules.
- Record the idempotency key before calling a write-capable tool.
- Define ordering per resource. Most systems need per-ticket, per-order, or per-account ordering, not global ordering.
- Use optimistic version checks or locks when two events can affect the same resource.
- Keep retry budgets separate for model calls, retrieval, tools, and whole-event reprocessing.
- Add a dead-letter review loop. A dead letter that no one reviews is silent data loss.
- Make replay version-aware. Replaying an old event against a new policy, prompt, or tool version should be explicit.

## Failure Modes

- Duplicate side effect: redelivery repeats a refund, email, ticket update, or infrastructure change.
- Event storm: one upstream bug floods the agent and burns model/tool budget.
- Poison event: malformed or hostile payload text reaches model context as instruction.
- Late event overwrite: an old event updates state after a newer decision.
- Context gap: the event lacks identity, tenant, resource, or authorization context.

- Retry loop: an unrecoverable error is treated as retryable.
- Dead-letter neglect: failed events pile up without owner, alert, or replay path.
- Replay drift: a replayed event uses a different policy or prompt without recording the version change.

## Review Checklist

Before production, check:

- Event type, version, source, actor, tenant, resource, and correlation ID are recorded.
- Unsupported event versions fail safely.
- Duplicate delivery returns a prior result or resumes the same run.
- Idempotency key is recorded before external writes.
- Ordering and concurrency rules are explicit.
- Retryable and fatal errors are classified.
- Dead-letter queue has an owner, alert, review cadence, and replay procedure.
- Event storm controls can pause, shed, sample, or route to deterministic fallback.
- Trace links event ID, run ID, policy decision, tool calls, and stop reason.

## Evaluation Strategy

- Test valid event, duplicate event, missing identity, unauthorized tenant, unsupported version, late event, and out-of-order event.
- Test retryable tool outage, fatal schema error, dead-letter replay, and model timeout.
- Test event storms with per-tenant and per-route concurrency limits.
- Test prompt injection embedded in payload text.
- Test idempotency by replaying the same event after a partial write.
- Measure accepted rate, duplicate rate, retry rate, dead-letter rate, processing latency, queue depth, model/tool spend, and replay success.

## Production Checklist

- Event contracts are versioned and reviewed before release.
- Webhooks verify signatures; queues verify source and permissions.
- The runtime stores event ID, run ID, idempotency key, route, risk class, and policy version.
- Side effects are idempotent, compensatable, or approval-gated.
- Retry budgets and backoff are documented per error class.
- Dead-letter replay requires owner approval and version awareness.
- Queue depth, retry rate, dead-letter count, latency, and spend have alerts.
- Operators can pause a route, drain a queue, replay one event, and disable write tools.

Use the online book's event-triggered agent review checklist before a production pilot.

## Code Walkthrough



Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

This pattern currently has no dedicated code excerpt. Use the source and download links below for the full pattern folder.

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `event-triggered-agent-pattern/` folder from this repository.

## Related Patterns

- [Durable Workflows](#)
- [Observability and Evals](#)
- [Policy Enforcement](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# Mastra Runtime

Mastra is a TypeScript runtime pattern for applications that need agents, workflows, tools, memory, evals, and observability in one framework.

## Source and downloads

- [Repository source](#)
- [Download code bundle](#)

## Intent

The Mastra Runtime Pattern uses Mastra as a TypeScript runtime for production agent applications. Mastra gives agents, workflows, tools, memory, evals, and observability a shared application structure.

## Use When

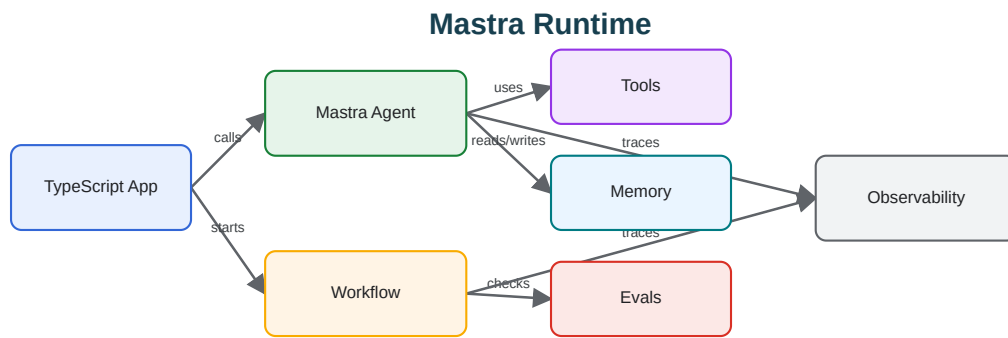
- You are building a TypeScript or Node-based agent product.
- You need agents and deterministic workflows in the same runtime.
- You want memory, tools, evals, and tracing to be first-class concerns.

## Avoid When

- You only need a small script or single model call.
- Your team is committed to a Python-first agent stack.
- You cannot accept framework conventions around project structure and deployment.

## Architecture

Use this diagram to read Mastra Runtime as a system boundary, not only a code shape. The key ownership question is: the runtime owns durable state, retries, traces, triggers, deployment configuration, and operational controls.



## System Shape

- **Application boundary:** the product service owns user identity, tenant scope, request validation, and response delivery.
- **Runtime boundary:** Mastra hosts the agent, workflow, tools, memory, evals, and observability concerns.
- **Workflow boundary:** deterministic state transitions, retries, approval waits, and rollback points belong in workflows, not prompts.
- **Tool boundary:** tools expose typed inputs, typed outputs, side-effect labels, permission requirements, and trace fields.
- **Policy boundary:** product policy runs before tools, memory writes, outbound messages, or external side effects.
- **Portability boundary:** prompts, tool manifests, eval fixtures, trace schema, and policy rules remain readable outside Mastra-specific code.

## Core Protocol

1. Accept a request with actor, tenant, goal, release version, and idempotency key.
2. Load runtime configuration, memory policy, tool registry, and workflow state.
3. Route deterministic steps through the workflow and open-ended decisions through the agent.
4. Check policy before retrieval, memory writes, tool calls, and final answers that require approved evidence.
5. Execute tools through typed wrappers that record status, latency, cost, retry count, and side-effect IDs.
6. Emit trace events for workflow steps, model calls, tool calls, policy decisions, memory access, and eval results.
7. Run post-run evals or CI evals against the trace before promoting the change.
8. Roll back by disabling the risky tool, prompt, model, workflow, or whole agent route.

## Implementation Notes

- Use agents for open-ended decisions where the next step is not known upfront.
- Use workflows for predetermined control flow, state transitions, retries, and production orchestration.
- Keep tools typed and independently testable.
- Capture traces and evals from the beginning rather than adding them after failures.
- Keep provider credentials in environment variables and document them in `.env.example`.
- Keep framework-generated defaults out of product policy. Product policy should be visible in code, tests, ADRs, and eval fixtures.
- Version prompts, tools, policies, memory contracts, eval datasets, and workflow definitions together.
- Treat framework upgrades like runtime changes: run regression evals and inspect traces before promotion.

## Failure Modes

- Treating the framework as the architecture instead of modeling goals, state, and failure modes.
- Putting deterministic workflow logic inside prompts.
- Creating tools with vague descriptions and unvalidated inputs.
- Shipping without eval datasets or trace review.
- Letting memory writes bypass retention, deletion, correction, or consent rules.
- Exporting traces without redaction or without enough fields to replay a failure.
- Hiding rollback inside code deploys instead of feature flags, tool disablement, or policy tightening.

## Evaluation Strategy

- Test the agent path, workflow path, policy denial path, approval path, and tool failure path separately.
- Assert that the trace contains workflow, model, tool, policy, memory, and evaluator events for representative runs.
- Compare prompt, model, tool, and framework changes against the same fixture set before release.
- Include a negative case where the runtime must draft or escalate instead of executing a side effect.

## Production Checklist

- Document install, local run, test, eval, and cleanup commands.
- Commit `.env.example` and keep secret values out of source.
- Define workflow state, memory retention, tool side effects, and policy enforcement points.
- Export redacted traces to the team's observability system.
- Add CI eval gates for prompt, model, tool, policy, memory, and workflow changes.
- Define rollback for model, prompt, tool, workflow, policy, and full agent disablement.

## Run the Example

```
npm run mastra-runtime:demo
npm run mastra-runtime:test
```

## Code Walkthrough

Read the excerpt as the smallest executable expression of the pattern. The surrounding chapter explains the design constraints; the code shows where those constraints become concrete interfaces, state, validation, or control flow.

## Source Code

These excerpts show the implementation shape. The complete code is available in the download bundle and repository source.

`mastra-runtime-pattern/typescript/src/runtime_packaging.ts`

[Open full source](#)

```
export type ToolCall = {
 name: string;
 input: Record<string, unknown>;
};

export type RuntimeTrace = {
 step: string;
 detail: Record<string, unknown>;
};

export type RuntimeState = {
 runId: string;
 goal: string;
 memory: Record<string, string>;
 traces: RuntimeTrace[];
 toolCalls: ToolCall[];
 result?: string;
};

export type Tool = {
 name: string;
 description: string;
 execute(input: Record<string, unknown>, state: RuntimeState): Promise<string>;
};

export type Agent = {
 name: string;
 instructions: string;
 decide(state: RuntimeState): Promise<ToolCall | { answer: string }>;
```

```

};

export type WorkflowStep = {
 name: string;
 run(state: RuntimeState): Promise<RuntimeState>;
};

export type PackagedRuntime = {
 agent: Agent;
 tools: Record<string, Tool>;
 workflow: WorkflowStep[];
 run(goal: string): Promise<RuntimeState>;
};

function trace(state: RuntimeState, step: string, detail: Record<string, unknown>) {
 state.traces.push({ step, detail });
}

export function createSupportRuntime(): PackagedRuntime {
 const tools: Record<string, Tool> = {
 read_policy: {
 name: "read_policy",
 description: "Read the support policy for a refund request.",
 execute: async input => `Policy ${input.policyId}: refunds under 30 days can be
drafted for review.`
 },
 draft_response: {
 name: "draft_response",
 description: "Draft a customer-safe response without sending it.",
 execute: async input => `Draft response for ${input.customerId}: refund request is
ready for review.`
 }
 };

 const agent: Agent = {
 name: "support-runtime-agent",
 instructions: "Check policy before drafting. Do not send messages directly.",
 decide: async state => {
 if (!state.memory.policy) {
 return { name: "read_policy", input: { policyId: "refund-v1" } };
 }
 if (!state.memory.draft) {
 return { name: "draft_response", input: { customerId: "cust_123" } };
 }
 }
 };
}

```

```

 return { answer: "Policy checked and draft created for human review." };
 },
};

const workflow: WorkflowStep[] = [
 {
 name: "agent_decision",
 run: async state => {
 const decision = await agent.decide(state);
 trace(state, "agent_decision", { decision });

 if ("answer" in decision) {
 state.result = decision.answer;
 return state;
 }

 const tool = tools[decision.name];
 if (!tool) throw new Error(`Unknown tool: ${decision.name}`);

```

*Excerpt truncated for readability. Download the bundle or open the source file for the complete implementation.*

[mastra-runtime-pattern/typescript/test/runtime\\_packaging.spec.ts](#)

[Open full source](#)

```

import { createSupportRuntime, evaluateRuntime } from "../src/runtime_packaging.ts";

function assert(condition: unknown, message: string): asserts condition {
 if (!condition) throw new Error(message);
}

const runtime = createSupportRuntime();
const state = await runtime.run("Prepare a policy-safe refund response");
const evaluation = evaluateRuntime(state);

assert(state.result === "Policy checked and draft created for human review.", "Expected final runtime result");
assert(state.toolCalls.map(call => call.name).join(",") === "read_policy,draft_response", "Expected ordered tool calls");
assert(state.memory.policy.includes("refunds under 30 days"), "Expected policy memory");
assert(state.memory.draft.includes("ready for review"), "Expected draft memory");
assert(state.traces.some(event => event.step === "workflow_step"), "Expected workflow trace");

```

```

assert(state.traces.some(event => event.step === "agent_decision"), "Expected agent
decision trace");
assert(evaluation.status === "pass", "Expected runtime evaluation to pass");

const unsafeEvaluation = evaluateRuntime({
 ...state,
 toolCalls: [...state.toolCalls, { name: "refunds.issue_refund", input: { amount: 42 } }],
});

assert(unsafeEvaluation.status === "fail", "Expected forbidden refund tool to fail
evaluation");
assert(
 unsafeEvaluation.reasons.includes("forbidden tool was called: refunds.issue_refund"),
 "Expected forbidden tool reason"
);

console.log("Mastra-style runtime packaging tests OK");

```

## Download

- [Download source bundle](#)
- [Open source folder](#)

The download bundle contains the current `mastra-runtime-pattern/` folder from this repository.

## Related Patterns

- [Durable Workflow](#)
- [Observability and Evals](#)
- [Agent Loop](#)



# Hands-On Labs

The labs turn the reference chapters into a build path. Each lab uses code that already lives in this repository, so you can read the pattern, run the example, change one thing, and connect the result back to production design.

The labs are intentionally framework-agnostic. They move between TypeScript and Python, and across minimal custom runtimes, LangChain/LangGraph-style retrieval, AutoGen-style manager/worker examples, A2A protocol code, MCP-style tool boundaries, and framework-neutral tests. The point is not to teach one API. The point is to show the architecture that survives when the framework changes.

Use [Lab Framework and Language Matrix](#) before starting if you want to see which language, framework, and architectural boundary each lab emphasizes. Use [Lab Production Readiness Checklist](#) and the [lab production readiness worksheet](#) after each lab to identify what the demo still needs before production. Use [From-Scratch Mini-Framework Track](#) when you want to understand what agent frameworks package under the hood. Use [Vertical Slice Examples](#) after the labs, or whenever you want to see several patterns composed into one realistic task. Use [Capstone Projects](#) when you want product-shaped examples with ADRs, traces, evals, runbooks, rollback plans, and native framework slices.

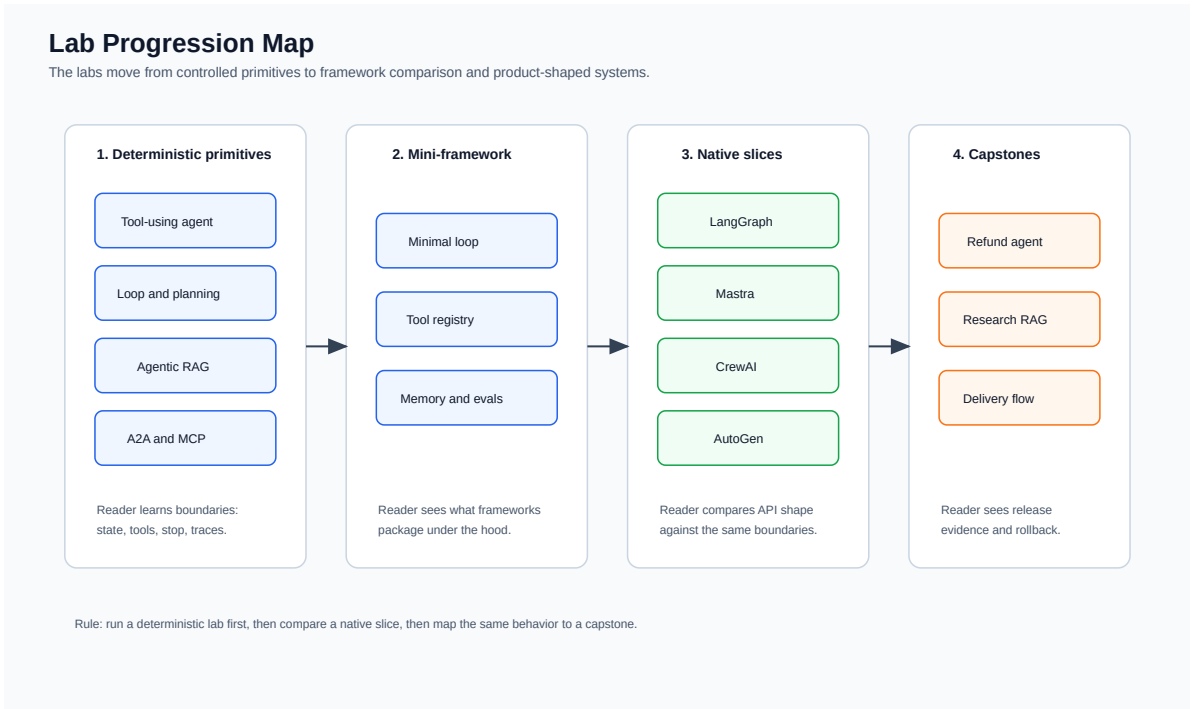
Run these commands from the repository root before starting:

```
npm install
npm test
npm run typecheck
```

Some examples can run with deterministic fallbacks. Examples that call live models require a `.env` file with `MISTRAL_API_KEY`.

## Lab Progression Map

Use this map to understand why the labs are ordered this way. The sequence starts with deterministic primitives, exposes the mini-framework underneath agent runtimes, compares native framework slices, and then moves into capstone-level release evidence.



## Lab Standard

Each lab should leave you with three things: a runnable example, a specific design boundary you can explain, and one production hardening step you know how to make.

Every lab follows the same learning contract:

1. State the objective.
2. Name the language, framework, and source files.
3. Run a baseline command.
4. Inspect the code boundary.
5. Change one thing.
6. Verify the result.
7. Identify what production would need next.

The examples stay small on purpose. A small example is useful only when the lab also says what is intentionally missing: durable state, policy enforcement, stronger schemas, approval, tracing, evals, deployment, or framework integration. When a native framework slice exists, treat it as the next comparison point, not as a replacement for the deterministic lab.

## Planning Table

Use this table to choose a lab by effort and outcome. Time estimates assume you can already run the repository tests. Each lab page also includes optional per-exercise time blocks so you can split the work across shorter sessions.

Download the reusable worksheet: [lab completion worksheet](#).

Download the production follow-up worksheet: [lab production readiness worksheet](#).

For the high-leverage labs, use the focused worksheets for [Lab 02 planning loops](#), [Lab 03 Agentic RAG](#), [Lab 06 observability and evals](#), [Lab 07 runtime packaging](#), and [Lab 12 state graphs](#).

Compare your finished worksheet with the [completed lab evidence examples](#) before treating the lab as review-ready.

Use the [captured lab](#) and [capstone command output examples](#) when you need a concrete model for saved command output, trace snapshots, eval snapshots, and release evidence.

| Lab                                                    | Time       | Level        | Prerequisite                          | Reusable Artifact                                                 |
|--------------------------------------------------------|------------|--------------|---------------------------------------|-------------------------------------------------------------------|
| <a href="#">Lab 01 - Tool-Using Agent</a>              | 20-30 min  | Beginner     | TypeScript basics                     | Tool boundary and error behavior.                                 |
| <a href="#">Lab 02 - Agent Loop and Planning</a>       | 35-55 min  | Beginner     | Lab 01 or equivalent tool boundary    | Plan/execute split with structured stop-condition evidence.       |
| <a href="#">Lab 03 - Agentic RAG</a>                   | 45-75 min  | Intermediate | Retrieval and Python basics           | Evidence-grounded answer path plus missing-evidence eval fixture. |
| <a href="#">Lab 04 - A2A Communication</a>             | 45-60 min  | Intermediate | JSON schema and HTTP/message concepts | Typed agent message envelope.                                     |
| <a href="#">Lab 05 - Multi-Agent Supervisor</a>        | 45-60 min  | Intermediate | Delegation and aggregation concepts   | Supervisor/worker contract.                                       |
| <a href="#">Lab 06 - Observability and Evals</a>       | 50-85 min  | Intermediate | Any earlier lab                       | Trace contract, negative eval, and CI gate sketch.                |
| <a href="#">Lab 07 - Mastra Runtime Packaging</a>      | 70-105 min | Advanced     | TypeScript runtime packaging          | Agent, tool, workflow, memory, eval, and rollback slice.          |
| <a href="#">Lab 08 - CrewAI Flows and Crews</a>        | 60-90 min  | Advanced     | Python and role/task orchestration    | Flow, crew, role, and acceptance contract.                        |
| <a href="#">Mini-Framework Track</a>                   | 2-4 hr     | Advanced     | Labs 01, 02, and 06                   | Runtime primitives you can compare to frameworks.                 |
| <a href="#">Lab 09 - Minimal Agent Loop</a>            | 45-75 min  | Intermediate | Mini-framework setup                  | Loop state, observations, budgets, and stop reasons.              |
| <a href="#">Lab 10 - Tool Registry and Policy Gate</a> | 45-75 min  | Intermediate | Lab 09                                | Tool registry with policy decisions.                              |
| <a href="#">Lab 11 - Context, Memory, Trace, and</a>   | 60-90 min  | Advanced     | Labs 09 and 10                        | Reviewable runtime trace and trajectory eval.                     |

| Lab                               | Time       | Level    | Prerequisite         | Reusable Artifact                                                   |
|-----------------------------------|------------|----------|----------------------|---------------------------------------------------------------------|
| Evals                             |            |          |                      |                                                                     |
| Lab 12 - LangGraph State Graph    | 70-105 min | Advanced | Graph/state concepts | Checkpointed state graph with interrupt, resume, and replay review. |
| Lab 13 - AutoGen Transcript Evals | 60-90 min  | Advanced | Multi-agent basics   | Transcript rubric and regression check.                             |

## Completion Standard

A lab is complete when you can show four things:

1. The baseline command runs.
2. The expected output matches the lab's success signal.
3. One intentional failure path is visible and controlled.
4. You can name the production gap before using the pattern with real users, data, credentials, or side effects.

Do not count a lab as finished just because the happy path works. The value comes from seeing the boundary: what the model can decide, what software must enforce, what gets traced, and what would block production.

## Lab Evidence Pack

Save a small evidence pack after each lab. It turns the lab from a one-time exercise into material you can reuse in a design review, ADR, eval suite, or capstone.

| Evidence         | What To Capture                                                 | Why It Matters                                        |
|------------------|-----------------------------------------------------------------|-------------------------------------------------------|
| Baseline command | Command, exit status, and expected output signal.               | Proves the example ran before you changed it.         |
| Source boundary  | Files inspected and the contract each file owns.                | Shows where the pattern becomes code.                 |
| Small change     | One input, rule, prompt, tool, schema, or policy change.        | Proves you can modify behavior intentionally.         |
| Failure path     | Error, refusal, denial, timeout, budget stop, or invalid input. | Shows the boundary fails visibly instead of silently. |
| Trace or log     | Minimal trace, transcript, or structured output.                | Gives future evals something concrete to assert.      |

| Evidence       | What To Capture                               | Why It Matters                               |
|----------------|-----------------------------------------------|----------------------------------------------|
| Production gap | Missing control and the next artifact needed. | Connects the lab to production architecture. |

Keep the pack short. One screen of evidence is better than a folder of unreviewed screenshots.

Use the [completed lab evidence examples](#) as calibration. A good evidence pack names the command, output, failure path, protected boundary, production gap, and next owner.

Use the [captured command output examples](#) to compare the shape of your saved terminal output. The important signal is not a screenshot. It is a short, reviewable record that shows command, success signal, trace or eval link, and production question.

## Framework-Agnostic Rule

Frameworks change the API, not the architecture questions. For every lab, ask:

- What owns state?
- What can the model decide?
- What can software validate?
- What tools are exposed?
- What policy is enforced outside the prompt?
- What is traced?
- Why does the run stop?

Those questions apply whether the code uses LangGraph, LangChain, Mastra AI, AutoGen-style agents, CrewAI, MCP, A2A, or a small custom runtime.

## End-To-End Reader Path

Use this path when you want to move from learning to implementation:

1. Start with [Lab Framework and Language Matrix](#) and choose the highest-risk boundary.
2. Run the matching deterministic lab.
3. Read the production extension and readiness checklist.
4. Compare the matching native example under [native-framework-examples/](#).
5. Map the same behavior to a capstone.
6. Fill out the framework selection ADR and rollback worksheet.
7. Add evals that fail the build before adding real side-effect tools.

For the support refund path, use Lab 07, [native-framework-examples/mastra-refund/](#), the Support Refund Agent capstone, and the production readiness worksheet.

## Lab Sequence

1. Lab Framework and Language Matrix
2. Lab Production Readiness Checklist
3. Build a Tool-Using Agent
4. Build an Agent Loop with Planning
5. Build Agentic RAG
6. Build A2A Agent Communication
7. Build a Multi-Agent Supervisor
8. Add Observability and Evals
9. Package Agents, Tools, Workflows, Memory, and Evals
10. Model Flows, Crews, Roles, and Task Contracts
11. Study the From-Scratch Mini-Framework Track
12. Build a Minimal Agent Loop
13. Build a Tool Registry and Policy Gate
14. Add Context, Memory, Trace, and Evals
15. Model State Graphs, Checkpoints, and Interrupts
16. Evaluate Multi-Agent Transcripts
17. Study Vertical Slice Examples

## How To Use These Labs

Read the objective first, then run the command exactly as shown. After that, inspect the named source files and make the small change in the lab. The goal is to see where the pattern becomes code: input contracts, state, tool boundaries, stop conditions, evaluation, and failure handling.

Each lab ends with a production extension. Treat that section as the bridge between a working demo and an architecture decision.

## Expected Output Map

Use this table as the quick success check before you move to the production extension.

| Lab    | Expected Output Signal                                                                                                      |
|--------|-----------------------------------------------------------------------------------------------------------------------------|
| Lab 01 | Structured <code>read_order</code> and <code>search_refund_policy</code> results with trust labels and evidence references. |
| Lab 02 | <code>Planning test OK</code> plus a deterministic plan and result for the CLI path.                                        |
| Lab 03 | Answer reflects retrieved evidence from the local document set.                                                             |
| Lab 04 | A2A test shows success, refusal, invalid-input error, and cancellation.                                                     |

| Lab    | Expected Output Signal                                                                                               |
|--------|----------------------------------------------------------------------------------------------------------------------|
| Lab 05 | Manager delegates bounded work and final aggregation uses worker outputs.                                            |
| Lab 06 | Eval records expose success and negative cases, not only final text.                                                 |
| Lab 07 | <code>Mastra-style runtime packaging tests OK</code> ; native slice exposes agent, tools, workflow, and eval gate.   |
| Lab 08 | <code>CrewAI-style flow and crew tests OK</code> ; Flow accepts only validated role outputs.                         |
| Lab 09 | Immediate answer stops with <code>success</code> ; repeated tool proposals stop with <code>budget_exhausted</code> . |
| Lab 10 | Unknown tools are refused, write tools require approval, and allowed tools record observations.                      |
| Lab 11 | Trace contains context, decision, tool/policy, and stop events; unsafe trajectory eval fails.                        |
| Lab 12 | <code>LangGraph-style state graph tests OK</code> ; resume preserves checkpointed state.                             |
| Lab 13 | <code>AutoGen-style transcript tests OK</code> ; transcript proves role order, stop reason, and final owner.         |

## Recommended Order

Do the labs in order if you are new to agent systems. The sequence starts with one agent and one tool, then adds planning, retrieval, remote agent communication, multi-agent coordination, and production-quality evaluation.

If you already know the basics, start with the lab closest to your current system and use the related chapters as reference material.

After the labs, read the vertical slices to see how the same patterns compose into support, coding, and research workflows. Then read the [Capstone Projects](#) to see production-shaped systems with framework mappings, native slices, and release evidence.

If you are evaluating frameworks, do the mini-framework track before choosing a production runtime. Building the primitives once makes it easier to see which responsibilities a framework owns and which responsibilities remain in your application.

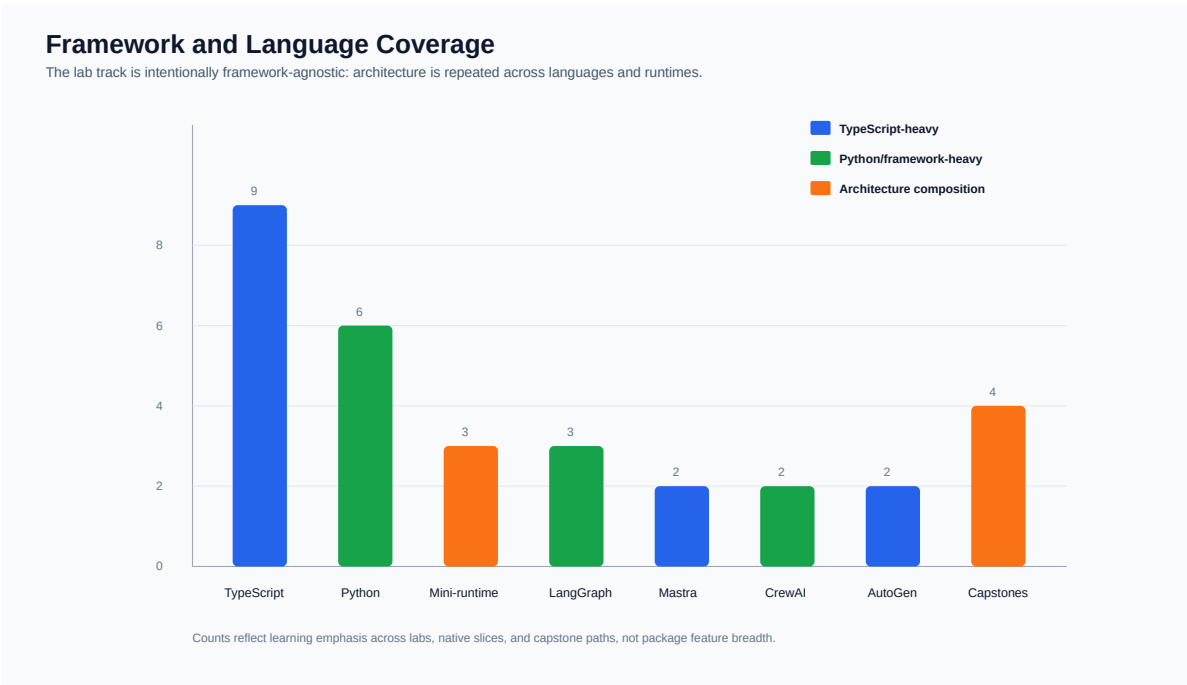
# Lab Framework and Language Matrix

The labs are intentionally language- and framework-agnostic. They use different tools so you can see the architectural pattern beneath the framework API.

Use this page before you start coding. It answers two questions: which lab should I run first, and what production boundary should I inspect while I run it?

## Coverage Graph

Use this graph to see the learning emphasis across languages, runtimes, framework slices, and capstones. The goal is balanced architecture exposure, not equal package coverage.



| Lab                       | Pattern  | Language   | Framework / Runtime                            | Framework-Agnostic Lesson                                                    |
|---------------------------|----------|------------|------------------------------------------------|------------------------------------------------------------------------------|
| Lab 01 - Tool-Using Agent | Tool use | TypeScript | Minimal custom runtime / AutoGen-style example | The model proposes a capability use; software owns validation and execution. |



| Lab                               | Pattern                     | Language                       | Framework / Runtime                                      | Framework-Agnostic Lesson                                                                       |
|-----------------------------------|-----------------------------|--------------------------------|----------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| Lab 02 - Agent Loop and Planning  | Planning and execution      | TypeScript, with Python mirror | Framework-neutral planner/executor                       | Planning and execution are separate responsibilities even when one framework packages both.     |
| Lab 03 - Agentic RAG              | Retrieval and grounding     | Python                         | LangChain/LangGraph-style retrieval stack                | Retrieval produces scoped evidence; generation must stay grounded in that evidence.             |
| Lab 04 - A2A Communication        | Agent-to-agent protocol     | TypeScript                     | Protocol-first runtime with Ajv schema validation        | Agent communication needs typed envelopes, correlation IDs, refusals, errors, and cancellation. |
| Lab 05 - Multi-Agent Supervisor   | Supervisor / worker         | TypeScript                     | AutoGen-style manager/worker example                     | A supervisor owns decomposition, worker contracts, and final synthesis.                         |
| Lab 06 - Observability and Evals  | Trace and eval harness      | TypeScript                     | Framework-neutral tests over examples                    | Evals should inspect trajectories, not only final answers.                                      |
| Lab 07 - Mastra Runtime Packaging | Runtime packaging           | TypeScript                     | Mastra-style agents, tools, workflows, memory, and evals | Framework runtime packaging does not remove product ownership of state, policy, and acceptance. |
| Lab 08 - CrewAI Flows and Crews   | Flow and crew orchestration | Python                         | CrewAI-style flows, crews, roles, and tasks              | Flows own state and acceptance; crews perform bounded specialist work.                          |
| Mini-Framework Track              | Runtime primitives          | TypeScript or Python           | From-scratch educational runtime                         | Building the primitives once                                                                    |

| Lab                                                        | Pattern                           | Language             | Framework / Runtime                                                    | Framework-Agnostic Lesson                                                                                  |
|------------------------------------------------------------|-----------------------------------|----------------------|------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
|                                                            |                                   |                      |                                                                        | clarifies what frameworks package.                                                                         |
| <a href="#">Lab 09 - Minimal Agent Loop</a>                | Agent loop                        | TypeScript or Python | From-scratch educational runtime                                       | State, decisions, observations, budgets, and stop reasons are the core loop.                               |
| <a href="#">Lab 10 - Tool Registry and Policy Gate</a>     | Tool and policy boundary          | TypeScript or Python | From-scratch educational runtime                                       | Tool availability and policy authorization are different runtime decisions.                                |
| <a href="#">Lab 11 - Context, Memory, Trace, and Evals</a> | Runtime observability             | TypeScript or Python | From-scratch educational runtime                                       | Context, memory, traces, and trajectory evals make the runtime reviewable.                                 |
| <a href="#">Lab 12 - LangGraph State Graph</a>             | State graph and resume            | Python               | LangGraph-style graph state, nodes, edges, checkpoints, and interrupts | Graph execution is strongest when state, branching, pause/resume, and node observability matter.           |
| <a href="#">Lab 13 - AutoGen Transcript Evals</a>          | Multi-agent transcript evaluation | TypeScript           | AutoGen-style agents, teams, messages, and transcript evals            | A multi-agent run needs a reviewable transcript, explicit stop reason, and role-level acceptance criteria. |

## How To Read The Matrix

Do not treat the framework column as the point of the lab. Treat it as the implementation surface. The durable lesson is the boundary: state, tools, policy, context, communication, evaluation, or runtime control.

If you later use LangGraph, Mastra AI, AutoGen, CrewAI, Semantic Kernel, MCP, or a custom runtime, keep the same questions in view:

- What does the framework own?
- What does your application still own?
- Where is state persisted?

- Where are tool calls validated?
- Where is policy enforced?
- What can be replayed after a failure?

## Choose By Situation

Use this table when you already know the problem you are trying to solve.

| If Your Current Problem Is...              | Start With                           | Then Read                                        | Why                                                                                       |
|--------------------------------------------|--------------------------------------|--------------------------------------------------|-------------------------------------------------------------------------------------------|
| Exposing one tool to a model               | <a href="#">Lab 01</a>               | <a href="#">Tool Capability Design</a>           | You need typed inputs, controlled errors, and permission boundaries before autonomy.      |
| Adding planning or step-by-step execution  | <a href="#">Lab 02</a>               | <a href="#">Planning and Execution</a>           | You need to separate plan creation, execution, and stop reasons.                          |
| Grounding answers in documents             | <a href="#">Lab 03</a>               | <a href="#">Semantic Recall and RAG</a>          | You need source eligibility, retrieval quality, citations, and missing-evidence behavior. |
| Connecting agents or services              | <a href="#">Lab 04</a>               | <a href="#">A2A Agent Interoperability</a>       | You need typed envelopes, identity, refusals, errors, cancellation, and replay.           |
| Splitting work across roles                | <a href="#">Lab 05</a>               | <a href="#">Supervisor / Worker</a>              | You need worker contracts, merge policy, and one final owner.                             |
| Proving behavior with traces and evals     | <a href="#">Lab 06</a>               | <a href="#">Observability and Evals</a>          | You need trajectory evidence, not just final answers.                                     |
| Comparing framework runtime packaging      | <a href="#">Lab 07</a>               | <a href="#">Framework Selection</a>              | You need to see what a runtime packages and what your product still owns.                 |
| Modeling crew-style orchestration          | <a href="#">Lab 08</a>               | <a href="#">Choosing Multi-Agent Topology</a>    | You need to distinguish flow state from role-based worker output.                         |
| Understanding what frameworks package      | <a href="#">Mini-Framework Track</a> | <a href="#">Building a Minimal Agent Runtime</a> | You need the primitives before comparing framework abstractions.                          |
| Testing resume, interrupts, or graph state | <a href="#">Lab 12</a>               | <a href="#">Durable Workflows</a>                | You need checkpoints, state transitions, and failure recovery.                            |
| Evaluating multi-agent conversations       | <a href="#">Lab 13</a>               | <a href="#">Debate and Consensus</a>             | You need transcript evidence, role acceptance, and stop conditions.                       |

If several rows apply, start with the riskiest boundary. A write-capable tool, private data source, long-running workflow, or multi-agent handoff should drive the lab choice.

## Fast Decision Shortcut

Use this shortcut when you have five minutes and need a starting path.

| Highest Risk In Your System    | Run First            | Add Next                  | Production Check                                                                        |
|--------------------------------|----------------------|---------------------------|-----------------------------------------------------------------------------------------|
| A model can call a tool        | Lab 01               | Lab 10                    | Tool schema, permission, idempotency, timeout, and audit record.                        |
| A model can plan several steps | Lab 02               | Lab 09                    | Stop reason, budget, retry rule, cancellation, and trace events.                        |
| A model answers from documents | Lab 03               | Lab 11                    | Source ACLs, freshness, citations, missing-evidence behavior, and retrieval evals.      |
| Agents exchange messages       | Lab 04               | Lab 13                    | Typed envelope, identity, correlation ID, refusal, cancellation, and transcript replay. |
| Work splits across roles       | Lab 05               | Lab 08                    | Worker contract, merge policy, final owner, role permissions, and acceptance eval.      |
| Release quality is unclear     | Lab 06               | Lab 11                    | Trace contract, negative cases, trajectory evals, and blocking thresholds.              |
| Work can pause and resume      | Lab 12               | Durable Workflows chapter | Checkpoint store, interrupt payload, state migration, and stuck-run dashboard.          |
| Framework choice is unclear    | Mini-Framework Track | Lab 07                    | Ownership split between framework runtime and product policy.                           |

Do not optimize for the most interesting lab. Optimize for the boundary most likely to hurt users, leak data, spend money, or block operators.

## Lab-To-Product Paths

Use these paths when you want a lab to become design-review material.

| Product Slice                 | Run                    | Compare With                                                                                                       | Finish With                            |
|-------------------------------|------------------------|--------------------------------------------------------------------------------------------------------------------|----------------------------------------|
| Support refund agent          | Lab 01, Lab 07, Lab 12 | <code>native-framework-examples/mastra-refund/</code> , <code>native-framework-examples/langgraph-refund/</code>   | Support Refund Agent Capstone          |
| Research RAG assistant        | Lab 03, Lab 06, Lab 11 | <code>native-framework-examples/langgraph-research-rag/</code>                                                     | Research RAG Agent Capstone            |
| Multi-agent delivery workflow | Lab 05, Lab 08, Lab 13 | <code>native-framework-examples/autogen-delivery/</code> , <code>native-framework-examples/crewai-delivery/</code> | Multi-Agent Delivery Workflow Capstone |
| Custom runtime foundation     | Labs 09, 10, 11        | Mini-framework TypeScript or Python implementation                                                                 | Reference Architecture                 |

Each path should end with a worksheet, not just a passing command. Use the [lab completion worksheet](#), then the [lab production readiness worksheet](#).

## Current Coverage

The current labs now cover TypeScript, Python, protocol boundaries, framework-neutral tests, LangGraph-style state graphs, AutoGen-style transcripts, Mastra-style runtime packaging, CrewAI-style flow orchestration, from-scratch runtime primitives, production readiness gates, and isolated native framework examples.

Repository native examples:

| Example                                                        | Framework | Connects To                                       |
|----------------------------------------------------------------|-----------|---------------------------------------------------|
| <code>native-framework-examples/langgraph-refund/</code>       | LangGraph | Lab 12 and Support Refund Agent capstone          |
| <code>native-framework-examples/langgraph-research-rag/</code> | LangGraph | Lab 03 and Research RAG Agent capstone            |
| <code>native-framework-examples/mastra-refund/</code>          | Mastra    | Lab 07 and Support Refund Agent capstone          |
| <code>native-framework-examples/autogen-delivery/</code>       | AutoGen   | Lab 13 and Multi-Agent Delivery Workflow capstone |
| <code>native-framework-examples/crewai-delivery/</code>        | CrewAI    | Lab 08 and Multi-Agent Delivery Workflow capstone |

Planned lab expansion should add:

- deeper deployment walkthroughs that connect the readiness checklist to concrete cloud/runtime targets.

## Matrix Review Gate

Before adding a new lab, update this page only if the lab changes reader choice. A new lab should add at least one of these:

- a new architecture boundary;
- a missing language or runtime comparison;
- a production concern not covered by existing labs;
- a capstone path that readers can follow end to end;
- a native framework slice that clarifies what the framework owns.

Do not add labs only to cover another library. Add them when they teach a boundary that engineers need to ship safer systems.

# Lab Production Readiness Checklist

The labs are teaching implementations. This checklist defines what must be added before a lab pattern becomes production work.

Use this after completing a lab. The goal is to identify the next engineering boundary: persistence, authorization, retries, idempotency, observability, eval gates, deployment, and rollback. For the full production path, continue with [Deployment Walkthrough](#), [Templates and Worksheets](#), and the [10/10 Production Gate](#).

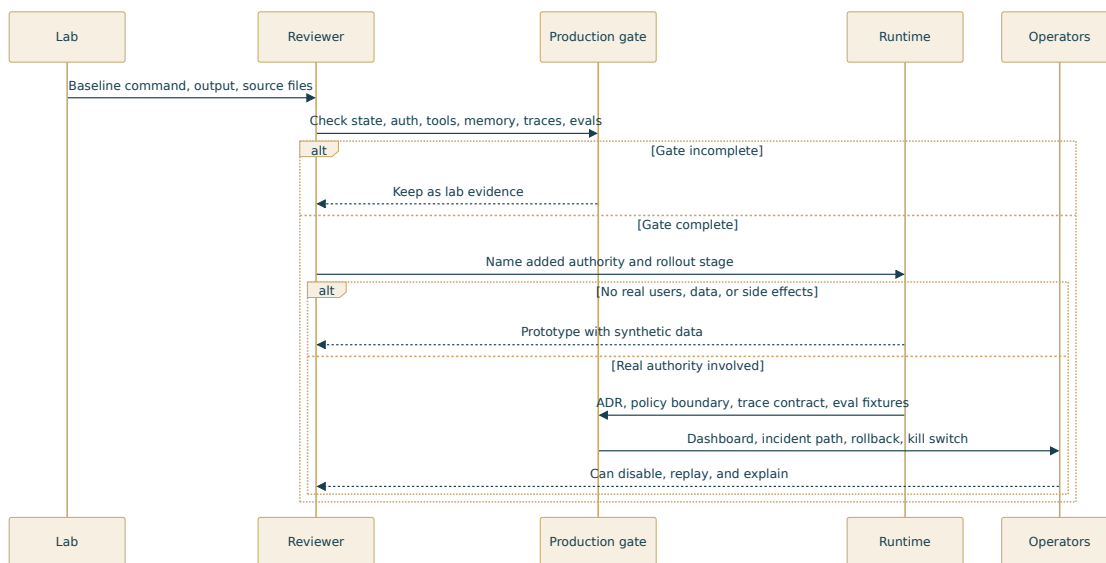
Download the reusable production review artifact: [10/10 production gate scorecard](#).

Download the lab-specific worksheet: [lab production readiness worksheet](#).

## Universal Production Gate

Every lab needs these controls before real users, real data, or real side effects:

| Gate                 | Required Evidence                                                                                     |
|----------------------|-------------------------------------------------------------------------------------------------------|
| State ownership      | State schema, owner, persistence strategy, migration plan, and replay story.                          |
| Auth and permissions | Actor identity, tenant/resource scope, tool permissions, approval rules, and audit records.           |
| Tool safety          | Input schemas, side-effect class, idempotency key, timeout, retry policy, and error contract.         |
| Memory safety        | Retention, deletion, redaction, consent, correction path, and write policy.                           |
| Observability        | Trace IDs, model/tool events, policy decisions, costs, latency, stop reasons, and redaction.          |
| Eval gates           | Golden tasks, negative cases, trajectory checks, safety checks, and release-blocking thresholds.      |
| Deployment           | Runtime owner, environment config, secrets, scaling limits, rollback, kill switch, and incident path. |
| Human control        | Approval UI/API, escalation rules, cancellation, reviewer identity, and expiry.                       |



Use this flow before changing access, data, or authority. A lab can move forward only when the next stage has reviewable evidence, not because the example command ran once.

## Lab Review Questions

Use these questions before a lab-derived system becomes a product slice:

| Question                              | Required Evidence                                                                            |
|---------------------------------------|----------------------------------------------------------------------------------------------|
| What did the lab prove?               | Baseline command, expected output, source files, and success signal.                         |
| What did the lab not prove?           | Missing persistence, policy, observability, evals, deployment, rollback, or human control.   |
| What authority will production add?   | Read data, write tools, memory, messages, approvals, money movement, or workflow execution.  |
| What must fail closed?                | Missing config, missing policy context, unsafe tool input, stale evidence, and failed evals. |
| What is the next production artifact? | ADR, trace contract, eval fixture, runbook, checklist, dashboard, or rollback plan.          |

The lab is a learning artifact until these answers exist in reviewable engineering documents.

## Promotion Ladder

Use this ladder to decide what the lab output is ready for.

| Stage        | Allowed Use               | Required Evidence                                         | Stop Condition                                           |
|--------------|---------------------------|-----------------------------------------------------------|----------------------------------------------------------|
| Lab evidence | Learning, comparison, and | Baseline command, success signal, inspected source files, | Do not connect real users, private data, credentials, or |



| Stage                | Allowed Use                                     | Required Evidence                                                                              | Stop Condition                                                                           |
|----------------------|-------------------------------------------------|------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
|                      | design discussion.                              | and one known failure path.                                                                    | side effects.                                                                            |
| Prototype            | Internal demo with synthetic data.              | Lab evidence plus basic schemas, deterministic test, and named production gaps.                | Stop if the demo requires real credentials or broad tool access.                         |
| Product slice        | Limited internal workflow with controlled data. | ADR, policy boundary, trace contract, eval fixtures, owner, and rollback note.                 | Stop if policy, trace, or rollback is missing.                                           |
| Pilot                | Small real-user or real-data rollout.           | Production gate scorecard, approval rules, incident path, dashboard, and release gate.         | Stop or roll back on failed blocking evals, missing trace spans, or unsafe side effects. |
| Production candidate | Controlled production release.                  | Deployment walkthrough evidence, runbook, kill switch, rollback path, and current eval report. | Stop if operators cannot disable, replay, or explain the run.                            |

Do not skip stages because a framework example runs successfully. A passing command proves execution; it does not prove authority, observability, recovery, or release safety.

## Per-Lab Readiness Matrix

| Lab                               | Production Additions                                                                                                 |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------|
| Lab 01 - Tool-Using Agent         | Tool schemas, side-effect labels, permission checks, idempotency, timeout/retry, and audit records.                  |
| Lab 02 - Agent Loop and Planning  | Durable state, plan versioning, step retry policy, cancellation, partial failure handling, and plan evals.           |
| Lab 03 - Agentic RAG              | Source ACLs, freshness, citation checks, retrieval evals, prompt-injection filtering, and evidence retention rules.  |
| Lab 04 - A2A Communication        | Agent identity, signed envelopes, correlation IDs, cancellation semantics, schema versioning, and replay logs.       |
| Lab 05 - Multi-Agent Supervisor   | Worker contracts, merge policy, per-worker traces, disagreement handling, final acceptance owner, and cost caps.     |
| Lab 06 - Observability and Evals  | Trace storage, redaction, eval datasets, release thresholds, incident-to-eval workflow, and dashboard ownership.     |
| Lab 07 - Mastra Runtime Packaging | Deployment config, tool/memory policy, workflow retries, eval integration, trace export, and framework upgrade plan. |

| Lab                                        | Production Additions                                                                                                     |
|--------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| Lab 08 - CrewAI Flows and Crews            | Flow checkpoints, role permissions, task schemas, crew output validators, human escalation, and flow acceptance evals.   |
| Lab 09 - Minimal Agent Loop                | Decision validation, tool registry, stop policy, state persistence, timeout/cancellation, and trace events.              |
| Lab 10 - Tool Registry and Policy Gate     | Policy context, approval records, schema validation, side-effect isolation, idempotency, and denial analytics.           |
| Lab 11 - Context, Memory, Trace, and Evals | Memory governance, trace redaction, eval fixture versioning, context audit, and incident replay.                         |
| Lab 12 - LangGraph State Graph             | Durable checkpointer, thread IDs, interrupt payloads, node idempotency, state migrations, and resume tests.              |
| Lab 13 - AutoGen Transcript Evals          | Message schemas, team termination, transcript redaction, role permissions, transcript replay, and team-level eval gates. |

### Framework-Specific Deployment Questions

| Framework Shape | Production Questions                                                                                                                    |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| LangGraph-style | Where is the checkpointer stored? How are thread IDs assigned? Which nodes can cause side effects? What state migrations are supported? |
| AutoGen-style   | Who owns the transcript? Which messages are durable? How does termination work? How are agent tools permissioned?                       |
| Mastra-style    | Which runtime features are framework-owned? How are workflows deployed? How are tools, memory, traces, and evals exported?              |
| CrewAI-style    | What does the flow own? What does the crew own? How are role outputs validated? How are crew failures escalated?                        |
| Mini-runtime    | Which production controls are you willing to build and operate yourself? Which should move into an existing workflow platform?          |

### Release Gate Template

Before shipping a lab-derived system, require:

```
state: persisted or explicitly stateless
policy: enforced before side effects
tools: typed, scoped, idempotent, timed out
memory: governed by retention/deletion rules
```

```
trace: redacted and replayable
evals: include happy path, negative path, and trajectory checks
deployment: rollback and kill switch documented
owner: team and escalation path assigned
```

If any line is unknown, the system is still a demo.

## Related Chapters

- [Cross-Framework Decision Matrix](#)
- [Framework Selection](#)
- [Real Framework Setup Notes](#)
- [Templates and Worksheets](#)
- [10/10 Production Gate](#)
- [Production Runtime Overview](#)
- [Deployment Walkthrough](#)
- [Policy Enforcement](#)
- [Observability and Evals](#)

# Lab 01 - Build a Tool-Using Agent

Download the [lab completion worksheet](#) and [lab production readiness worksheet](#) before you start.

## Objective

Build the smallest useful tool boundary: a runtime receives a proposed tool call, validates the route and arguments, executes the tool behind code, marks trusted and untrusted outputs, and returns a structured result.

## What You Will Use

- Language: TypeScript
- Framework/runtime: minimal custom runtime with an AutoGen-style example path
- Framework-agnostic lesson: the model may propose tool use, but software owns validation, execution, and error handling.
- Pattern chapter: [Tool Use](#)
- Source folder: `tool-using-agent-pattern/`
- Download: [tool-use.zip](#)
- Main file: `tool-using-agent-pattern/typescript/src/tool_runtime.ts`
- Demo file: `tool-using-agent-pattern/typescript/src/run_demo.ts`

## Exercise Time Budget

These estimates assume dependencies are already installed.

| Exercise                      | Time     | Output                                                              |
|-------------------------------|----------|---------------------------------------------------------------------|
| Setup and baseline run        | 5 min    | Passing tool-runtime command output.                                |
| Inspect the tool boundary     | 8 min    | Notes on route validation, trusted data, and untrusted policy text. |
| Change one route or tool case | 8-10 min | A visible success, denial, or controlled error.                     |
| Review the production gap     | 5-7 min  | One missing control to carry into the readiness worksheet.          |

## Setup

From the repository root:

```
npm install
```

This lab runs without a model key. It exercises the tool runtime directly so you can inspect the software boundary before adding model proposals.

## Run It

```
npm run tool-using-agent
```

Expected output:

```
{
 "order": {
 "status": "ok",
 "tool": "read_order",
 "trust": "trusted_system"
 },
 "policy": {
 "status": "ok",
 "tool": "search_refund_policy",
 "trust": "untrusted_content"
 }
}
```

The actual output also includes tool data and `evidenceRef` values. Record those fields in the lab completion worksheet.

## Inspect The Code

Open `tool-using-agent-pattern/typescript/src/tool_runtime.ts` and find:

- `ToolName`: the exposed tool names.
- `ToolContext`: the runtime context for route, approvals, timeouts, and attempts.
- `disclosedTools(route)`: the route-based tool disclosure boundary.
- `validateProposal(proposal, context)`: the validation boundary.
- `execute(proposal, context)`: the execution boundary.

Then open `tool-using-agent-pattern/typescript/src/run_demo.ts` and find the two demo proposals: `read_order` and `search_refund_policy`.

The important design point is that a model may propose a tool call, but code owns disclosure, validation, execution, trust marking, and evidence references.

## Change One Thing

Change the route in `run_demo.ts` from:

```
route: "refund_investigation",
```

to:

```
route: "order_status",
```

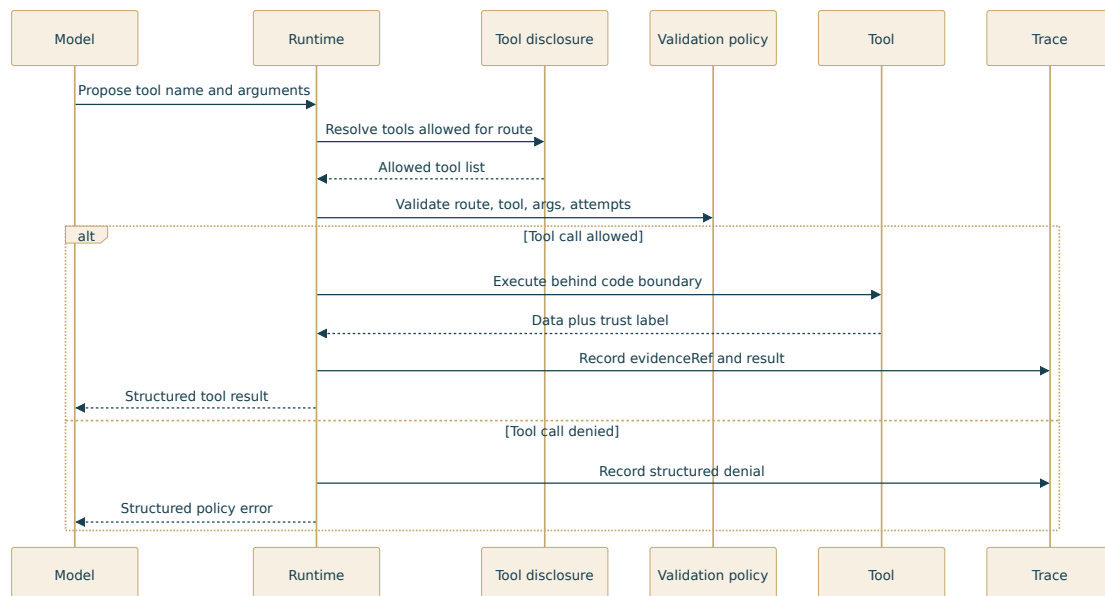
Run the lab again.

## Expected Result

The `read_order` call should still succeed because `order_status` may use that tool. The `search_refund_policy` call should be denied because that route exposes only `read_order`.

Record the denial as the intentional failure path. It proves that tool availability depends on the route, not on what the model asks for.

Use this flow as the acceptance model for the lab. The model proposes a call, but the runtime owns disclosure, validation, execution, trust marking, and trace evidence.



## Lab Review Gate

Before moving on, verify the boundary instead of only checking the happy path:

| Check                        | Evidence                                                                                                                          |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| Tool disclosure is narrow    | <code>order_status</code> exposes <code>read_order</code> ; <code>refund_investigation</code> exposes refund-investigation tools. |
| Tool execution is code-owned | <code>ToolRuntime</code> executes <code>read_order</code> and <code>search_refund_policy</code> ; model text does not.            |

| Check                                | Evidence                                                                                                      |
|--------------------------------------|---------------------------------------------------------------------------------------------------------------|
| Invalid route/tool use is controlled | A hidden or disallowed tool returns a structured denial.                                                      |
| Trust is explicit                    | Order data is <code>trusted_system</code> ; policy text is <code>untrusted_content</code> .                   |
| Evidence is traceable                | Successful tool results include <code>evidenceRef</code> .                                                    |
| The production gap is visible        | The lab names missing authorization integration, trace export, dashboards, deployment, and incident handling. |

Record the command, output, and failure behavior in the lab completion worksheet.

## Production Extension

Replace prefix routing with a typed tool request:

- tool name
- JSON schema for arguments
- authorization check
- timeout
- structured success or error result
- trace ID for the run

Do not give the model broad access to arbitrary functions. Expose narrow tools with explicit permissions.

## Production Bridge

Use this table when adapting the lab to a real product tool:

| Lab Concept                        | Production Version                                                              |
|------------------------------------|---------------------------------------------------------------------------------|
| <code>ToolName</code> union        | Versioned tool manifest with owner, permission, timeout, and side-effect class. |
| <code>disclosedTools(route)</code> | Route, actor, tenant, and risk-based tool disclosure.                           |
| <code>validateProposal(...)</code> | Schema, policy, approval, budget, and state validation.                         |
| <code>trust</code> field           | Data classification that separates trusted system data from untrusted content.  |
| <code>evidenceRef</code>           | Trace-linked evidence reference for audit, replay, and evals.                   |
| Demo console output                | Structured response plus trace event, eval record, and dashboard signal.        |

The first production milestone is not adding more tools. It is proving that one tool can be called, denied, traced, and tested safely.

## Cross-Framework Mapping

- In LangGraph, this boundary usually appears as a tool node or callable bound to graph state.
- In Mastra AI, it maps to a typed tool exposed through an agent or workflow.
- In AutoGen-style systems, it maps to a function/tool call proposed by an assistant and executed by the runtime.
- In CrewAI, it maps to tools assigned to an agent role, with the flow or crew configuration limiting access.

## Related Chapters

- [MCP-first Tool Use](#)
- [Human Approval Gates](#)
- [Policy Enforcement](#)



# Lab 02 - Build an Agent Loop with Planning

Download the [Lab 02 planning loop guided exercise worksheet](#), [lab completion worksheet](#), and [lab production readiness worksheet](#) before you start.

## Objective

Separate planning from execution. The planner decides the steps; the executor runs bounded operations and records results. This gives you the control structure behind many agent loops without making every step autonomous.

## What You Will Use

- Language: TypeScript, with a Python mirror
- Framework/runtime: framework-neutral planner and executor
- Framework-agnostic lesson: planning and execution are separate responsibilities even when a framework packages them together.
- Pattern chapters: [Agent Loop](#), [Planning and Execution](#)
- Source folder: `planning-pattern/`
- Download: [planning-and-execution.zip](#)
- Main files:
  - `planning-pattern/typescript/src/planner.ts`
  - `planning-pattern/typescript/src/executor.ts`
  - `planning-pattern/typescript/src/run.ts`

## Exercise Time Budget

These estimates assume dependencies are already installed.

| Exercise                          | Time      | Output                                                                 |
|-----------------------------------|-----------|------------------------------------------------------------------------|
| Setup and baseline test           | 5-8 min   | Passing planner/executor test output.                                  |
| Run the baseline plan trace       | 10-12 min | Plan steps, result map, and stop signal.                               |
| Change input and inspect failures | 10-15 min | Evidence for changed input plus unsupported or missing-input behavior. |
| Write the stop-condition rule     | 10-15 min | A caller-facing stop reason and production handling note.              |
| Complete the review gate          | 5 min     | Worksheet notes for execution boundary and production gap.             |

## Setup

From the repository root:

```
npm install
```

## Run It

Run the deterministic test:

```
npm run plan:test
```

Run the CLI path:

```
npm run plan:run -- "Compute average of [1,2,3,4]"
```

Run the Python mirror:

```
npm run plan:py
```

## Inspect The Code

Open `planning-pattern/typescript/src/planner.ts` and inspect how a task becomes a list of steps. Then open `planning-pattern/typescript/src/executor.ts` and inspect how execution turns those steps into named results.

Look for these boundaries:

- the plan object
- step IDs
- deterministic executor functions
- result map
- failure surface

## Change One Thing

Change the task text:

```
npm run plan:run -- "Compute average of [10,20,30]"
```

Then inspect whether the deterministic fallback still produces the expected plan shape.

## Expected Result

The test should print:

```
Planning test OK
```

The CLI should print a plan, progress events, and a computed result:

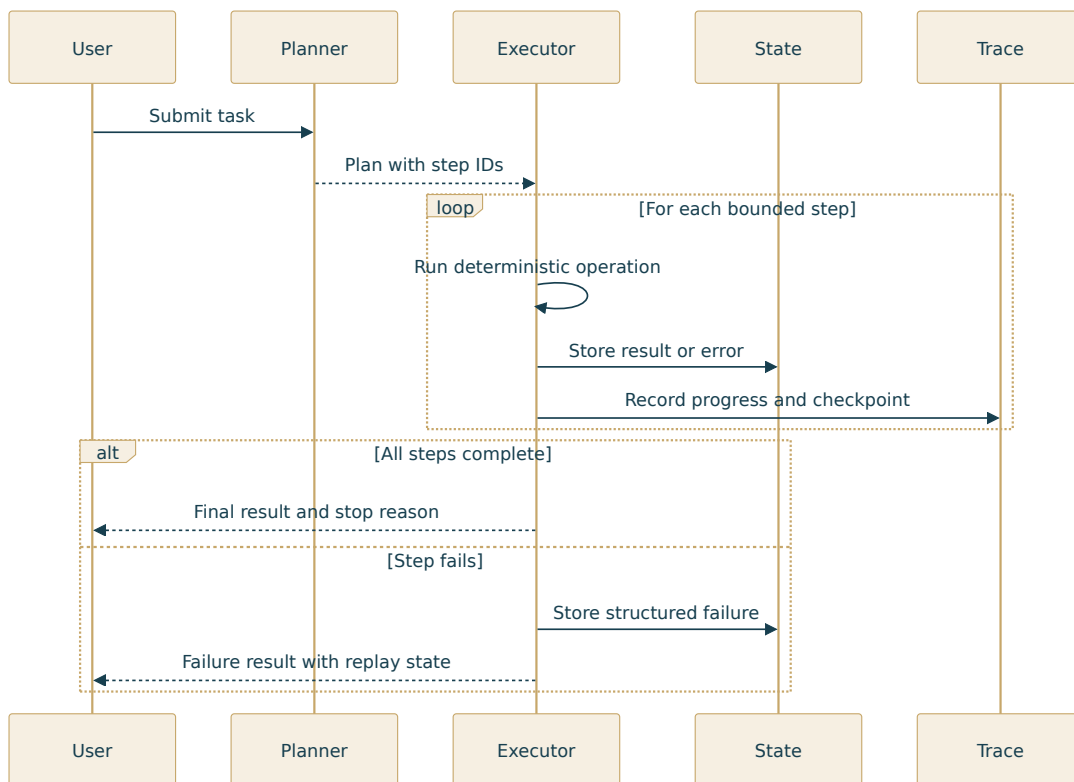
```
Plan: {
 steps: [
 { id: 's1', description: 'Load numbers [1,2,3,4]' },
 { id: 's2', description: 'Compute average' }
],
 rationale: 'synthetic'
}
Progress 0 s1
Progress 50 s2
Progress 100 done
Results: { s1: [1, 2, 3, 4], s2: 2.5 }
```

After changing the task to `[10,20,30]`, the result should be:

```
Results: { s1: [10, 20, 30], s2: 20 }
```

If you extend the planner, keep the executor deterministic and testable.

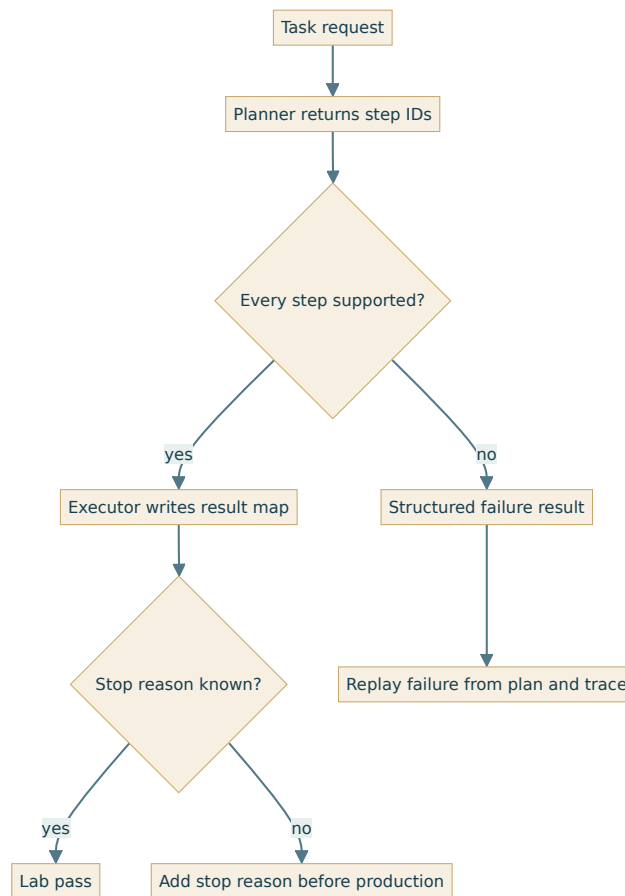
Use this flow as the acceptance model for the lab. The planner may choose the steps, but execution remains bounded, traceable, and replayable.



## Guided Exercises

Use these exercises to turn the happy path into a reviewable loop trace.

| Exercise               | Time   | What To Do                                                                                     | Evidence To Save                                                                                                         |
|------------------------|--------|------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| Baseline plan trace    | 10 min | Run <code>npm run plan:test</code> and the CLI command for <code>[1,2,3,4]</code> .            | Plan steps, progress events, result map, and stop signal.                                                                |
| Input-change trace     | 8 min  | Run the CLI command for <code>[10,20,30]</code> .                                              | The changed <code>s1</code> input and changed <code>s2</code> average.                                                   |
| Unsupported-step trace | 8 min  | Inspect the negative cases in <code>planning-pattern/typescript/test/planning.spec.ts</code> . | A structured <code>unsupported_step</code> failure instead of <code>null</code> .                                        |
| Missing-input trace    | 8 min  | Inspect the test case that runs <code>Compute average</code> without loading numbers first.    | A structured <code>missing_numbers</code> failure.                                                                       |
| Production stop rule   | 10 min | Write the stop reason you would expose to a caller.                                            | <code>success</code> , <code>unsupported_step</code> , <code>missing_numbers</code> , or <code>budget_exhausted</code> . |



## Stop-Condition Exercise

The executor now returns a failure object for unsupported or malformed work. Use this as the model for production stop conditions:

```
await executePlan([
 { id: "s9", description: "Send refund directly" }
]);
```

Expected evidence:

```
{
 "s9": {
 "status": "failed",
 "error_type": "unsupported_step",
 "step_id": "s9",
 "description": "Send refund directly"
 }
}
```

This failure is valuable because it is replayable. A production loop should never make the reviewer infer whether a plan stopped because it finished, failed validation, exhausted budget, or hit policy.

## Lab Review Gate

Before moving on, verify the control loop:

| Check                               | Evidence                                                                                  |
|-------------------------------------|-------------------------------------------------------------------------------------------|
| Planning is separate from execution | <code>planner.ts</code> produces steps; <code>executor.ts</code> runs bounded operations. |
| Steps are identifiable              | Each step can be named, traced, and connected to a result.                                |
| The executor stays deterministic    | The same supported plan produces the same result.                                         |
| Failure has a surface               | Unsupported or malformed tasks can return a structured failure path.                      |
| The stop condition is explicit      | The lab can explain why the run ended.                                                    |

Record the plan shape, result map, and one failure path in the lab completion worksheet.

## Production Extension

Add loop controls before using this pattern in production:

- maximum steps
- maximum retries
- stop reason
- checkpoint after each step
- structured error result
- human review for high-risk steps

Planning is useful only when execution is bounded and inspectable.

## Production Bridge

Use this table when adapting the lab to a product workflow:

| Lab Concept            | Production Version                                                      |
|------------------------|-------------------------------------------------------------------------|
| Plan object            | Versioned plan schema with owner and risk class.                        |
| Step IDs               | Workflow step IDs with trace spans and retry policy.                    |
| Result map             | Durable state with checkpoint, status, and error class.                 |
| Deterministic executor | Tool or workflow node with timeout, idempotency, and policy gate.       |
| CLI run                | Request envelope with actor, tenant, budget, trace ID, and stop reason. |

The first production milestone is a replayable plan. If a failure cannot be replayed from state and trace, the loop is still a demo.

## Cross-Framework Mapping

- In LangGraph, planning and execution can be separate nodes connected through shared graph state.
- In Mastra AI, the same split can appear as a workflow that coordinates agent and tool steps.
- In AutoGen-style systems, a manager agent may propose a plan while executor functions perform bounded work.
- In CrewAI, a flow can own the sequence while crews or agents handle delegated tasks.

## Related Chapters

- [Goals and State](#)
- [Self-Healing Workflows](#)
- [Durable Workflows](#)

# Lab 03 - Build Agentic RAG

Download the [Lab 03 Agentic RAG guided exercise worksheet](#), [lab completion worksheet](#), and [lab production readiness worksheet](#) before you start.

## Objective

Build the retrieval boundary behind an Agentic RAG system: retrieve scoped evidence, inject it into context, answer from that evidence, and refuse or escalate when the evidence is not enough.

## What You Will Use

- Language: Python
- Framework/runtime: LangChain/LangGraph-style retrieval stack with FAISS and Hugging Face embeddings
- Framework-agnostic lesson: retrieval produces scoped evidence; generation should stay grounded in that evidence.
- Pattern chapters: [Semantic Recall and RAG](#), [Agentic RAG Systems](#)
- Source folder: `context-engineering-pattern/`
- Download: [semantic-recall-rag.zip](#)
- Main file: `context-engineering-pattern/langgraph_python_example/rag_example.py`
- Native comparison: `native-framework-examples/langgraph-research-rag/` ([download](#))

## Exercise Time Budget

These estimates assume the Python environment is ready.

| Exercise                       | Time      | Output                                                              |
|--------------------------------|-----------|---------------------------------------------------------------------|
| Setup and baseline retrieval   | 10-15 min | Query, retrieved context, and generated or fallback answer.         |
| Change the corpus or query     | 10-15 min | Evidence that the answer changes because retrieval changed.         |
| Run the missing-evidence check | 10-15 min | Refusal or escalation note for unsupported questions.               |
| Sketch the source contract     | 15 min    | Source packet fields for owner, freshness, access, and allowed use. |
| Compare the native graph       | 15 min    | Mapping from retrieval steps to graph nodes and eval gates.         |



## Setup

The RAG example is Python-based. It can run in local fallback mode without a model key or optional vector-store dependencies. Install the full requirements when you want the FAISS and Hugging Face path.

From the repository root:

```
python3 -m venv .venv-rag
source .venv-rag/bin/activate
pip install -r context-engineering-pattern/langgraph_python_example/requirements.txt
```

For live answer generation, set `MISTRAL_API_KEY`. Without the key, the script prints a deterministic local answer from retrieved context.

## Run It

```
python3 context-engineering-pattern/langgraph_python_example/rag_example.py
```

## Inspect The Code

Open `context-engineering-pattern/langgraph_python_example/rag_example.py` and find:

- `docs`: the tiny demo corpus.
- `HuggingFaceEmbeddings`: the embedding model.
- `FAISS.from_texts`: the vector index.
- `retrieve(query)`: the retrieval boundary.
- `chat_mistral(messages)`: the generation boundary.

The key design move is to keep retrieved evidence separate from the instruction. The model should answer from the context, not from vague memory.

## Change One Thing

Add a new document to the `docs` list:

```
{"content": "Agentic RAG uses retrieval, planning, tool use, and verification around a knowledge base."}
```

Then change the query:

```
query = "What makes RAG agentic?"
```

## Guided Exercises

Use these exercises to turn the lab into an evidence pack. Each exercise should leave one note in the guided worksheet.

| Exercise                 | Time   | Action                                                                                                                                         | Done When                                                                                                |
|--------------------------|--------|------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|
| Baseline retrieval trace | 10 min | Run the script unchanged and copy the query, retrieved context, and answer.                                                                    | You can point to the exact text that entered generation.                                                 |
| Grounding change         | 15 min | Add the Agentic RAG document above and change the query.                                                                                       | The answer changes because retrieved evidence changed.                                                   |
| Missing-evidence check   | 15 min | Ask a question the tiny corpus cannot support, such as <code>What is the refund policy?</code> .                                               | You can explain why this demo should refuse or escalate in production.                                   |
| Source contract sketch   | 15 min | Rewrite one document as a source packet with <code>source_id</code> , <code>owner</code> , <code>freshness</code> , and <code>allowed</code> . | You can name the metadata a real retriever must carry.                                                   |
| Native graph comparison  | 15 min | Open <code>native-framework-examples/langgraph-research-rag/research_rag_graph.py</code> and compare its nodes with this lab.                  | You can map access policy, retrieval, filtering, answer synthesis, escalation, and evals to graph nodes. |

For the missing-evidence check, do not treat a fluent answer as success. The success signal is that you can identify the unsupported claim and the production control that should block it.

## Expected Result

Without a model key, the script should print a local fallback answer:

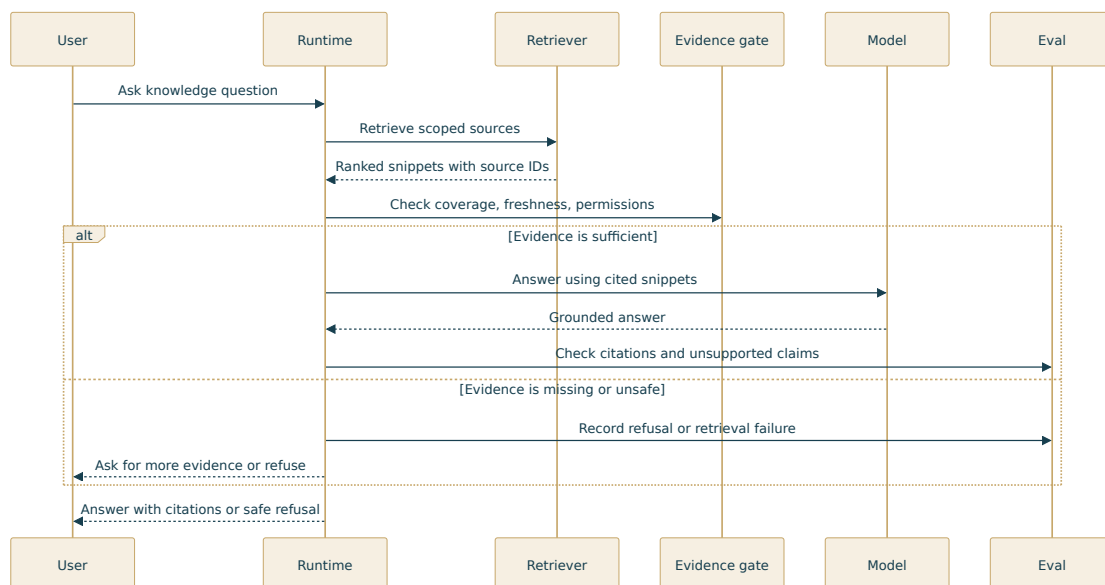
```
Answer: Local fallback answer from retrieved context: Agentic systems are autonomous AI systems.
```

```
Prompt engineering improves LLM outputs.
```

With `MISTRAL_API_KEY`, the answer should reflect the retrieved context through the Mistral generation call.

After you add the new document and change the query, the answer should reflect the new document. If the answer does not cite or use retrieved evidence, the retrieval boundary is not doing enough work.

Use this flow as the lab's acceptance model: every answer must pass through scoped retrieval, evidence checks, and citation evaluation before it reaches the user.



## Lab Review Gate

Before moving on, verify the evidence path:

| Check                          | Evidence                                                                               |
|--------------------------------|----------------------------------------------------------------------------------------|
| Retrieval is visible           | The code has a clear <code>retrieve(query)</code> boundary.                            |
| Evidence is scoped             | Retrieved text is separated from instructions and generation logic.                    |
| Local fallback works           | The script can answer from retrieved context without a provider key.                   |
| The answer uses evidence       | The response changes when the corpus and query change.                                 |
| Missing evidence is recognized | You can name what the demo would do poorly when no approved document exists.           |
| Production controls are named  | The lab identifies source IDs, ACLs, freshness, citation checks, and refusal behavior. |

Record the query, retrieved evidence, answer, and missing-evidence gap in the lab completion worksheet.

## Source-Grounding Exercise

Use this mini-review after the grounding change:

| Question                                        | Answer                                                                                             |
|-------------------------------------------------|----------------------------------------------------------------------------------------------------|
| Which document should be retrieved first?       | The document that mentions Agentic RAG.                                                            |
| Which answer sentence depends on that document? | The sentence explaining that Agentic RAG wraps retrieval with planning, tool use, or verification. |

| Question                                        | Answer                                                                       |
|-------------------------------------------------|------------------------------------------------------------------------------|
| Which sentence would be unsupported without it? | Any claim about “what makes RAG agentic.”                                    |
| What would production add?                      | Source ID, citation label, freshness rule, access filter, and citation eval. |

If the answer does not change after adding the new document, inspect the retrieval boundary first. The problem may be query wording, ranking, missing metadata, or a context packet that gives the model too little evidence.

## Intentional Failure Exercise

Change the query to a topic outside the corpus:

```
query = "What is the refund policy?"
```

The toy script will still build a context packet from the closest available documents. That is acceptable for a minimal demo, but it is not acceptable production behavior. Record the failure as:

```
failure_type: missing_approved_evidence
observed_behavior: local fallback used unrelated retrieved context
production_behavior: refuse, clarify, or retrieve from an approved refund-policy source
eval_fixture: question should not answer unless approved refund-policy evidence is present
```

This is the key lesson of the lab: retrieval is not enough. The system needs an evidence gate that can say “the corpus does not support this answer.”

## Production Extension

Add production controls:

- document IDs
- source URLs
- freshness timestamps
- access-control filters
- citation requirements
- prompt-injection checks on retrieved text
- refusal when evidence is missing

Agentic RAG is not only vector search. It is a controlled loop around retrieval, evidence, tool use, and verification.

## Production Bridge

Use this table when adapting the lab to a real knowledge system:

| Lab Concept                 | Production Version                                                               |
|-----------------------------|----------------------------------------------------------------------------------|
| Tiny <code>docs</code> list | Governed corpus with source IDs, owners, ACLs, freshness, and retention rules.   |
| FAISS demo index            | Managed retrieval service with index version, filters, evals, and rollback.      |
| Retrieved text              | Evidence packet with source metadata, omitted-source notes, and citation labels. |
| Mistral generation call     | Answer synthesis constrained to approved evidence.                               |
| Manual inspection           | Citation eval, stale-source eval, forbidden-source eval, and escalation test.    |

The first production milestone is not a bigger index. It is a retrieval path that can refuse unsupported answers and explain which evidence it used.

## Native Framework Extension

After the deterministic RAG example passes, compare it with `native-framework-examples/langgraph-research-rag/`. The native slice maps the same boundary to a LangGraph `StateGraph` with nodes for access policy, retrieval, source filtering, answer synthesis, escalation, and citation evals.

Completion standard: the native graph should prove that stale and forbidden sources are omitted before answer synthesis, and that missing approved evidence escalates instead of producing an unsupported answer.

## Cross-Framework Mapping

- In LangGraph, retrieval can be a graph node that updates state with evidence before generation.
- In LangChain, this maps to retrievers, document loaders, vector stores, and chains or runnables.
- In Mastra AI, retrieval becomes a knowledge or tool capability used by an agent or workflow.
- In CrewAI, a research role may retrieve evidence, but the flow still needs to validate grounding and citations.

## Related Chapters

- [Context Engineering](#)
- [Knowledge-Bound Agents](#)
- [Working Memory](#)
- [Research RAG Agent Capstone](#)

# Lab 04 - Build A2A Agent Communication

Download the [lab completion worksheet](#) and [lab production readiness worksheet](#) before you start.

## Objective

Build a typed communication boundary between two agents. One agent requests work, another agent accepts, refuses, errors, or receives cancellation through explicit messages.

## What You Will Use

- Language: TypeScript
- Framework/runtime: protocol-first runtime with Ajv JSON schema validation
- Framework-agnostic lesson: agent communication needs typed envelopes, correlation IDs, refusal states, error states, and cancellation.
- Pattern chapter: [A2A Agent Interoperability](#)
- Source folder: `agent-to-agent-communication-pattern/`
- Download: [a2a-agent-interoperability.zip](#)
- Main files:
  - `agent-to-agent-communication-pattern/src/agent_a.ts`
  - `agent-to-agent-communication-pattern/src/agent_b.ts`
  - `agent-to-agent-communication-pattern/src/bus_memory.ts`
  - `agent-to-agent-communication-pattern/protocol/a2a.schema.json`

## Exercise Time Budget

These estimates assume dependencies are already installed.

| Exercise                          | Time      | Output                                                             |
|-----------------------------------|-----------|--------------------------------------------------------------------|
| Setup and baseline protocol test  | 8 min     | Passing A2A test output.                                           |
| Inspect the message schema        | 10-12 min | Notes on task IDs, states, refusals, errors, and cancellation.     |
| Change one message path           | 10-15 min | A visible success, refusal, invalid-input, or cancellation result. |
| Review protocol failure ownership | 10-15 min | Owner and handling rule for malformed or cancelled work.           |
| Complete production mapping       | 5-10 min  | Transport, auth, replay, and schema-version notes.                 |

## Setup

From the repository root:

```
npm install
```

## Run It

Run the protocol test:

```
npm run a2a:test
```

Run the demo:

```
npm run a2a:run
```

## Inspect The Code

Open `agent-to-agent-communication-pattern/src/agent_a.ts` and `agent-to-agent-communication-pattern/src/agent_b.ts`.

Look for:

- handshake
- task request
- task response
- refusal
- invalid input error
- cancellation
- schema validation with Ajv

## Change One Thing

In the test, add a second valid task request with a different ID:

```
a.requestTask('t5', 'sum', { a: 10, b: 15 });
```

Run:

```
npm run a2a:test
```

## Expected Result

The receiver should process the new task without confusing it with the existing task IDs. If correlation IDs are missing, progress and results become hard to match.

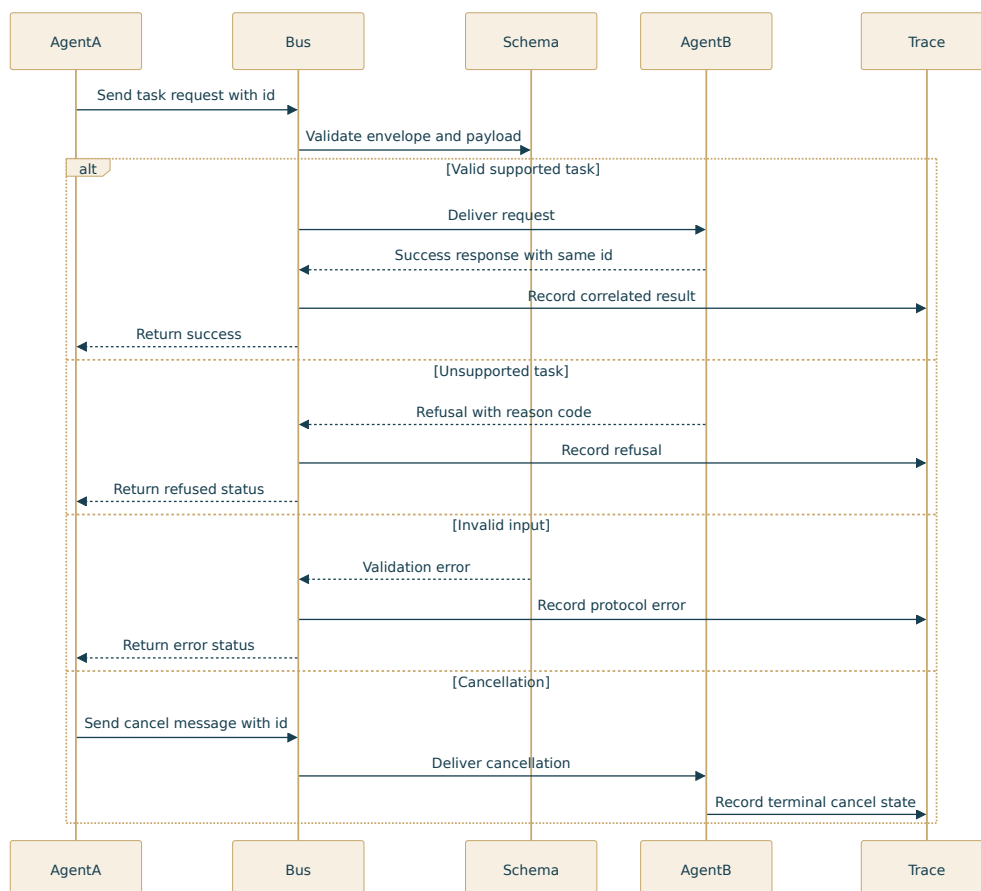
The test should print four outcomes:

```
AgentA received response: { id: 't1', status: 'success', output: { sum: 3 } }
AgentA received response: { id: 't2', status: 'refused', error: 'unsupported_task' }
AgentA received response: { id: 't3', status: 'error', error: 'invalid_input' }
AgentB cancel received: { id: 't4', reason: 'user_request' }
A2A tests executed: success, refusal, error, cancel
```

The demo should print the happy path:

```
AgentA received response: { id: 't1', status: 'success', output: { sum: 7 } }
```

Use this flow as the acceptance model for the lab. A2A communication is healthy only when the protocol handles success, refusal, invalid input, and cancellation with the same correlation discipline.



## Lab Review Gate

Before moving on, verify the protocol boundary:



| Check                          | Evidence                                                                                |
|--------------------------------|-----------------------------------------------------------------------------------------|
| Messages are typed             | The A2A schema validates task requests, responses, refusals, errors, and cancellations. |
| Correlation is explicit        | Each task has an ID that connects request, progress, result, and cancellation.          |
| Refusal is a first-class state | The receiver can reject unsupported work without pretending success.                    |
| Invalid input is controlled    | Malformed payloads produce an error outcome.                                            |
| Cancellation is observable     | Cancelled work has a message and terminal state.                                        |

Record the task IDs, message types, and one invalid-input case in the lab completion worksheet.

## Production Extension

Before using A2A across services, add:

- authentication
- authorization
- idempotency keys
- task leases
- retry policy
- durable task state
- audit logs
- transport-level encryption

A2A is a protocol boundary, not just one agent calling another function.

## Production Bridge

Use this table when adapting the lab to service-to-service agent communication:

| Lab Concept          | Production Version                                              |
|----------------------|-----------------------------------------------------------------|
| In-memory bus        | Authenticated transport with durable delivery and retry policy. |
| Task ID              | Correlation ID plus idempotency key and trace ID.               |
| JSON schema          | Versioned protocol contract with compatibility tests.           |
| Refusal message      | Policy-aware denial with reason code and audit record.          |
| Cancellation message | Lease, timeout, cancellation reason, and cleanup behavior.      |

The first production milestone is a message contract that can survive retries, refusal, cancellation, and replay without losing ownership.

## Cross-Framework Mapping

- In LangGraph, this resembles graph-to-graph or service-to-service handoff with typed state.
- In Mastra AI, this maps to service or workflow boundaries around agent calls.
- In AutoGen-style systems, it maps to structured messages between agents rather than free-form chat alone.
- In CrewAI, it maps to task handoffs and role outputs that still need schema, identity, and trace correlation.

## Related Chapters

- [Secure Agent Communication](#)
- [MCP-first Tool Use](#)
- [Open Personal Agent Architectures](#)

# Lab 05 - Build a Multi-Agent Supervisor

Download the [lab completion worksheet](#) and [lab production readiness worksheet](#) before you start.

## Objective

Build the supervision shape behind multi-agent systems: one coordinator owns the goal, delegates bounded work, gathers worker outputs, and produces the final answer.

## What You Will Use

- Language: TypeScript
- Framework/runtime: AutoGen-style manager/worker example
- Framework-agnostic lesson: a supervisor owns decomposition, worker contracts, merge policy, and final synthesis.
- Pattern chapters: [Supervisor / Worker](#), [Task Delegation](#)
- Source folder: `hierarchical-agent-pattern/`
- Download: [supervisor-worker.zip](#)
- Main file: `hierarchical-agent-pattern/autogen_typescript_example/hierarchical_agent.ts`

## Exercise Time Budget

These estimates assume dependencies are already installed.

| Exercise                                | Time     | Output                                                         |
|-----------------------------------------|----------|----------------------------------------------------------------|
| Setup and baseline run                  | 8-10 min | Deterministic or live supervisor output.                       |
| Inspect supervisor and worker contracts | 12 min   | Notes on goal ownership, role boundaries, and merge behavior.  |
| Change delegation or parsing            | 12 min   | Evidence that bounded worker output still controls synthesis.  |
| Review aggregation failure              | 10 min   | Handling rule for missing, conflicting, or weak worker output. |
| Complete production mapping             | 5-10 min | Worker contract, merge policy, trace, and final-owner notes.   |

## Setup

This example can run in two modes:

- Local deterministic mode when `MISTRAL_API_KEY` is not set.
- Live Mistral mode when `MISTRAL_API_KEY` is present in `.env`.

From the repository root:

```
npm install
cp .env.example .env
```

Set `MISTRAL_API_KEY` in `.env` only if you want the live-model path. Leave it unset if you want the deterministic lab output below.

## Run It

```
npm run hierarchical-agent
```

When prompted, enter a goal such as:

```
Draft a short plan for evaluating an agentic RAG prototype.
```

## Inspect The Code

Open `hierarchical-agent-pattern/autogen_typescript_example/hierarchical_agent.ts` and find:

- the manager prompt
- worker prompts
- subtask extraction
- aggregation prompt
- final answer path

The supervisor owns decomposition and final acceptance. Workers should not silently redefine the goal.

## Change One Thing

Change the manager instruction so it asks for three subtasks instead of two. Then inspect the parsing logic:

```
const subTasks = managerPlan.match(/Sub-task [12]: (.*)/g) || [];
```

Update the regex so the code can collect the third subtask.

## Expected Result

The manager should produce a plan, workers should produce bounded results, and the final aggregation should combine the worker outputs. If the parsing is brittle, the supervisor loses work.

With no `MISTRAL_API_KEY`, the deterministic path should show this shape:

#### Manager Agent Plan:

Sub-task 1: Define evaluation criteria for answer quality, retrieval grounding, latency, and failure handling.

Sub-task 2: Create a small test set with expected evidence, negative cases, and acceptance thresholds.

#### Worker Agent 1 Result:

Worker 1 result: Sub-task 1: Define evaluation criteria for answer quality, retrieval grounding, latency, and failure handling. Criteria should include citation accuracy, unsupported-claim rate, p95 latency, and visible refusal behavior.

#### Worker Agent 2 Result:

Worker 2 result: Sub-task 2: Create a small test set with expected evidence, negative cases, and acceptance thresholds. The test set should include grounded answers, missing-evidence questions, stale-document checks, and threshold failures.

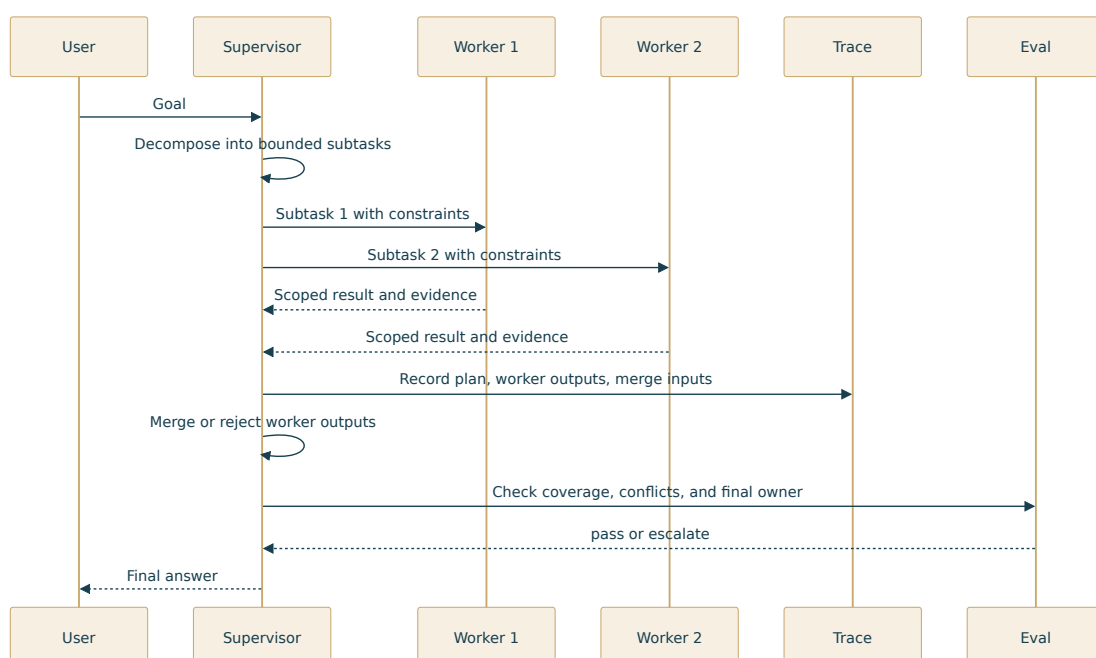
#### Final Aggregated Result for User:

Final answer: evaluate the RAG prototype with quality, grounding, latency, and failure-handling criteria.

Use a small test set with positive cases, missing-evidence cases, stale-document cases, and blocking thresholds.

Accept the prototype only when worker evidence meets the supervisor policy.

The exact wording may differ in live-model mode. The required signal is the same: one manager plan, bounded worker outputs, and a final answer that uses those outputs.



Use this flow as the lab’s acceptance model. The supervisor owns decomposition, worker contracts, merge policy, trace evidence, and final acceptance.

## Lab Review Gate

Before moving on, verify the supervision boundary:

| Check                        | Evidence                                                                            |
|------------------------------|-------------------------------------------------------------------------------------|
| The supervisor owns the goal | The manager prompt decomposes work without letting workers redefine the request.    |
| Worker work is bounded       | Each worker receives a specific subtask and returns a scoped result.                |
| Aggregation is explicit      | The final answer uses worker outputs instead of hiding merge behavior.              |
| Parsing risk is visible      | The regex change shows why natural-language plans need structured output.           |
| Failure ownership is named   | The lab can explain who handles missing, conflicting, or low-quality worker output. |

Record the manager plan, worker outputs, aggregation result, and parsing gap in the lab completion worksheet.

## Production Extension

Replace natural-language subtask parsing with structured output:

- `subtasks: Array<{ id, role, objective, constraints, expected_output }>`
- worker-specific permissions
- per-worker timeout
- merge policy
- judge or evaluator
- human escalation for disagreement

Multi-agent systems need strong contracts. More agents without a merge policy usually means more failure modes.

## Production Bridge

Use this table when adapting the lab to a production multi-agent workflow:

| Lab Concept    | Production Version                                                          |
|----------------|-----------------------------------------------------------------------------|
| Manager prompt | Supervisor policy with task schema, routing rules, and acceptance criteria. |

| Lab Concept           | Production Version                                                             |
|-----------------------|--------------------------------------------------------------------------------|
| Worker prompts        | Role contracts with permissions, tools, timeouts, and expected output schema.  |
| Regex subtask parsing | Structured output with validation and retry-on-invalid behavior.               |
| Aggregation prompt    | Merge policy with conflict handling, source attribution, and final owner.      |
| Console result        | Trace with per-worker spans, cost, latency, stop reason, and evaluator result. |

The first production milestone is not adding more agents. It is proving that delegated work can be scoped, merged, rejected, and audited.

### Cross-Framework Mapping

- In LangGraph, this maps to a coordinator graph that routes work to specialized nodes or subgraphs.
- In Mastra AI, this maps to workflows that coordinate agents and tools under one runtime.
- In AutoGen-style systems, this is the manager/worker conversation pattern made explicit.
- In CrewAI, this maps to crews with role-specific tasks, while a flow should still own state and final acceptance.

### Related Chapters

- [Parallel Agents](#)
- [Debate and Consensus](#)
- [CrewAI Flows and Crews](#)

# Lab 06 - Add Observability and Evals

Download the [Lab 06 observability and evals guided exercise worksheet](#), [lab completion worksheet](#), and [lab production readiness worksheet](#) before you start.

## Objective

Turn the examples into something you can evaluate. A production agent needs trace data, regression tasks, expected outcomes, and failure review before it needs more autonomy.

## What You Will Use

- Language: TypeScript
- Framework/runtime: framework-neutral tests over the repository examples
- Framework-agnostic lesson: evals should inspect trajectories, tool calls, policy outcomes, and stop reasons, not only final answers.
- Pattern chapters: [Observability and Evals](#), [Evaluator-Optimizer](#)
- Source folder: `observability-and-evals-pattern/`
- Download: [observability-and-evals.zip](#)
- Existing test commands:
  - `npm run observability:test`
  - `npm run plan:test`
  - `npm run a2a:test`
  - `npm run mcp:test`
  - `npm test`

## Exercise Time Budget

These estimates assume dependencies are already installed.

| Exercise                             | Time      | Output                                               |
|--------------------------------------|-----------|------------------------------------------------------|
| Setup and baseline eval run          | 8-10 min  | Passing scoped test output.                          |
| Inspect the trace contract           | 10-12 min | Required trace fields and protected invariant.       |
| Review negative cases                | 15-20 min | Failure reason, severity, and release-blocking note. |
| Sketch CI and incident-to-eval gates | 15-20 min | Command, owner, threshold, and future fixture name.  |
| Complete the review gate             | 5-10 min  | Worksheet notes for trace, eval, and production gap. |



## Setup

From the repository root:

```
npm install
```

## Run It

Run the deterministic checks:

```
npm run observability:test
npm run plan:test
npm run a2a:test
npm run mcp:test
```

Then run the full suite:

```
npm test
```

## Inspect The Code

Use the test files as the first eval dataset:

- `observability-and-evals-pattern/trace-contract.spec.ts`
- `observability-and-evals-pattern/trace-contract.ts`
- `planning-pattern/typescript/test/planning.spec.ts`
- `agent-to-agent-communication-pattern/test/a2a.spec.ts`
- `modern-tool-use-pattern/typescript/test/modern-tools.spec.ts`

Each test checks a contract:

- traces contain correlation IDs, model spans, stop reasons, and policy decisions for tool spans
- planning produces executable steps
- A2A handles success, refusal, error, and cancel
- MCP tool use discovers tools and returns useful data

## Change One Thing

Add one negative case to the A2A test by sending malformed input with a new task ID:

```
a.requestTask('t6', 'sum', { a: null, b: 10 } as any);
```

Run:

```
npm run a2a:test
```

## Expected Result

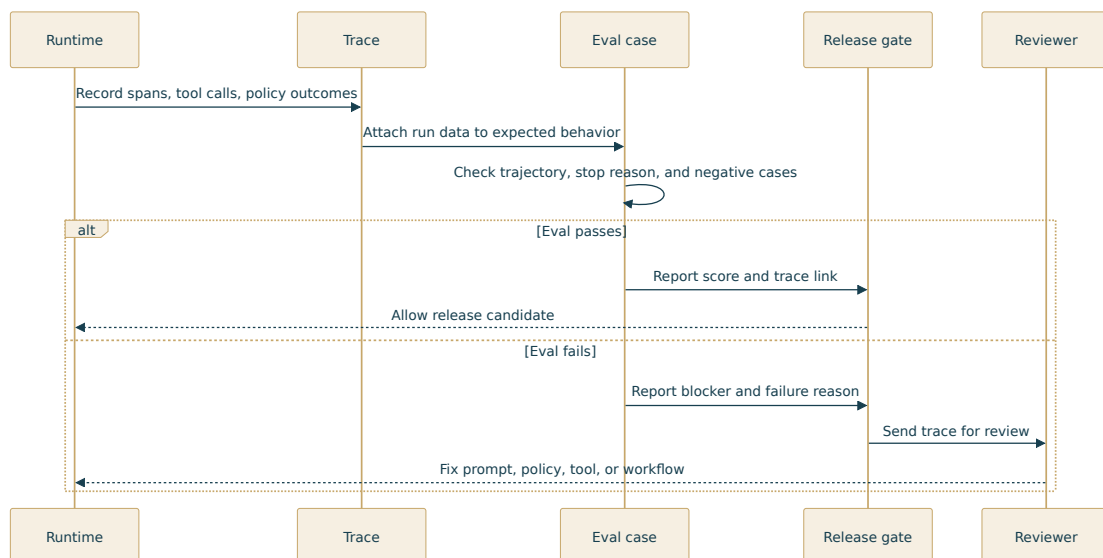
The system should report an error outcome, not a successful result. In an eval dataset, negative cases are as important as happy paths.

The trace contract gate should print:

```
Trace contract test OK
```

That test proves both sides of the eval boundary: a complete trace passes, and a successful tool span without a policy decision fails.

Use this flow as the acceptance model for the lab. Observability captures what happened; evals decide whether the run is safe enough to release.

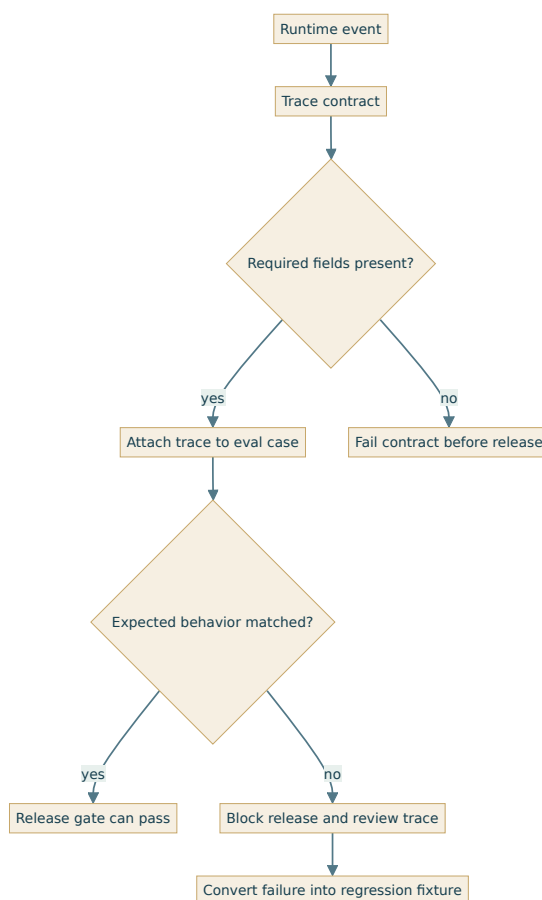


## Guided Exercises

Use these exercises to make the eval boundary visible.

| Exercise                | Time   | What To Do                                                                       | Evidence To Save                                                                                |
|-------------------------|--------|----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| Trace contract baseline | 10 min | Run <code>npm run observability:test</code> .                                    | The passing contract output and the fields protected by the test.                               |
| Missing-policy failure  | 10 min | Inspect <code>missingPolicyTrace</code> in <code>trace-contract.spec.ts</code> . | The exact failure reason: <code>policy decision for tool span span_tool_missing_policy</code> . |

| Exercise              | Time   | What To Do                                                                          | Evidence To Save                                                               |
|-----------------------|--------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| Negative A2A case     | 15 min | Add the malformed A2A input from this lab and rerun <code>npm run a2a:test</code> . | Error outcome, task ID, and why it should block release.                       |
| CI gate sketch        | 10 min | Decide which command should block release for this slice.                           | <code>npm test</code> , a scoped test, or a future eval command with an owner. |
| Incident-to-eval note | 10 min | Pick one failure from the lab and turn it into a regression fixture.                | Fixture name, severity, expected outcome, and trace field.                     |



## Intentionally Failing Eval Exercise

In `observability-and-evals-pattern/trace-contract.spec.ts`, the `missingPolicyTrace` object models a successful tool call without a policy decision. That is a release blocker because the tool used authority without an auditable allow, deny, approval, or escalation decision.

Review the failure as if it came from production:

| Question               | Answer To Record                             |
|------------------------|----------------------------------------------|
| Which span failed?     | <code>span_tool_missing_policy</code>        |
| Which invariant broke? | Every tool span must have a policy decision. |

| Question                   | Answer To Record                                                   |
|----------------------------|--------------------------------------------------------------------|
| What should CI do?         | Fail the eval gate.                                                |
| What should the owner fix? | Add the policy decision before the tool result can count as valid. |

## Lab Review Gate

Before moving on, verify the eval boundary:

| Check                      | Evidence                                                                                                     |
|----------------------------|--------------------------------------------------------------------------------------------------------------|
| Trace contract is enforced | <code>npm run observability:test</code> accepts a complete trace and rejects a missing tool policy decision. |
| Tests check contracts      | Planning, A2A, and MCP tests assert behavior, not only that commands run.                                    |
| Negative cases exist       | The malformed A2A input returns an error outcome.                                                            |
| Trajectory is inspected    | Tests look at steps, messages, tool discovery, or outcomes before final text.                                |
| Release risk is visible    | The lab names which failures would block a production release.                                               |
| Eval ownership is implied  | Each test protects a specific pattern boundary.                                                              |

Record the commands, pass/fail output, negative case, and protected boundary in the lab completion worksheet.

## Production Extension

Create a trace and eval record for every run:

```
{
 "run_id": "run_001",
 "pattern": "a2a-agent-interoperability",
 "input": "sum task",
 "expected": "success with sum",
 "actual": "success with sum",
 "score": 1,
 "stop_reason": "completed",
 "latency_ms": 25
}
```

Then add release gates:

- no schema failures
- no missing trace IDs
- no unexpected tool calls
- no regression in golden tasks
- no unresolved safety or policy failures

Observability records what happened. Evals decide whether it is good enough to ship.

## Production Bridge

Use this table when adapting the lab to production evaluation:

| Lab Concept                | Production Version                                                             |
|----------------------------|--------------------------------------------------------------------------------|
| Test command               | Release gate tied to changed prompt, model, tool, policy, memory, or workflow. |
| Test fixture               | Versioned eval case with owner, severity, and failure reason.                  |
| Console output             | Eval report with pass/fail, trace link, score, threshold, and blocker status.  |
| Negative A2A case          | Regression fixture for schema failure and unsafe trajectory.                   |
| Full <code>npm test</code> | CI gate plus canary monitoring and incident-to-eval workflow.                  |

The first production milestone is not a bigger dashboard. It is a blocking eval that catches a known bad trajectory before release.

## Cross-Framework Mapping

- In LangGraph, traces and checkpoints let you inspect node paths, state changes, and stop conditions.
- In Mastra AI, evals and observability should connect agent, tool, workflow, and memory events.
- In AutoGen-style systems, message logs need to become structured traces before they are reliable eval data.
- In CrewAI, task and flow outputs need eval cases that check role behavior and final synthesis.

## Related Chapters

- [Agent Development Lifecycle](#)
- [Evaluation-Driven Agent Development](#)
- [Policy Enforcement](#)

# Lab 07 - Package Agents, Tools, Workflows, Memory, and Evals

Download the [Lab 07 runtime packaging guided exercise worksheet](#), [lab completion worksheet](#), and [lab production readiness worksheet](#) before you start.

## Objective

Use a Mastra-style TypeScript runtime shape to package the same responsibilities you built from scratch: agent decision, tool execution, workflow control, memory, trace events, and evals.

## What You Will Use

- Language: TypeScript
- Framework/runtime: Mastra-style runtime packaging
- Framework-agnostic lesson: a framework can package primitives, but your application still owns state, policy, tool contracts, traces, and acceptance criteria.
- Pattern chapters: [Mastra Runtime](#), [Building a Minimal Agent Runtime](#), [Observability and Evals](#)
- Source files:
  - `mastra-runtime-pattern/typescript/src/runtime_packaging.ts`
  - `mastra-runtime-pattern/typescript/src/run_demo.ts`
  - `mastra-runtime-pattern/typescript/test/runtime_packaging.spec.ts`

## Exercise Time Budget

These estimates assume dependencies are already installed.

| Exercise                               | Time      | Output                                                               |
|----------------------------------------|-----------|----------------------------------------------------------------------|
| Setup and baseline runtime run         | 10 min    | Demo and test output.                                                |
| Map runtime package boundaries         | 15 min    | Owner notes for agent, tool, workflow, memory, trace, and eval.      |
| Inspect tool order and forbidden tools | 20 min    | Tool-order trace plus forbidden-tool eval result.                    |
| Run rollback and native comparison     | 20-30 min | Disable path and mapping to the native Mastra-style slice.           |
| Complete production mapping            | 10-20 min | Release notes for policy, trace export, eval gate, and upgrade risk. |

## Setup

From the repository root:

```
npm install
```

This lab is deterministic and does not require a model key. The point is runtime packaging, not model quality.

## Run It

```
npm run mastra-runtime:demo
npm run mastra-runtime:test
```

## Expected Result

The test command should print:

```
Mastra-style runtime packaging tests OK
```

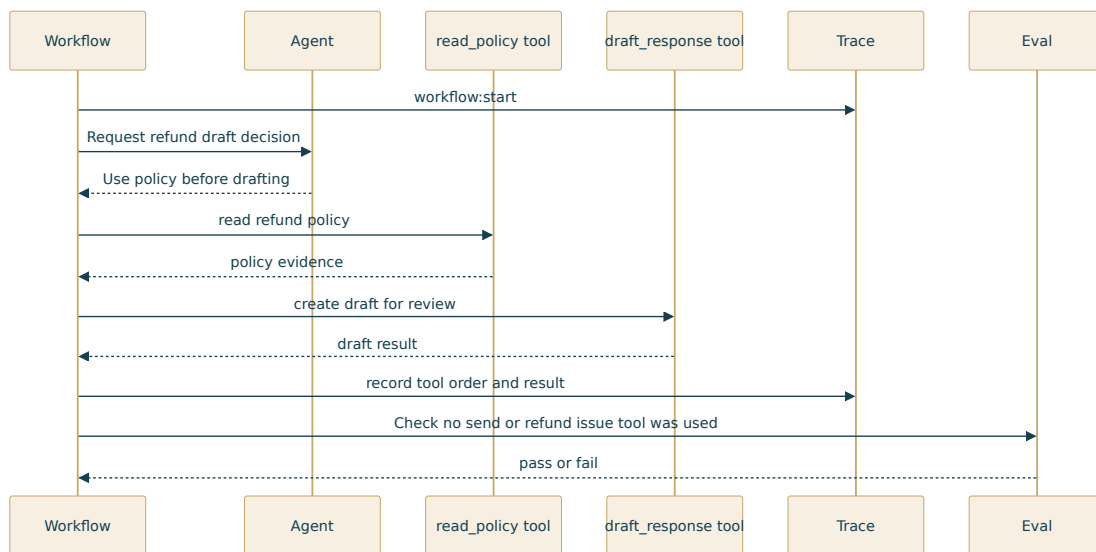
The run should also prove these behavior signals:

- policy is read before drafting;
- the draft is created for review, not sent;
- workflow steps and tool results are traced;
- the eval fails if a direct send or refund-issue tool is used.

The demo command should include this evidence:

```
{
 "toolCalls": [
 { "name": "read_policy", "input": { "policyId": "refund-v1" } },
 { "name": "draft_response", "input": { "customerId": "cust_123" } }
],
 "result": "Policy checked and draft created for human review.",
 "evaluation": {
 "status": "pass"
 }
}
```

The exact JSON also includes memory and trace fields. Use the `toolCalls`, `result`, and `evaluation.status` fields as the quick check.



Use this flow as the lab’s acceptance model. A framework can package the runtime, but the application still owns tool order, side-effect policy, trace evidence, and release evals.

Native Mastra comparison point:

```

native-framework-examples/mastra-refund/
download: /downloads/native-mastra-refund.zip
agent: refundDraftAgent
workflow: refundDraftWorkflow
tools: refund_policy.retrieve, refunds.create_draft
eval gate: refund_draft_no_money_movement

```

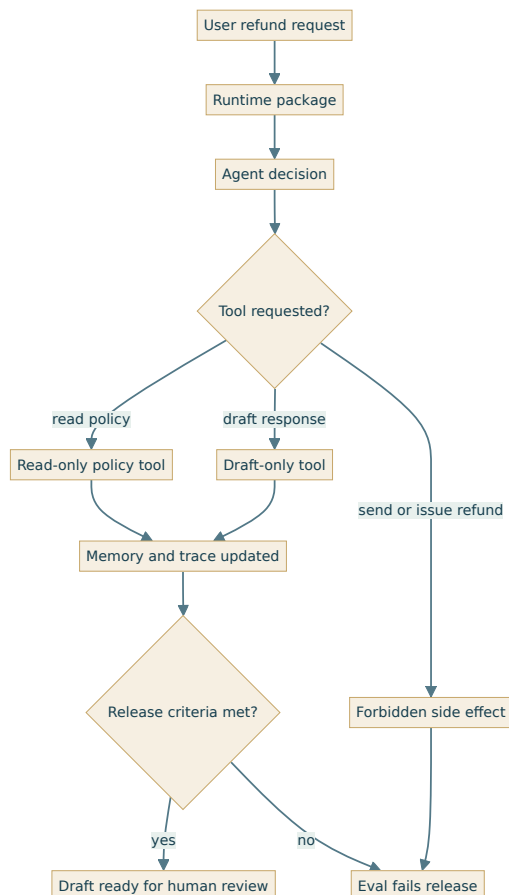
## Guided Exercises

Use these exercises to inspect what the runtime package owns and what product policy still owns.

| Exercise            | Time   | What To Do                                                                                                                                  | Evidence To Save                                                            |
|---------------------|--------|---------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| Runtime package map | 15 min | Inspect <code>Agent</code> , <code>Tool</code> , <code>WorkflowStep</code> , <code>RuntimeState</code> , and <code>evaluateRuntime</code> . | One sentence for what each boundary owns.                                   |
| Tool-order trace    | 10 min | Run <code>npm run mastra-runtime:demo</code> and inspect <code>toolCalls</code> .                                                           | <code>read_policy</code> before <code>draft_response</code> .               |
| Forbidden-tool eval | 10 min | Read the <code>refunds.issue_refund</code> negative case in <code>runtime_packaging.spec.ts</code> .                                        | Eval status and failure reason.                                             |
| Rollback exercise   | 10 min | Name the fastest way to disable the risky capability.                                                                                       | Feature flag, tool removal, workflow route disable, or deployment rollback. |



| Exercise          | Time   | What To Do                                                                                 | Evidence To Save                                            |
|-------------------|--------|--------------------------------------------------------------------------------------------|-------------------------------------------------------------|
| Native comparison | 20 min | Compare the deterministic lab with <code>native-framework-examples/mastra-refund/</code> . | Which parts map to agent, workflow, tools, trace, and eval. |



## Deployment And Rollback Exercise

Treat the lab as a production candidate. Before a real runtime ships, write the rollback action that disables the highest-risk capability without deleting the whole service.

| Capability                        | Rollback Question                                                   | Acceptable Evidence                                        |
|-----------------------------------|---------------------------------------------------------------------|------------------------------------------------------------|
| <code>read_policy</code>          | Can the system fall back to a static policy snapshot?               | Config flag, versioned policy ID, or cached source.        |
| <code>draft_response</code>       | Can draft creation be disabled while policy lookup stays available? | Tool manifest change or workflow route flag.               |
| <code>send_message</code>         | Can direct sending remain impossible unless explicitly approved?    | Forbidden-tool eval and approval gate.                     |
| <code>refunds.issue_refund</code> | Can money movement stay outside this runtime?                       | Missing tool registration plus failing eval if it appears. |

## Inspect The Code

Open `mastra-runtime-pattern/typescript/src/runtime_packaging.ts` and find these boundaries:

- `Agent`: owns instructions and proposes the next decision.
- `Tool`: owns typed capability execution.
- `WorkflowStep`: owns deterministic runtime sequence.
- `RuntimeState`: owns memory, traces, tool calls, and result.
- `evaluateRuntime`: checks trajectory, not only final text.

The local code is intentionally small, but the mapping follows the production shape: agents, workflows, tools, memory, traces, and evals belong in one runtime package.

## Change One Thing

Add a new tool named `send_message` with a direct side effect, then update `evaluateRuntime` so the eval fails if that tool appears in `toolCalls`. The repository test also checks the refund side-effect case:

```
{ name: "refunds.issue_refund", input: { amount: 42 } }
```

Expected lesson: framework tool registration is not permission. Application policy still decides what is safe.

## Verify

Run:

```
npm run mastra-runtime:test
```

Compare the output with the expected result above before moving to the production extension.

## Lab Review Gate

Before moving on, verify the runtime package:

| Check                                | Evidence                                                                   |
|--------------------------------------|----------------------------------------------------------------------------|
| Runtime responsibilities are visible | Agent, tool, workflow, memory, trace, and eval types are separate in code. |
| Tool order is constrained            | Policy lookup happens before draft creation.                               |
| Side effects are blocked             | The eval fails if a direct send or refund issue tool appears.              |
| Trace explains behavior              | The run records workflow, agent decision, and tool events.                 |

| Check                          | Evidence                                                               |
|--------------------------------|------------------------------------------------------------------------|
| Framework ownership is limited | Product policy and release criteria remain outside framework defaults. |

Record the command output, trace events, tool order, and eval result in the lab completion worksheet.

## Production Extension

Before a real Mastra implementation ships, add:

- real tool schemas and input validation;
- workflow retry and compensation policy;
- memory retention, redaction, and deletion controls;
- scorer/eval fixtures tied to release gates;
- trace export to the observability backend;
- approval gates for write tools and external communication.

## Production Bridge

Use this table when adapting the lab to a real Mastra runtime:

| Lab Concept                  | Production Version                                                                |
|------------------------------|-----------------------------------------------------------------------------------|
| <code>Agent</code> type      | Agent route with versioned instructions, allowed tools, and stop reasons.         |
| <code>Tool</code> type       | Typed capability with permission, side-effect class, timeout, and idempotency.    |
| <code>WorkflowStep</code>    | Durable workflow step with retry, compensation, checkpoint, and approval support. |
| <code>RuntimeState</code>    | Persisted run state with tenant, actor, trace ID, budget, and memory policy.      |
| <code>evaluateRuntime</code> | Release gate tied to known incidents, policy rules, and forbidden trajectories.   |

The first production milestone is a runtime slice that can be disabled, replayed, and evaluated without reading framework internals.

## Native Framework Extension

After the deterministic lab passes, port one vertical slice into a real Mastra project. Use [Real Framework Setup Notes](#) for current setup commands and compare your work with the repository example at `native-framework-examples/mastra-refund/`.

Native porting steps:

1. scaffold a Mastra project;
2. create one agent for the refund draft decision;
3. create typed tools for policy lookup and draft creation;

4. create a workflow that calls policy lookup before draft creation;
5. add an eval that fails if a send or refund-issue tool is called;
6. export a trace with workflow, agent, tool, policy, and eval events;
7. document rollback for disabling the draft tool.

Keep these assets outside framework-only code:

| Asset         | Why                                                                        |
|---------------|----------------------------------------------------------------------------|
| tool manifest | product authority should be reviewable without reading framework internals |
| policy rules  | policy must run before tool execution                                      |
| eval fixtures | release gates should survive framework changes                             |
| trace schema  | operations need stable fields across runtimes                              |
| ADR           | the framework choice needs a recorded owner and rollback path              |

Completion standard: the native project proves the same behavior as this lab and links to the [Support Refund Agent capstone](#). A native Mastra run is not complete just because an agent responds; the workflow must preserve tool order, side-effect policy, trace evidence, and eval gates.

## Troubleshooting

| Symptom                                                    | Likely Cause                                                 | Fix                                                                                                 |
|------------------------------------------------------------|--------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| <code>mastra dev</code> cannot find provider credentials   | <code>.env</code> is missing or provider variables are unset | Copy <code>.env.example</code> , set provider keys, and restart the dev server.                     |
| workflow drafts before policy lookup                       | workflow step order is wrong                                 | Keep policy retrieval as the first workflow step and make the draft step consume the policy output. |
| eval passes after adding <code>refunds.issue_refund</code> | forbidden tool list is incomplete                            | Add the tool name to the release eval and rerun the validation.                                     |
| trace explains final text but not tool order               | trace is only model-level                                    | Emit workflow, tool, policy, and eval events separately.                                            |

## Cross-Framework Mapping

- In LangGraph, this becomes graph state, nodes, tools, checkpoints, and evals around graph runs.
- In Mastra AI, these concerns are packaged as agents, workflows, tools, memory, observability, and evals.
- In AutoGen-style systems, the runtime package is split across manager, workers, functions, messages, and transcript evals.

- In CrewAI, the equivalent packaging splits between flow state, crew tasks, agent roles, tools, and flow-level evaluation.

## Related Chapters

- [Mastra Runtime](#)
- [Framework Selection](#)
- [Building a Minimal Agent Runtime](#)
- [Production Evaluation Feedback Loops](#)
- [Support Refund Agent Capstone](#)

# Lab 08 - Model Flows, Crews, Roles, and Task Contracts

Download the [lab completion worksheet](#) and [lab production readiness worksheet](#) before you start.

## Objective

Use a CrewAI-style Python shape to separate flow ownership from crew collaboration. The flow owns state, sequence, evaluation, and final acceptance. The crew owns bounded specialist work.

## What You Will Use

- Language: Python
- Framework/runtime: CrewAI-style flows and crews
- Framework-agnostic lesson: multi-agent value comes from role boundaries, task contracts, flow state, and explicit acceptance, not from adding more agents.
- Pattern chapters: [CrewAI Flows and Crews](#), [Choosing Multi-Agent Topology](#), [Supervisor / Worker](#)
- Source files:
  - `crewai-flows-and-crews-pattern/python/flow_crew.py`
  - `crewai-flows-and-crews-pattern/python/test_flow_crew.py`

## Exercise Time Budget

These estimates assume dependencies are already installed.

| Exercise                           | Time      | Output                                                             |
|------------------------------------|-----------|--------------------------------------------------------------------|
| Setup and baseline flow run        | 10 min    | Flow demo and test output.                                         |
| Inspect flow and role boundaries   | 15 min    | Notes on state ownership, role output, and acceptance.             |
| Change one role or acceptance case | 20 min    | A visible accepted or rejected state.                              |
| Verify and troubleshoot            | 10-15 min | Passing test or recorded failure cause.                            |
| Complete production mapping        | 10-20 min | Notes for checkpoints, permissions, validators, and fallback path. |

## Setup

From the repository root:

```
npm install
```

This lab uses only Python standard library code. It is intentionally deterministic so you can inspect the control boundary without model variability.

## Run It

```
npm run crewai-flow
npm run crewai-flow:test
```

## Expected Result

The test command should print:

```
CrewAI-style flow and crew tests OK
```

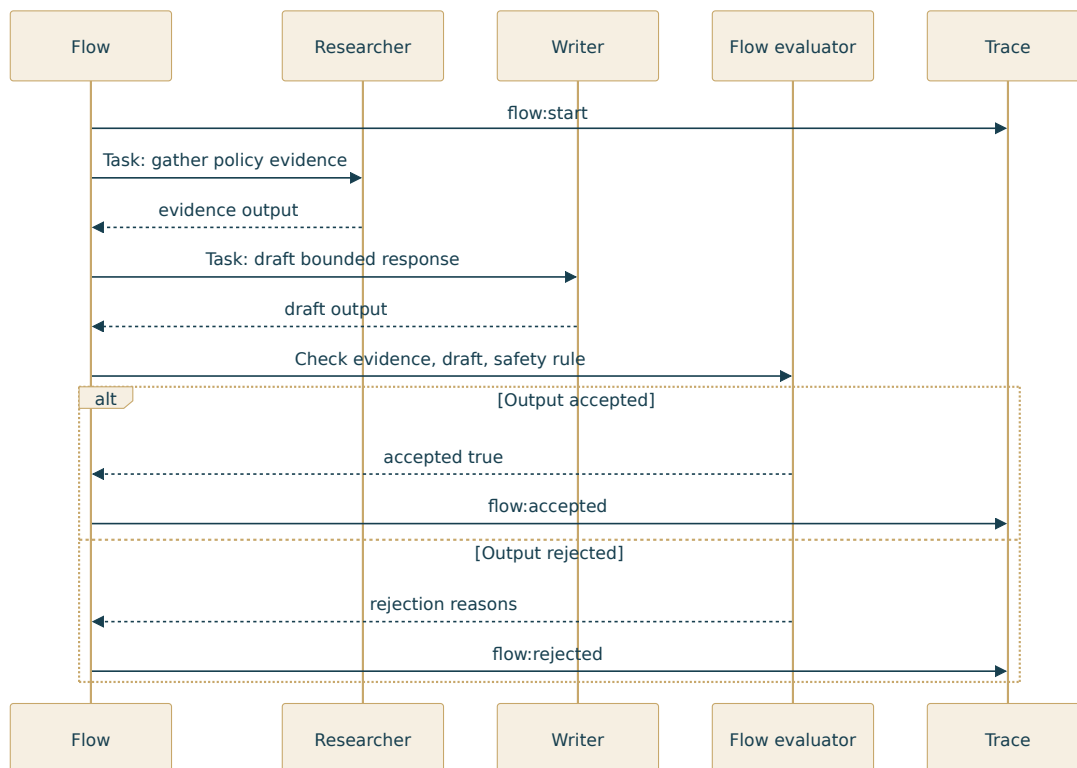
The run should also prove these behavior signals:

- researcher and writer roles produce separate outputs;
- the flow trace records start, crew kickoff, evaluation, and acceptance;
- the flow rejects output that fails the acceptance rule;
- the final state has one accountable owner.

The demo command should include this accepted-flow shape:

```
FlowState(goal='Prepare a refund response', accepted=True, crew_outputs={'evidence':
'policy evidence for Prepare a refund response: refund window is 30 days', 'draft': 'draft
based on policy evidence: offer review, do not promise payment'}, trace=['flow:start',
'flow:crew_kickoff', 'flow:evaluate', 'flow:accepted'], final='Crew output accepted by the
flow.')
```

```
{'status': 'pass'}
```



Use this flow as the lab's acceptance model. The crew performs specialist work, but the flow owns state, evaluation, final acceptance, and trace evidence.

Native CrewAI comparison point:

```

native-framework-examples/crewai-delivery/
download: /downloads/native-crewai-delivery.zip
flow: DeliveryFlow
crew: Planner, Risk reviewer, Test planner
acceptance owner: Flow
eval gate: role outputs present before acceptance

```

## Inspect The Code

Open `crewai-flows-and-crews-pattern/python/flow_crew.py` and find these boundaries:

- `FlowState`: the flow-owned source of truth.
- `Agent`: a role with a specific goal.
- `Task`: a bounded assignment to one role.
- `Crew`: the collaboration unit that executes tasks.
- `evaluate_flow`: the flow-level acceptance gate.

The important design rule is that crew output is not automatically accepted. The flow evaluates it before setting `accepted`.

## Change One Thing



Change the writer output so it omits `do not promise payment`.

Expected rejection:

```
Crew output rejected by the flow.
{'status': 'fail', 'reasons': ['flow did not accept crew output']}
```

The repository test now runs that rejection path with an injected unsafe crew. That keeps the lab deterministic while proving that the flow, not the crew, owns final acceptance.

Then restore the writer output and rerun:

```
npm run crewai-flow:test
```

## Verify

Compare the output with the expected result above before moving to the production extension.

## Lab Review Gate

Before moving on, verify the flow and crew boundary:

| Check                    | Evidence                                                                                  |
|--------------------------|-------------------------------------------------------------------------------------------|
| Flow owns state          | <code>FlowState</code> records request, role outputs, trace, acceptance, and stop reason. |
| Roles are distinct       | Researcher and writer outputs differ by responsibility.                                   |
| Crew output is validated | <code>evaluate_flow</code> rejects output that violates the acceptance rule.              |
| Acceptance has one owner | The flow, not the crew, sets final acceptance.                                            |
| Rejection is observable  | A bad writer output produces a rejected state and reason.                                 |

Record the accepted run, rejected run, role outputs, and flow trace in the lab completion worksheet.

## Production Extension

Before a real CrewAI implementation ships, add:

- typed task inputs and outputs;
- explicit role permission boundaries;
- crew result schemas and validators;
- flow checkpoints and resumability;
- trace records per agent, task, crew, and flow step;

- evaluator cases for disagreement, missing evidence, duplicate work, and bad synthesis.

## Production Bridge

Use this table when adapting the lab to a real CrewAI implementation:

| Lab Concept                | Production Version                                                              |
|----------------------------|---------------------------------------------------------------------------------|
| <code>FlowState</code>     | Durable flow state with tenant, actor, trace ID, checkpoint, and rollback data. |
| <code>Agent</code> role    | Role contract with permissions, tools, model route, and expected output schema. |
| <code>Task</code>          | Typed assignment with input contract, acceptance criteria, timeout, and owner.  |
| <code>Crew</code>          | Bounded collaboration unit with trace spans and cost budget.                    |
| <code>evaluate_flow</code> | Flow-level release gate for evidence, role coverage, synthesis, and safety.     |

The first production milestone is not a more elaborate crew. It is a flow that can reject weak collaboration and explain why.

## Native Framework Extension

After the deterministic lab passes, port one vertical slice into a real CrewAI Flow and Crew. Use [Real Framework Setup Notes](#) for setup guidance and compare your work with the repository example at `native-framework-examples/crewai-delivery/`.

Native porting steps:

1. scaffold a CrewAI project with a Flow;
2. define Flow state for request, role outputs, acceptance, stop reason, and trace ID;
3. create researcher, writer, and reviewer agents only if each role has a distinct contract;
4. define task inputs and expected output shapes;
5. validate crew output before mutating Flow state;
6. add evals for missing evidence, bad synthesis, role disagreement, and rejected output;
7. document rollback for disabling the Crew route while keeping the deterministic Flow path.

The Flow remains the accountable owner:

| Boundary          | Owner       |
|-------------------|-------------|
| state             | Flow        |
| task assignment   | Flow        |
| role behavior     | Crew agents |
| output validation | Flow        |
| final acceptance  | Flow        |

| Boundary | Owner                          |
|----------|--------------------------------|
| rollback | runtime or deployment platform |

Completion standard: the native project proves the same role and acceptance behavior as this lab and links to the [Multi-Agent Delivery Workflow capstone](#). A native CrewAI run is not complete just because the crew returns text; the Flow must preserve separate role outputs and validate them before acceptance.

## Troubleshooting

| Symptom                                    | Likely Cause                                      | Fix                                                                                                |
|--------------------------------------------|---------------------------------------------------|----------------------------------------------------------------------------------------------------|
| crew returns one aggregate answer only     | task outputs are not preserved separately         | Read task-level outputs and copy planner, reviewer, and tester results into Flow state separately. |
| Flow accepts weak or duplicate role output | acceptance is based on completion, not validation | Add Flow-level validators before setting <code>accepted</code> .                                   |
| provider credentials fail                  | CrewAI model provider variables are missing       | Configure the provider environment required by your CrewAI setup.                                  |
| roles overlap heavily                      | agents do not have distinct contracts             | Remove the role or rewrite task contracts until each role changes risk or output quality.          |
| rollback is unclear                        | delegation is embedded in the main path           | Keep a deterministic single-owner fallback path outside the Crew route.                            |

## Cross-Framework Mapping

- In LangGraph, the flow can be a graph while each role maps to a node or subgraph.
- In Mastra AI, the same shape can be modeled as workflows that coordinate agents and tools.
- In AutoGen-style systems, the crew resembles manager-directed specialist agents, but the flow still needs final acceptance.
- In CrewAI, flows provide structured control while crews perform delegated collaborative work.

## Related Chapters

- [CrewAI Flows and Crews](#)
- [Choosing Multi-Agent Topology](#)
- [Task Delegation](#)
- [Observability and Evals](#)
- [Multi-Agent Delivery Workflow Capstone](#)

# From-Scratch Mini-Framework Track

This track builds a tiny educational agent runtime from first principles. It is not a replacement for LangGraph, Mastra AI, AutoGen-style systems, CrewAI, or production workflow engines. It is a way to understand the primitives those systems package.

Download the [lab completion worksheet](#) and [lab production readiness worksheet](#) before starting the track.

Start with [Building a Minimal Agent Runtime](#) for the theory. Then use these labs if you want to implement the primitives yourself.

The maintained TypeScript implementation lives in `minimal-agent-runtime/typescript`. Use it as the reference solution while implementing your own small version in TypeScript or Python.

```
npm run mini-runtime
npm run mini-runtime:test
```

## What You Build

| Lab                                                        | Component                                                      | Main Lesson                                        |
|------------------------------------------------------------|----------------------------------------------------------------|----------------------------------------------------|
| <a href="#">Lab 09 - Minimal Agent Loop</a>                | State, decisions, loop, stop reasons                           | An agent is a controlled loop, not a prompt.       |
| <a href="#">Lab 10 - Tool Registry and Policy Gate</a>     | Tools, policy, approval-required outcomes                      | Tool execution is a software boundary.             |
| <a href="#">Lab 11 - Context, Memory, Trace, and Evals</a> | Context packets, scoped memory, trace events, trajectory evals | Runtime behavior must be inspectable and testable. |

## Learning Contract

By the end of the track, you should be able to explain:

- what the loop owns;
- what belongs in state;
- why model decisions are proposals;
- how tools become narrow capabilities;
- why policy is separate from prompts;
- how context packets are assembled;
- what a trace must record;
- why trajectory evals catch failures final-answer evals miss.

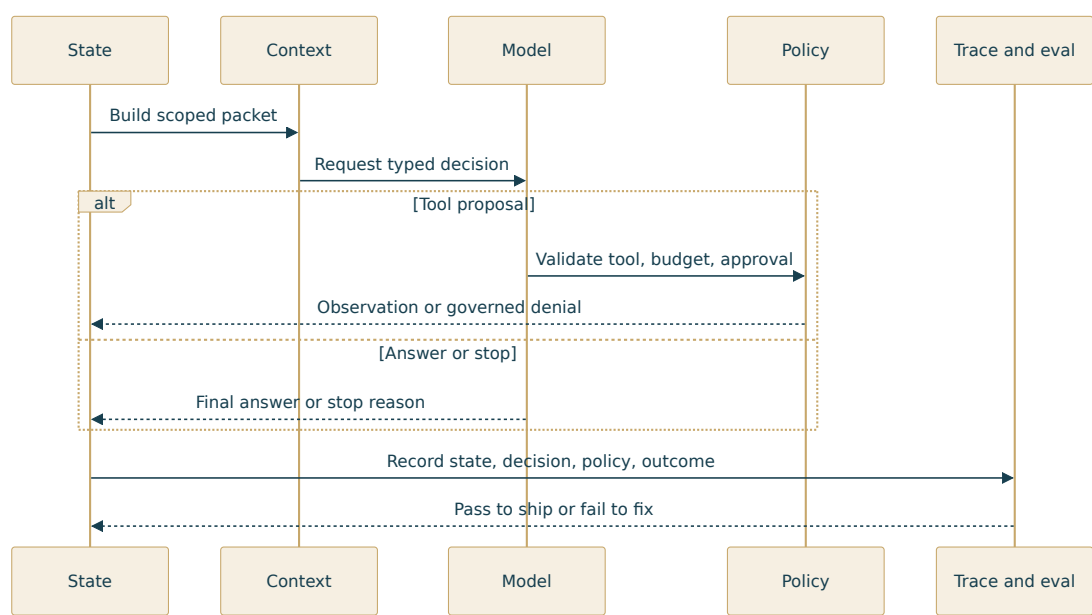
## Implementation Choice

The labs are written so you can implement the runtime in TypeScript or Python. Use TypeScript if you want strong discriminated unions and a direct path to the repository’s existing examples. Use Python if you want a compact teaching implementation before mapping the ideas to LangGraph or CrewAI.

Keep the implementation small. The target is not a general-purpose framework. The target is a runtime you can understand in one sitting.

The TypeScript reference is intentionally dependency-free. The Python version should preserve the same contracts: decisions are typed proposals, policy runs before execution, traces record the path, and evals inspect the trajectory.

Use this model to connect the three labs. The mini-runtime is valuable because every step has an owner: state holds the run, context scopes what the model sees, policy controls action, trace records behavior, and evals judge the trajectory.



## Track Review Gate

Before treating the track as complete, verify the runtime primitives:

| Check               | Evidence                                                                                         |
|---------------------|--------------------------------------------------------------------------------------------------|
| Loop control exists | The runtime has state, decisions, observations, budgets, and stop reasons.                       |
| Tools are governed  | Tool lookup, policy decisions, approval-required outcomes, and unknown-tool refusals are tested. |
| Context is explicit | Context packets and memory reads are scoped rather than hidden in prompt text.                   |
| Trace is reviewable | Runs emit events that explain decisions, tool calls, policy outcomes, and final status.          |

| Check                    | Evidence                                                          |
|--------------------------|-------------------------------------------------------------------|
| Evals inspect trajectory | Tests can fail unsafe paths even when final text looks plausible. |

Record the commands, outputs, tested failure paths, and production gaps in the lab completion worksheet.

## Production Bridge

Use this table when comparing your mini-runtime to a real framework:

| Mini-Runtime Primitive | Production Runtime Question                                                          |
|------------------------|--------------------------------------------------------------------------------------|
| Loop state             | Where is state persisted, migrated, resumed, and deleted?                            |
| Tool registry          | How are tools versioned, permissioned, disabled, and audited?                        |
| Policy gate            | Can policy stop retrieval, memory writes, tools, and final answers before execution? |
| Trace events           | Can operators reconstruct one failed run without raw secrets?                        |
| Trajectory evals       | Which evals block prompt, model, tool, policy, memory, and workflow changes?         |

The track succeeds when it improves framework judgment. It fails if it tempts the team to ship an unoperated framework.

## Production Warning

Do not ship this mini-runtime as a production platform without adding durable execution, persistence, authentication, authorization, concurrency, deployment, observability integrations, retry control, and incident response.

The right production question is not “can we build our own framework?” It is “now that we understand the primitives, which framework or runtime gives us the controls we need?”

## Compare With Frameworks

After the labs, compare each primitive with mature frameworks:

- LangGraph: graph state, nodes, edges, checkpoints, and interrupts.
- Mastra AI: agents, tools, workflows, memory, evals, and runtime packaging.
- AutoGen-style systems: manager/worker roles, messages, and function execution.
- CrewAI: flows, crews, tasks, roles, and tool assignment.

- MCP and A2A: protocol boundaries for tools and agent-to-agent calls.

## Related Chapters

- [Building a Minimal Agent Runtime](#)
- [Agent Harnesses](#)
- [Tool Capability Design](#)
- [Observability and Evals](#)

# Lab 09 - Build a Minimal Agent Loop

Download the [lab completion worksheet](#) and [lab production readiness worksheet](#) before you start.

## Objective

Build the smallest useful runtime primitive: a loop that receives a goal, asks for a typed decision, updates state, and stops for an explicit reason.

## What You Will Use

- Language: TypeScript or Python
- Framework/runtime: from-scratch educational runtime
- Framework-agnostic lesson: an agent is a controlled loop with state, decisions, observations, budgets, and stop reasons.
- Pattern chapters: [What Is An Agent?](#), [Agent Loop](#), [Goals and State](#)
- Theory chapter: [Building a Minimal Agent Runtime](#)

## Exercise Time Budget

These estimates assume dependencies are already installed.

| Exercise                               | Time      | Output                                                          |
|----------------------------------------|-----------|-----------------------------------------------------------------|
| Run the reference baseline             | 10 min    | Passing mini-runtime test output.                               |
| Implement or inspect the loop contract | 20-25 min | Typed state, decision, observation, and stop-reason boundaries. |
| Exercise failure and budget cases      | 10-15 min | Refused, blocked, or budget-exhausted behavior.                 |
| Review trace and stop reasons          | 10-20 min | Notes on runtime-owned limits and caller-facing outcomes.       |

## Setup

Use the maintained TypeScript reference or create your own small file outside production code, such as `scratch/minimal-agent-loop.ts` or `scratch/minimal_agent_loop.py`.

Reference files:

- `minimal-agent-runtime/typescript/src/runtime.ts`
- `minimal-agent-runtime/typescript/src/run_demo.ts`
- `minimal-agent-runtime/typescript/test/runtime.spec.ts`



Run the reference test first:

```
npm run mini-runtime:test
```

This lab does not require a model key. Use a deterministic `decide` function so you can test the runtime without model variability.

## Runtime Contract

Use this shape if you implement the smallest version in TypeScript:

```
type StopReason =
 | "success"
 | "blocked"
 | "budget_exhausted"
 | "invalid_decision"
 | "tool_failure";

type Decision =
 | { kind: "answer"; text: string }
 | { kind: "tool"; name: string; input: unknown }
 | { kind: "ask_human"; question: string }
 | { kind: "stop"; reason: StopReason };

type Observation = {
 kind: "decision" | "tool" | "system";
 summary: string;
};

type AgentState = {
 goal: string;
 steps: number;
 maxSteps: number;
 observations: Observation[];
 stopReason?: StopReason;
};
```

The equivalent Python implementation can use dataclasses, typed dictionaries, or plain dictionaries. Keep the fields the same.

The maintained reference extends this contract with production-facing fields: `runId`, `toolsCalled`, scoped memory, tool definitions, policy decisions, context packets, trace events, and trajectory eval cases. Start with the small contract, then compare your result with `minimal-agent-runtime/typescript/src/runtime.ts`.

## Guided Change

Implement `runAgent(state, decide)`.

The loop should:

1. call `decide(state)`;
2. record an observation for the decision;
3. return with `success` when the decision is an answer;
4. return with the supplied reason when the decision is `stop`;
5. continue for tool decisions, or execute tools through a registry when using the reference runtime;
6. stop with `budget_exhausted` when `steps` reaches `maxSteps`.

## Baseline Run

Use the reference demo:

```
npm run mini-runtime
```

Then inspect the immediate-answer case in `minimal-agent-runtime/typescript/test/runtime.spec.ts`, or use a decision function that answers immediately:

```
const answerImmediately = async (): Promise<Decision> => ({
 kind: "answer",
 text: "done",
});
```

## Expected Result

The demo command should show a policy-read trajectory:

```
{
 "runId": "demo_001",
 "steps": 2,
 "toolsCalled": ["lookup_policy"],
 "answer": "Policy was checked and the draft can be prepared safely.",
 "stopReason": "success"
}
```

It should also include trace events with these types:

```
context_built
decision
policy_decision
```

```
tool_result
stop
```

The eval result should be:

```
{
 "status": "pass",
 "caseId": "demo-policy-read"
}
```

The immediate-answer case should end with:

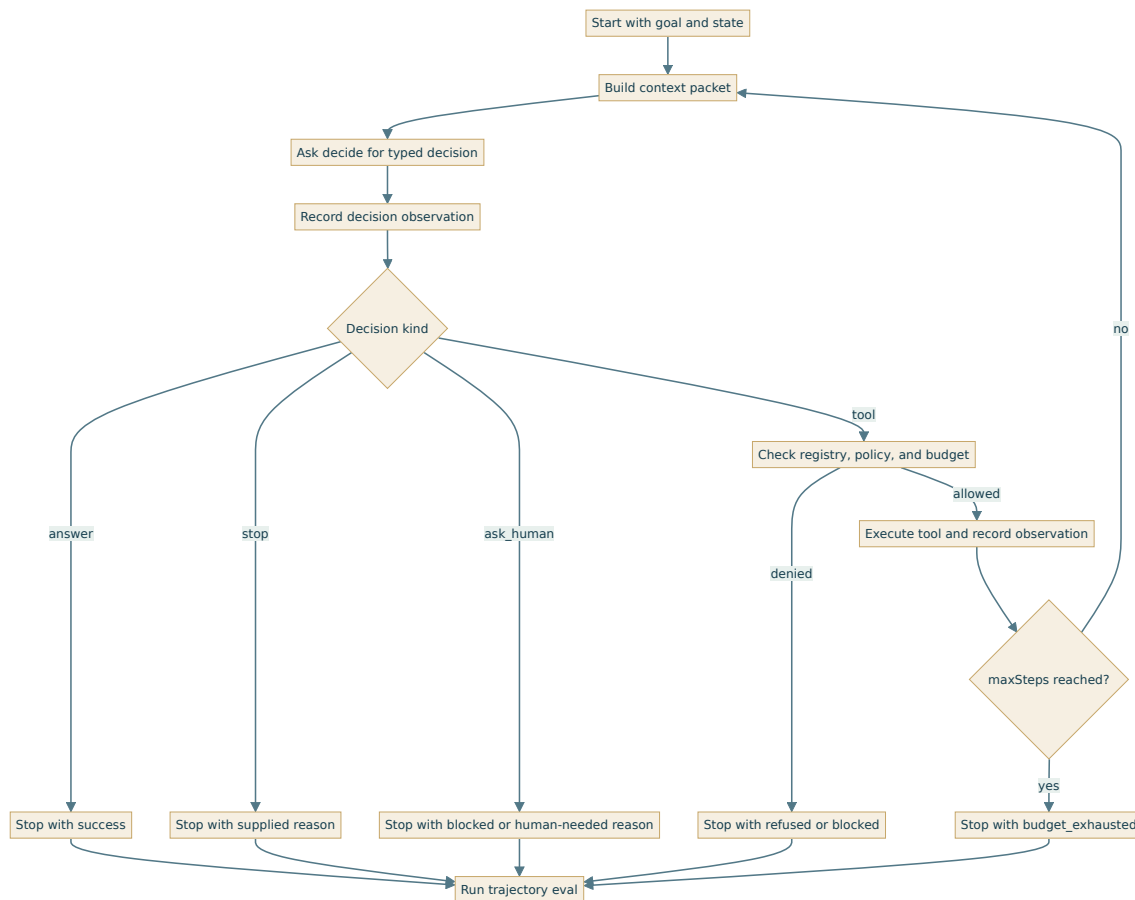
```
stopReason: success
steps: 1
observations: at least one decision observation
```

The repeated-tool case should end with:

```
stopReason: budget_exhausted
steps: maxSteps
```

The reference test also covers:

| Case                        | Expected Signal                                                                                           |
|-----------------------------|-----------------------------------------------------------------------------------------------------------|
| unknown tool                | <code>stopReason: refused</code> ; the unknown tool is not executed.                                      |
| write tool without approval | <code>stopReason: blocked</code> ; <code>send_message</code> is not executed.                             |
| permissive write policy     | final answer can look successful, but trajectory eval fails because <code>send_message</code> was called. |
| scoped memory               | task and project memory are included; user-scope memory is omitted as <code>out_of_scope</code> .         |



Use this loop as the lab's acceptance model. The runtime, not the model, owns budgets, tool authority, stop reasons, observations, and final trajectory evaluation.

## Failure Case

Use a decision function that always asks for a tool:

```
const neverStops = async (): Promise<Decision> => ({
 kind: "tool",
 name: "search",
 input: { query: "keep going" },
});
```

This is the first safety property of an agent runtime: the model cannot create an infinite loop just by continuing to ask.

## Verify

Check these assertions manually or with the reference test:

- immediate answer stops with `success`;
- repeated tool proposals stop with `budget_exhausted`;
- every loop step records an observation;

- the final state contains a stop reason.

The reference test covers these cases with deterministic decisions, so the result is stable across machines.

## Lab Review Gate

Before moving on, verify the loop boundary:

| Check                                        | Evidence                                                                          |
|----------------------------------------------|-----------------------------------------------------------------------------------|
| Decisions are typed                          | The runtime handles answer, tool, ask-human, and stop decisions explicitly.       |
| State changes are visible                    | Goal, step count, observations, and stop reason are recorded.                     |
| Budget stops the loop                        | Repeated tool proposals end with <code>budget_exhausted</code> .                  |
| Success is explicit                          | Immediate answers stop with <code>success</code> .                                |
| The model cannot self-authorize continuation | <code>maxSteps</code> belongs to the runtime, not the decision function.          |
| Trajectory eval catches hidden risk          | A run that sends a message can fail eval even when the final answer says success. |

Record the immediate-answer run, repeated-tool run, observations, and stop reasons in the lab completion worksheet.

## Production Extension

Before this loop can run real work, add:

- structured validation for model-produced decisions;
- tool execution through a registry;
- policy checks before side effects;
- trace events for every decision and stop;
- cancellation and timeout controls;
- durable state if the run can pause or resume.

## Production Bridge

Use this table when adapting the loop to production:

| Lab Concept             | Production Version                                                           |
|-------------------------|------------------------------------------------------------------------------|
| <code>AgentState</code> | Durable run state with actor, tenant, trace ID, budget, and checkpoint data. |

| Lab Concept              | Production Version                                                             |
|--------------------------|--------------------------------------------------------------------------------|
| <code>Decision</code>    | Validated model proposal with schema, policy context, and confidence metadata. |
| <code>Observation</code> | Trace event with timestamp, span ID, status, and redaction class.              |
| <code>maxSteps</code>    | Runtime budget with cost, latency, retry, tool, and delegation limits.         |
| <code>stopReason</code>  | Operator-visible reason tied to evals, dashboards, and incident review.        |

The first production milestone is a loop that always stops for a reason operators can inspect.

## Cross-Framework Mapping

- In LangGraph, the loop is expressed through graph traversal, state updates, and edges.
- In Mastra AI, the loop is packaged inside agent and workflow runtime behavior.
- In AutoGen-style systems, the loop appears as message turns between manager, worker, and tool executors.
- In CrewAI, the loop is shaped by flow execution and task progression.

## Related Chapters

- [Building a Minimal Agent Runtime](#)
- [Agent Loop](#)
- [Goals and State](#)
- [Durable Workflows](#)

# Lab 10 - Build a Tool Registry and Policy Gate

Download the [lab completion worksheet](#) and [lab production readiness worksheet](#) before you start.

## Objective

Extend the mini-runtime with a tool registry and policy gate. The model can propose a tool call, but software decides whether the tool exists, whether the input is acceptable, and whether policy allows execution.

## What You Will Use

- Language: TypeScript or Python
- Framework/runtime: from-scratch educational runtime
- Framework-agnostic lesson: tool descriptions are not permissions; registry and policy are separate runtime boundaries.
- Pattern chapters: [Tool Use](#), [Tool Capability Design](#), [Policy Enforcement](#)
- Previous lab: [Lab 09 - Minimal Agent Loop](#)

## Exercise Time Budget

These estimates assume the Lab 09 loop is already available.

| Exercise                          | Time      | Output                                                            |
|-----------------------------------|-----------|-------------------------------------------------------------------|
| Run the baseline tool contract    | 10 min    | Passing mini-runtime test output.                                 |
| Add registry and policy checks    | 20-25 min | Separate tool lookup, schema handling, and policy decision.       |
| Exercise denial and failure cases | 10-15 min | Unknown-tool, approval-required, and tool-failure signals.        |
| Review production boundary        | 10-20 min | Notes for registry ownership, approval records, and trace fields. |

## Setup

Start from the Lab 09 loop. Add a registry object or map keyed by tool name.

Use deterministic tools. Do not call external systems in this lab.

Reference files:

- `minimal-agent-runtime/typescript/src/runtime.ts`

- `minimal-agent-runtime/typescript/test/runtime.spec.ts`

Run the reference test before editing:

```
npm run mini-runtime:test
```

## Runtime Contract

```
type ToolResult =
 | { status: "ok"; data: unknown }
 | { status: "refused"; reason: string }
 | { status: "error"; reason: string };

type ToolDefinition = {
 name: string;
 description: string;
 sideEffect: "read" | "draft" | "write";
 execute(input: unknown): Promise<ToolResult>;
};

type PolicyDecision =
 | { status: "allow" }
 | { status: "deny"; reason: string }
 | { status: "approval_required"; reason: string };
```

## Guided Change

Add two tools:

```
const tools: Record<string, ToolDefinition> = {
 lookup_policy: {
 name: "lookup_policy",
 description: "Read policy guidance for the current task.",
 sideEffect: "read",
 execute: async input => ({ status: "ok", data: { input, policy: "approval required for
writes" } })),
 },
 draft_message: {
 name: "draft_message",
 description: "Create a draft message for review.",
 sideEffect: "draft",
 execute: async input => ({ status: "ok", data: { draft: String(input) } })),
}
```



```
},
};
```

Then add policy:

```
function authorize(tool: ToolDefinition): PolicyDecision {
 if (tool.sideEffect === "write") {
 return { status: "approval_required", reason: "write_tool_requires_approval" };
 }
 return { status: "allow" };
}
```

Update the loop so tool decisions pass through:

1. registry lookup;
2. policy decision;
3. tool execution only when allowed;
4. observation recording for refused, denied, approval-required, and successful outcomes.

## Baseline Run

Use the reference demo or a decision function that calls `lookup_policy`.

```
npm run mini-runtime
```

## Expected Result

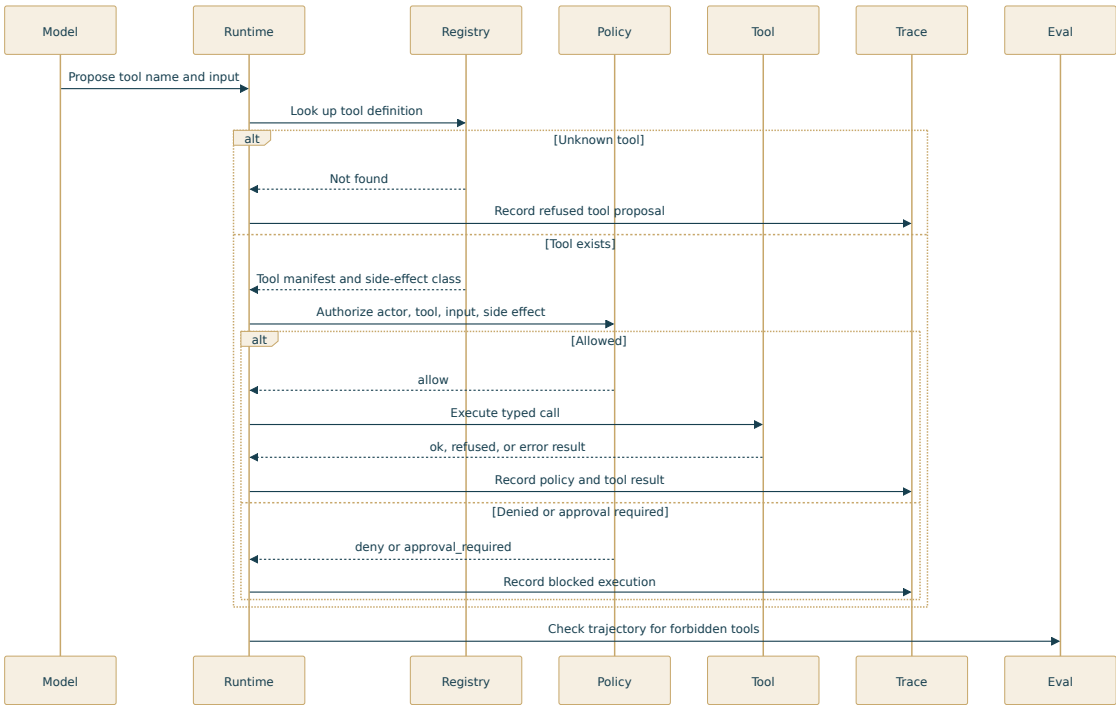
The allowed read-tool path in the reference demo should include:

```
toolsCalled: ["lookup_policy"]
observations include: lookup_policy:allow
observations include: lookup_policy:ok
trace includes: policy_decision
trace includes: tool_result
stopReason: success
```

The reference test covers these refusal and failure signals:

| Case                                         | Expected Signal                                                                  |
|----------------------------------------------|----------------------------------------------------------------------------------|
| Unknown tool<br><code>delete_customer</code> | <code>stopReason: refused</code> ; <code>delete_customer</code> is not executed. |

| Case                                           | Expected Signal                                                                                                    |
|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| Write tool <code>send_message</code>           | <code>stopReason: blocked</code> ; <code>send_message</code> is not executed because approval is required.         |
| Failing read tool<br><code>flaky_lookup</code> | <code>stopReason: tool_failure</code> ; trace includes <code>upstream_timeout</code> .                             |
| Permissive unsafe write policy                 | Final answer can be <code>success</code> , but trajectory eval fails because <code>send_message</code> was called. |



Use this flow as the lab's acceptance model. The model may propose a tool, but the registry, policy gate, trace, and eval decide whether execution is allowed and reviewable.

## Failure Cases

Test these cases:

1. Unknown tool name.
2. Tool with `write` side effect.
3. Tool execution returns `error`.
4. Permissive policy that allows a write tool, so trajectory eval must catch the unsafe path.

The exact stop behavior can vary, but the runtime must not silently execute forbidden or unknown tools.

## Verify

Check these assertions manually or with `npm run mini-runtime:test`:

- unknown tools are not executed;
- policy runs before execution;
- approval-required is represented as a runtime state;
- tool results are structured;
- observations record both allowed and refused paths.

The reference test also proves that `send_message` is blocked before execution when the default policy requires approval for write tools.

## Lab Review Gate

Before moving on, verify the tool boundary:

| Check                          | Evidence                                                                                 |
|--------------------------------|------------------------------------------------------------------------------------------|
| Tool lookup is explicit        | Unknown tool names are refused before execution.                                         |
| Policy runs before execution   | Write tools produce <code>approval_required</code> before any side effect.               |
| Results are structured         | Tool outcomes use <code>ok</code> , <code>refused</code> , or <code>error</code> status. |
| Side-effect class matters      | Read, draft, and write tools can be governed differently.                                |
| Observations preserve the path | Allowed and refused outcomes are visible in runtime observations.                        |

Record the allowed read path, unknown-tool refusal, write-tool approval requirement, and error case in the lab completion worksheet.

## Production Extension

Before using this pattern with real tools, add:

- input schema validation;
- idempotency keys;
- timeout and retry policy;
- actor, tenant, route, and approval context;
- trace IDs for proposed call, policy decision, execution, and result;
- separate policies for read, draft, write, external communication, money movement, memory write, and code execution.

## Production Bridge

Use this table when adapting the registry to production:

| Lab Concept | Production Version                                                         |
|-------------|----------------------------------------------------------------------------|
| Tool map    | Versioned capability registry with owner, permissions, and disable switch. |

| Lab Concept                    | Production Version                                                               |
|--------------------------------|----------------------------------------------------------------------------------|
| <code>ToolDefinition</code>    | Tool manifest with schema, side-effect class, timeout, retry, and audit fields.  |
| <code>authorize</code>         | Policy engine using actor, tenant, resource, risk, budget, and approval context. |
| <code>approval_required</code> | Durable approval request with reviewer, expiry, exact action, and trace link.    |
| Tool observation               | Trace span with proposed input, decision, result, cost, latency, and redaction.  |

The first production milestone is a tool path that can prove why execution was allowed, denied, or paused.

## Cross-Framework Mapping

- In LangGraph, this can be implemented as a tool node guarded by a policy node.
- In Mastra AI, this maps to tool definitions plus workflow or tool-level policy.
- In AutoGen-style systems, this maps to function execution guarded by the manager or runtime.
- In CrewAI, this maps to role-assigned tools plus flow-level constraints.

## Related Chapters

- [Tool Use](#)
- [Tool Capability Design](#)
- [Human Approval Gates](#)
- [Policy Enforcement](#)

# Lab 11 - Add Context, Memory, Trace, and Evals

Download the [lab completion worksheet](#) and [lab production readiness worksheet](#) before you start.

## Objective

Make the mini-runtime inspectable. Add context packets, scoped memory reads, trace events, and a trajectory eval that can fail even when the final answer looks plausible.

## What You Will Use

- Language: TypeScript or Python
- Framework/runtime: from-scratch educational runtime
- Framework-agnostic lesson: runtime behavior must be observable and testable, not only runnable.
- Pattern chapters: [Context Engineering](#), [Working Memory](#), [Observability and Evals](#)
- Previous labs: [Lab 09](#), [Lab 10](#)

## Exercise Time Budget

These estimates assume the Lab 10 runtime is already available.

| Exercise                              | Time      | Output                                                             |
|---------------------------------------|-----------|--------------------------------------------------------------------|
| Run the baseline demo and test        | 10 min    | Passing runtime command output.                                    |
| Add context and scoped memory         | 20 min    | Context packet with included and omitted memory references.        |
| Add trace events and trajectory evals | 20 min    | Trace path plus eval result for risky behavior.                    |
| Exercise one failure case             | 10-15 min | Failed eval or unsafe trajectory signal.                           |
| Complete production review            | 10-25 min | Notes for memory governance, trace redaction, and incident replay. |

## Setup

Start from the Lab 10 runtime. Keep tools deterministic and small.

Reference files:

- `minimal-agent-runtime/typescript/src/runtime.ts`

- `minimal-agent-runtime/typescript/src/run_demo.ts`
- `minimal-agent-runtime/typescript/test/runtime.spec.ts`

Run the reference demo and test:

```
npm run mini-runtime
npm run mini-runtime:test
```

Add a memory fixture:

```
const memory = [
 { id: "mem_1", scope: "project", text: "Write tools require approval." },
 { id: "mem_2", scope: "task", text: "The current task may use read tools only." },
];
```

## Runtime Contract

```
type ContextPacket = {
 runId: string;
 goal: string;
 stateSummary: string;
 observations: Array<{ summary: string }>;
 toolsDisclosed: string[];
 memoryRefs: string[];
 omittedRefs: Array<{ ref: string; reason: string }>;
};
```

```
type TraceEvent = {
 runId: string;
 step: number;
 type:
 | "context_built"
 | "decision"
 | "policy_decision"
 | "tool_result"
 | "stop";
 data: unknown;
};
```

```
type EvalCase = {
 caseId: string;
 input: string;
 expected: {
```

```

toolsCalled?: string[];
toolsNotCalled?: string[];
stopReason: string;
};
};

```

## Guided Change

Add `buildContext(state)` so every model decision receives a deliberate packet:

- active goal;
- compact state summary;
- recent observations;
- disclosed tools;
- selected memory refs;
- omitted memory refs with reasons.

Add `recordTrace(event)` and emit trace events for:

1. context built;
2. decision proposed;
3. policy decision;
4. tool result;
5. stop reason.

## Baseline Run

Run a case where the agent calls a read tool and then answers. The reference demo does this with `lookup_policy`.

## Expected Result

The demo command should show the scoped context packet. The first `context_built` event should include:

```

{
 "toolsDisclosed": ["draft_message", "lookup_policy", "send_message"],
 "memoryRefs": ["mem_1", "mem_2"],
 "omittedRefs": [
 { "ref": "mem_3", "reason": "out_of_scope" }
]
}

```

The read-tool path should include a trace like this:

```

context_built
decision
policy_decision
tool_result
context_built
decision
stop

```

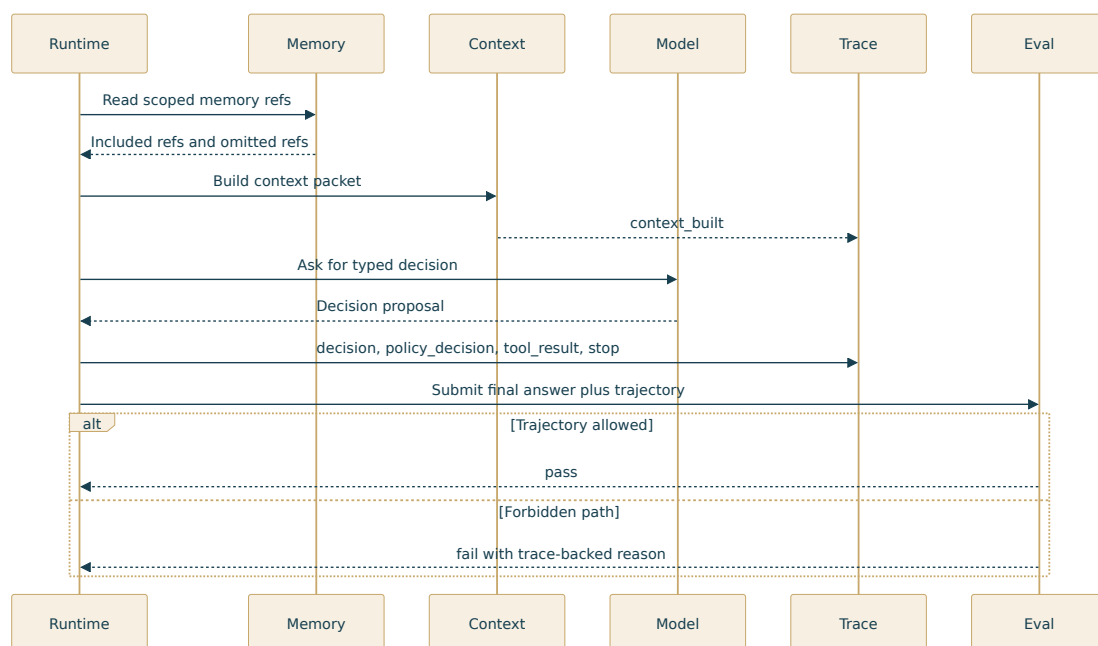
The exact order can vary if your loop stops immediately after a tool, but the trace must show enough to reconstruct the path.

The unsafe trajectory case should produce:

```

final answer: done
stopReason: success
trajectory eval: fail
reason: forbidden tool was called: send_message

```



Use this flow as the lab’s acceptance model. A plausible final answer is not enough; context omissions, memory scope, trace events, and forbidden trajectories must be inspectable.

## Failure Case

Create an eval where the final answer says “done”, but the runtime called a forbidden write tool.

This is why final-answer-only evals are too weak for agentic systems.

## Verify



Check these assertions manually or with `npm run mini-runtime:test`:

- every run has a trace;
- every stop has a stop reason;
- context records included and omitted memory;
- evals can check tools called and tools not called;
- a forbidden trajectory fails even when final text looks acceptable.

The reference test includes an intentionally unsafe run where the final answer is `done`, but the trajectory eval fails because a forbidden write tool was called.

## Lab Review Gate

Before moving on, verify the inspectability boundary:

| Check                       | Evidence                                                                                       |
|-----------------------------|------------------------------------------------------------------------------------------------|
| Context is deliberate       | The context packet lists goal, state summary, observations, tools, memory refs, and omissions. |
| Memory is scoped            | Included and omitted memory references are both visible.                                       |
| Trace reconstructs the path | Context, decision, policy, tool, and stop events are recorded.                                 |
| Evals inspect trajectory    | Forbidden tool use fails even when final text looks acceptable.                                |
| Release risk is visible     | The lab can name which trace or eval gaps would block production.                              |

Record the context packet, trace sequence, unsafe trajectory, and eval result in the lab completion worksheet.

## Production Extension

Before production, add:

- redaction before trace storage;
- retention and deletion policy for memory and traces;
- eval fixtures versioned with prompts, tools, models, and policies;
- incident-to-eval workflow;
- dashboards for stop reasons, tool errors, policy denials, cost, and latency;
- release gates that block risky changes when trajectory evals fail.

## Production Bridge

Use this table when adapting inspectability to production:

| Lab Concept                | Production Version                                                                             |
|----------------------------|------------------------------------------------------------------------------------------------|
| <code>ContextPacket</code> | Versioned context contract with evidence refs, memory policy, omissions, and token budget.     |
| Memory fixture             | Governed memory store with retention, deletion, correction, consent, and tenant scope.         |
| <code>TraceEvent</code>    | Correlated trace span with run ID, span ID, parent span, status, cost, latency, and redaction. |
| <code>EvalCase</code>      | Release fixture with owner, severity, threshold, version set, and incident link.               |
| Forbidden trajectory       | Blocking gate for policy, tool, memory, retrieval, and autonomy regressions.                   |

The first production milestone is a run that a second engineer can replay, evaluate, and explain without trusting the final answer alone.

### Cross-Framework Mapping

- In LangGraph, context and memory are state inputs, while traces can follow node transitions and checkpoints.
- In Mastra AI, memory, evals, and observability are runtime-level capabilities that should still expose product-owned policy.
- In AutoGen-style systems, message history must be converted into structured trace and eval data.
- In CrewAI, flow and task records need enough structure to evaluate role behavior and final synthesis.

### Related Chapters

- [Context Engineering](#)
- [Working Memory](#)
- [Observability and Evals](#)
- [Production Evaluation Feedback Loops](#)

# Lab 12 - Model State Graphs, Checkpoints, and Interrupts

Download the [Lab 12 state graph guided exercise worksheet](#), [lab completion worksheet](#), and [lab production readiness worksheet](#) before you start.

## Objective

Use a LangGraph-style Python state graph to make state, nodes, edges, checkpoints, interrupts, and resume behavior explicit.

## What You Will Use

- Language: Python
- Framework/runtime: LangGraph-style state graph
- Framework-agnostic lesson: graph execution is valuable when state transitions, branching, pause/resume, and node-level observability matter.
- Official terminology checked: LangGraph graph state, nodes, edges, checkpoints, and interrupts.
- Pattern chapters: [Agent Loop](#), [Goals and State](#), [Durable Workflows](#)
- Source files:
  - `langgraph-state-graph-pattern/python/state_graph.py`
  - `langgraph-state-graph-pattern/python/test_state_graph.py`
- Download: [langgraph-state-graph.zip](#)

## Exercise Time Budget

These estimates assume dependencies are already installed.

| Exercise                                    | Time      | Output                                                               |
|---------------------------------------------|-----------|----------------------------------------------------------------------|
| Setup and baseline graph run                | 10 min    | Demo and test output.                                                |
| Inspect state, nodes, and checkpoints       | 20 min    | Notes on state schema, node boundaries, and checkpoint placement.    |
| Exercise interrupt and resume               | 20 min    | Interrupted trace and resumed trace evidence.                        |
| Review checkpoint failure and replay safety | 20-25 min | Failing assertion or replay-risk note.                               |
| Compare native graph and production bridge  | 10-30 min | Mapping to native graph, durable checkpointer, and approval payload. |

## Setup

From the repository root:

```
npm install
```

This lab is deterministic and does not require a model key. It models the LangGraph execution contract without external dependencies so the state behavior is easy to inspect.

## Run It

```
npm run langgraph-state
npm run langgraph-state:test
```

## Expected Result

The test command should print:

```
LangGraph-style state graph tests OK
```

The first run should stop at human approval:

```
stop_reason: human_interrupt
trace includes checkpoint:review
trace includes interrupt:approval_required
```

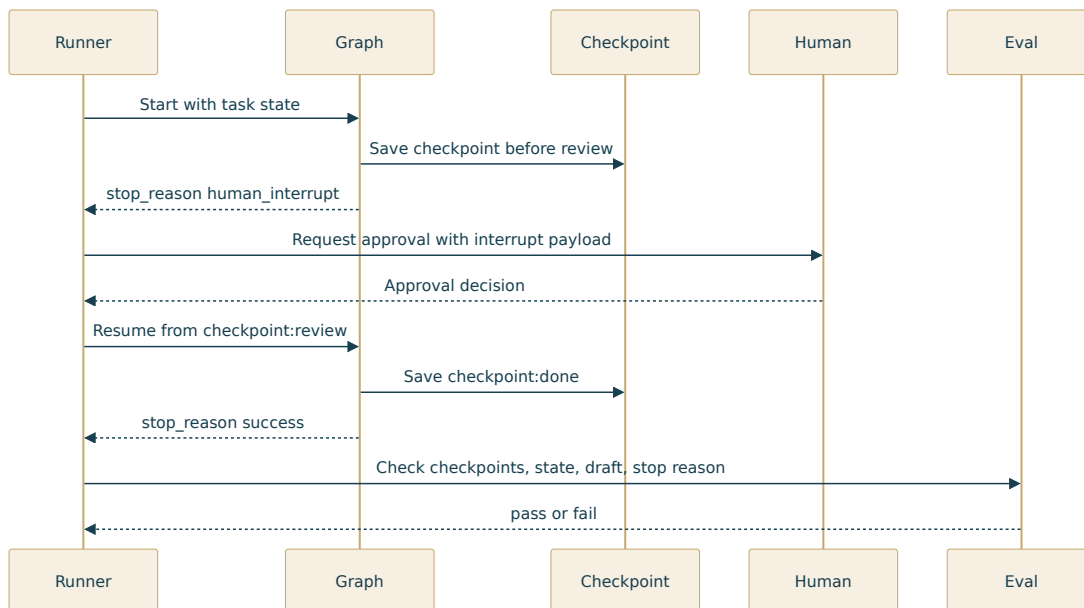
The resumed run should start from the review node with approval:

```
stop_reason: success
trace: checkpoint:review -> node:review -> checkpoint:done -> graph:done
```

The demo command should print two graph runs. Use these fields as the quick check:

```
first.state.stop_reason: human_interrupt
first.state.interrupted: True
first.trace: ... checkpoint:review, node:review, interrupt:approval_required
first.eval.status: pass

resumed.state.stop_reason: success
resumed.state.approved: True
resumed.state.draft: Draft a refund response for human review; do not promise payment.
resumed.trace: checkpoint:review, node:review, checkpoint:done, graph:done
resumed.eval.status: pass
```



Use this flow as the lab's acceptance model. A correct run must prove where it paused, what state survived, which approval resumed it, and why the graph stopped.

Native LangGraph comparison point:

```

native-framework-examples/langgraph-refund/
download: /downloads/native-langgraph-refund.zip
graph: StateGraph
checkpointer: InMemorySaver for local development
interrupt: finance approval
eval gate: draft stops before money movement

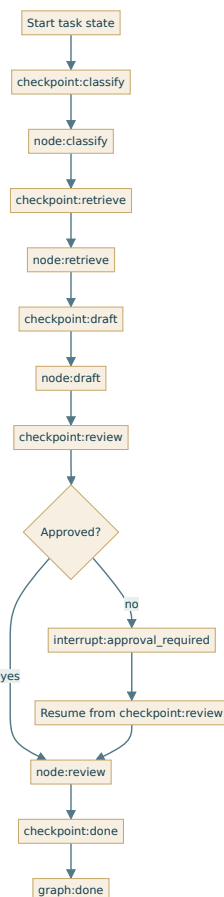
```

## Guided Exercises

Use these exercises to prove that pause, resume, and replay are first-class graph behavior.

| Exercise              | Time   | What To Do                                                                                      | Evidence To Save                                                                                                            |
|-----------------------|--------|-------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Interrupted run trace | 10 min | Run <code>npm run langgraph-state</code> .                                                      | <code>checkpoint:review</code> , <code>interrupt:approval_required</code> , and <code>stop_reason: human_interrupt</code> . |
| Resumed run trace     | 10 min | Inspect the resumed run after approval.                                                         | Trace starts at <code>checkpoint:review</code> and ends at <code>graph:done</code> .                                        |
| Checkpoint failure    | 15 min | Temporarily remove <code>checkpoint(run, node)</code> before node execution and rerun the test. | The failing assertion or eval reason.                                                                                       |

| Exercise             | Time   | What To Do                                                                       | Evidence To Save                                                     |
|----------------------|--------|----------------------------------------------------------------------------------|----------------------------------------------------------------------|
| Replay safety review | 15 min | Decide which nodes would be unsafe to replay in production.                      | Node name, side effect, idempotency key, and checkpoint requirement. |
| Native comparison    | 20 min | Compare this lab with <code>native-framework-examples/langgraph-refund/</code> . | State schema, checkpointer, interrupt, and eval mapping.             |



## Checkpoint And Resume Failure Exercise

The lab is not complete until you can explain what fails when checkpoints disappear. Remove the checkpoint call before node execution, run the test, and then restore it:

```
npm run langgraph-state:test
```

Record the failure in the worksheet:

| Review Question                | Expected Answer                                                                  |
|--------------------------------|----------------------------------------------------------------------------------|
| Where did the first run pause? | <code>checkpoint:review</code> before <code>interrupt:approval_required</code> . |
| What survived resume?          | Intent, evidence, draft, approval state, and stop reason context.                |

| Review Question                   | Expected Answer                                                                      |
|-----------------------------------|--------------------------------------------------------------------------------------|
| What should never replay blindly? | Any node that reads external state, writes state, sends messages, or moves money.    |
| What proves the resumed path?     | Trace starts at <code>checkpoint:review</code> and reaches <code>graph:done</code> . |

## Inspect The Code

Open `langgraph-state-graph-pattern/python/state_graph.py` and find these boundaries:

- `GraphState`: shared graph state.
- `NODES`: node functions that update state.
- `EDGES`: control flow from one node to the next.
- `checkpoint`: state snapshot before node execution.
- `review`: interrupt point when human approval is missing.
- `run_graph(..., resume_from=...)`: resume path from a saved state.
- `evaluate_graph`: trajectory eval over checkpoints, state, and stop reason.

## Baseline Run

This is the core state-graph lesson: the runtime should not have to replay classification, retrieval, and drafting when a saved checkpoint is enough.

## Change One Thing

Remove the `checkpoint(run, node)` call before node execution.

Expected failure: the test should fail because the graph can no longer prove where it paused and resumed.

Restore the checkpoint call and rerun:

```
npm run langgraph-state:test
```

## Verify

Check that:

- every node boundary can create a checkpoint;
- interrupt state is explicit;
- resume starts from a known node;
- retrieved evidence and draft state survive resume;
- evals inspect trajectory and state, not only final text.

- success without a draft fails evaluation.

## Lab Review Gate

Before moving on, verify the graph boundary:

| Check                              | Evidence                                                              |
|------------------------------------|-----------------------------------------------------------------------|
| State is explicit                  | <code>GraphState</code> carries the data each node needs and mutates. |
| Node boundaries are visible        | Each node produces traceable state changes.                           |
| Checkpoints prove pause and resume | <code>checkpoint:review</code> exists before interrupt and resume.    |
| Interrupt is controlled            | Human approval is a structured stop, not a hidden prompt convention.  |
| Eval checks trajectory             | The evaluator checks checkpoints, state, and stop reason.             |

Record the interrupted run, resumed run, checkpoint, approval payload, and eval result in the lab completion worksheet.

## Production Extension

Before using a real LangGraph implementation in production, add:

- durable checkpointer storage;
- thread IDs for independent user/task state;
- idempotency around node side effects;
- typed state schemas and reducers;
- interrupt payloads for human approval;
- replay tests for failed, interrupted, and resumed runs;
- trace export for node inputs, outputs, errors, and stop reasons.

## Production Bridge

Use this table when adapting the graph to production:

| Lab Concept             | Production Version                                                           |
|-------------------------|------------------------------------------------------------------------------|
| <code>GraphState</code> | Versioned state schema with migration, tenant isolation, and deletion rules. |
| Node function           | Idempotent step with typed input, output, policy check, and trace span.      |
| Checkpoint              | Durable checkpointer keyed by thread ID, run ID, tenant, and version set.    |
| Interrupt               | Approval request with exact action, reviewer role, expiry, and resume token. |



| Lab Concept                 | Production Version                                                               |
|-----------------------------|----------------------------------------------------------------------------------|
| <code>evaluate_graph</code> | Release gate over route, checkpoints, state diff, side effects, and stop reason. |

The first production milestone is a graph run that can pause, resume, and prove it did not replay unsafe work.

## Native Framework Extension

After the deterministic lab passes, port one vertical slice into a real LangGraph app. Use [Real Framework Setup Notes](#) for setup guidance and compare your work with the repository example at `native-framework-examples/langgraph-refund/`.

Native porting steps:

- 1. define the graph state schema from `GraphState`;
- 2. convert deterministic functions into node functions;
- 3. express route decisions as conditional edges;
- 4. compile the graph with a checkpointer;
- 5. use an interrupt for human approval instead of a local stop flag;
- 6. add a thread ID strategy that prevents cross-tenant state access;
- 7. add evals over final output, state diff, selected route, checkpoints, and stop reason.

Do not consider the native port complete until it proves:

| Requirement                               | Evidence                                      |
|-------------------------------------------|-----------------------------------------------|
| interrupt can pause safely                | saved checkpoint and approval payload         |
| resume does not replay prior side effects | idempotency key or side-effect record         |
| state is inspectable                      | serialized state and migration note           |
| trace is useful                           | node, tool, policy, interrupt, and eval spans |
| rollback is possible                      | disable one graph route or tool               |

This extension maps directly to the [Support Refund Agent capstone](#) and [Research RAG Agent capstone](#). The native example is intentionally small: it proves state, interrupt, resume, and eval behavior before adding real order tools or model calls.

## Troubleshooting

| Symptom                      | Likely Cause                                   | Fix                                                                         |
|------------------------------|------------------------------------------------|-----------------------------------------------------------------------------|
| resume restarts earlier work | graph lacks a checkpointer or stable thread ID | Compile with a checkpointer and pass a tenant-safe <code>thread_id</code> . |

| Symptom                           | Likely Cause                       | Fix                                                                                              |
|-----------------------------------|------------------------------------|--------------------------------------------------------------------------------------------------|
| side effects repeat after resume  | side-effect node is not idempotent | Move side effects behind idempotency keys or approval records.                                   |
| approval payload is hard to audit | interrupt data is unstructured     | Store ticket ID, draft ID, approver role, expiry, and requested action in the interrupt payload. |
| state migrations are unclear      | state schema is only implicit      | Version the state schema and document migration rules before production.                         |
| trace only shows final output     | node spans are missing             | Emit node, tool, policy, interrupt, resume, and eval events.                                     |

## Cross-Framework Mapping

- In LangGraph, this maps directly to graph state, nodes, edges, checkpoints, and interrupts.
- In Mastra AI, the same responsibility may be represented as workflow steps, memory, and runtime traces.
- In AutoGen-style systems, checkpointing usually needs explicit transcript and task state outside the conversation.
- In CrewAI, flow state provides the equivalent durable control boundary while crews perform delegated work.

## Related Chapters

- [Goals and State](#)
- [Agent Loop](#)
- [Durable Workflows](#)
- [Observability and Evals](#)
- [Support Refund Agent Capstone](#)
- [Research RAG Agent Capstone](#)

# Lab 13 - Evaluate Multi-Agent Transcripts

Download the [lab completion worksheet](#) and [lab production readiness worksheet](#) before you start.

## Objective

Use an AutoGen-style team conversation to make agents, team turns, structured messages, transcript ownership, termination, and transcript evals explicit.

## What You Will Use

- Language: TypeScript
- Framework/runtime: AutoGen-style AgentChat team transcript
- Framework-agnostic lesson: multi-agent collaboration needs a reviewable transcript and evals that check who said what, in what order, and why the team stopped.
- Official terminology checked: AutoGen AgentChat agents, teams, messages, and observable team behavior.
- Pattern chapters: [Supervisor / Worker](#), [Task Delegation](#), [Observability and Evals](#)
- Source files:
  - `autogen-transcript-pattern/typescript/src/team_transcript.ts`
  - `autogen-transcript-pattern/typescript/src/run_demo.ts`
  - `autogen-transcript-pattern/typescript/test/team_transcript.spec.ts`
- Download: [autogen-transcript.zip](#)

## Exercise Time Budget

These estimates assume dependencies are already installed.

| Exercise                                 | Time      | Output                                                                    |
|------------------------------------------|-----------|---------------------------------------------------------------------------|
| Setup and baseline transcript run        | 10 min    | Demo and test output.                                                     |
| Inspect transcript schema and turn order | 15 min    | Notes on sender, recipient, type, task ID, turn, and stop reason.         |
| Exercise transcript failure cases        | 20 min    | Missing-role, wrong-order, or task-ID failure signal.                     |
| Compare native team behavior             | 10-15 min | Mapping to AutoGen-style agents, team, messages, and termination.         |
| Complete production transcript gate      | 10-30 min | Notes for redaction, replay, retention, role permissions, and eval gates. |

## Setup

From the repository root:

```
npm install
```

This lab is deterministic and does not require a model key. It models the team transcript contract without live model calls.

## Run It

```
npm run autogen-transcript
npm run autogen-transcript:test
```

## Expected Result

The test command should print:

```
AutoGen-style transcript tests OK
```

The transcript should include this role flow:

```
manager -> researcher: task
researcher -> reviewer: evidence
reviewer -> manager: review
manager -> team: final
```

The final state should include:

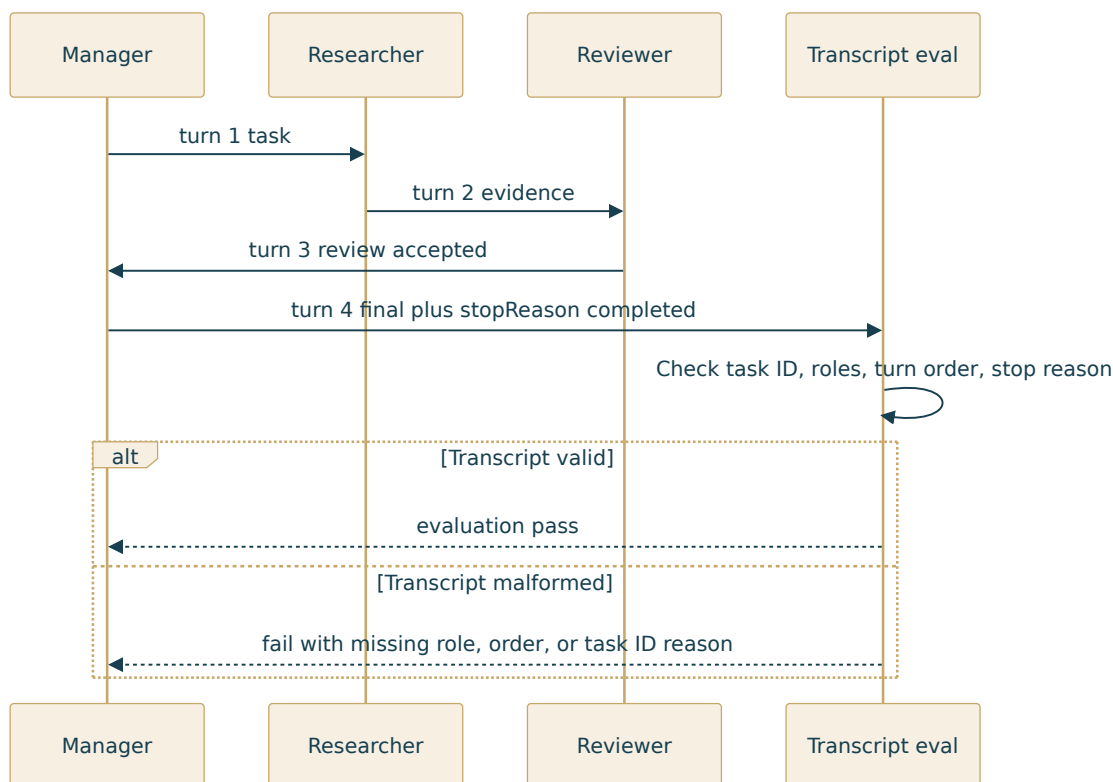
```
stopReason: completed
evaluation: pass
```

The demo command should include this four-message transcript shape:

```
turn 1: manager -> researcher, type: task
turn 2: researcher -> reviewer, type: evidence
turn 3: reviewer -> manager, type: review, accepted: true
turn 4: manager -> team, type: final
```

Every message should carry the same task ID:

```
taskId: autogen_style_001
```



Use this flow as the lab's acceptance model. The final answer is not enough; the transcript must prove evidence, review, final ownership, and the reason the team stopped.

The repository test also checks three malformed transcript cases:

| Case                        | Expected Failure                                                                      |
|-----------------------------|---------------------------------------------------------------------------------------|
| researcher evidence removed | <code>missing role: researcher</code> and <code>missing message type: evidence</code> |
| final message before review | <code>review must precede final</code>                                                |
| mismatched task ID          | <code>message task IDs do not match team task</code>                                  |

Native AutoGen comparison point:

```

native-framework-examples/autogen-delivery/
download: /downloads/native-autogen-delivery.zip
team: RoundRobinGroupChat
agents: delivery_manager, delivery_planner, risk_reviewer, test_planner
termination: TextMentionTermination("ACCEPTED") OR MaxMessageTermination(8)
eval gate: delivery_transcript_acceptance

```

## Inspect The Code

Open `autogen-transcript-pattern/typescript/src/team_transcript.ts` and find these boundaries:

- `TeamMessage`: structured transcript event.

- `Agent`: named participant with a response contract.
- `createTeam`: manager, researcher, and reviewer roles.
- `runTeam`: fixed team turn sequence and termination.
- `evaluateTranscript`: transcript-level acceptance criteria.

The transcript is the core artifact. It should show task assignment, evidence, review, final synthesis, and stop reason.

## Baseline Run

Use the expected result above as the baseline transcript contract.

## Change One Thing

Remove the researcher turn or change its message type from `evidence` to `final`.

Expected failure: the transcript eval should fail because the team can no longer prove that evidence preceded review and final synthesis.

Restore the researcher turn and rerun:

```
npm run autogen-transcript:test
```

## Verify

Check that:

- every message has a sender, recipient, type, task ID, and turn number;
- all required roles appear in the transcript;
- turn numbers are sequential;
- task IDs match the team task;
- evidence precedes review, and review precedes final;
- final output does not bypass human review;
- the stop reason is explicit;
- transcript evals fail on missing roles or malformed message flow.

## Lab Review Gate

Before moving on, verify the transcript boundary:

| Check                   | Evidence                                                          |
|-------------------------|-------------------------------------------------------------------|
| Messages are structured | Sender, recipient, type, task ID, and turn number are present.    |
| Roles are required      | Manager, researcher, and reviewer all appear before final output. |

| Check                        | Evidence                                                                    |
|------------------------------|-----------------------------------------------------------------------------|
| Turn order is enforced       | Evidence precedes review, and review precedes final synthesis.              |
| Termination is explicit      | The team stops with <code>completed</code> , not an ambiguous last message. |
| Eval protects the transcript | Missing roles or malformed flow fail the transcript eval.                   |

Record the transcript, stop reason, failed transcript case, and eval result in the lab completion worksheet.

## Production Extension

Before using a real AutoGen implementation in production, add:

- message schemas for every team event;
- termination conditions and max-turn budgets;
- tool-call and human-input records in the transcript;
- redaction before transcript storage;
- transcript replay and regression evals;
- per-agent role contracts and permission boundaries;
- migration notes if adopting Microsoft Agent Framework for new projects.

## Production Bridge

Use this table when adapting transcript evals to production:

| Lab Concept              | Production Version                                                                   |
|--------------------------|--------------------------------------------------------------------------------------|
| <code>TeamMessage</code> | Normalized event schema with role, task, turn, tool, approval, and redaction fields. |
| Agent role               | Contract with allowed tools, authority, model route, and output schema.              |
| Fixed team sequence      | Termination policy plus max-turn budget and escalation rule.                         |
| Transcript eval          | Release gate for role order, tool permission, final owner, stop reason, and safety.  |
| Raw conversation         | Redacted transcript store with replay, retention, deletion, and incident links.      |

The first production milestone is a transcript that can prove who owned the final answer and why the team stopped.

## Native Framework Extension

After the deterministic lab passes, port one vertical slice into a real AutoGen AgentChat team. Use [Real Framework Setup Notes](#) for setup guidance and compare your work with the repository example at [native-framework-examples/autogen-delivery/](#).

Native porting steps:

1. define AssistantAgent roles that match the deterministic manager, researcher, and reviewer contracts;
2. define team termination rules and max-turn budgets;
3. persist structured message events outside the raw chat transcript;
4. wrap tools so permissions and side effects are enforced outside model text;
5. redact transcript content before storage;
6. replay transcripts through evals that check role order, tool calls, final owner, and stop reason;
7. document rollback for disabling multi-agent delegation.

Transcript evals should check more than final text:

| Check                   | Failure It Catches                     |
|-------------------------|----------------------------------------|
| required roles present  | fake specialization or skipped review  |
| turn order valid        | final answer before evidence or review |
| stop reason explicit    | endless or ambiguous team behavior     |
| tool calls permissioned | worker uses a tool outside its role    |
| final owner present     | no accountable acceptance boundary     |

Completion standard: the native project proves the same transcript guarantees as this lab and links to the [Multi-Agent Delivery Workflow capstone](#). A native AutoGen team is not complete just because agents exchange messages; the system must normalize the transcript, preserve stop reason, and replay it through evals.

## Troubleshooting

| Symptom                               | Likely Cause                                   | Fix                                                                                                      |
|---------------------------------------|------------------------------------------------|----------------------------------------------------------------------------------------------------------|
| team runs until the max-message limit | termination phrase is missing or too ambiguous | Add an explicit <code>TextMentionTermination</code> phrase and require only the final owner to use it.   |
| eval cannot prove role order          | raw chat is not normalized                     | Store sender, recipient, message type, task ID, turn number, and stop reason outside the raw transcript. |
| reviewer or tester role is skipped    | team composition or turn policy is too loose   | Use a bounded team pattern and eval required roles before accepting output.                              |



| Symptom                               | Likely Cause                                  | Fix                                                                                               |
|---------------------------------------|-----------------------------------------------|---------------------------------------------------------------------------------------------------|
| provider import or model client fails | optional AutoGen extension package is missing | Install <code>autogen-ext[openai]</code> or the provider extension used by the project.           |
| new project concern                   | AutoGen maintenance status is a risk          | Compare AutoGen with Microsoft Agent Framework before committing long-term platform architecture. |

## Cross-Framework Mapping

- In LangGraph, the same collaboration can be represented as graph nodes or subgraphs with explicit state.
- In Mastra AI, a workflow can coordinate agents and tools while traces capture the path.
- In AutoGen-style systems, the team transcript is the reviewable execution artifact.
- In CrewAI, crews and role tasks produce similar collaborative outputs, while flows own acceptance.

## Related Chapters

- [Supervisor / Worker](#)
- [Task Delegation](#)
- [Choosing Multi-Agent Topology](#)
- [Production Evaluation Feedback Loops](#)
- [Multi-Agent Delivery Workflow Capstone](#)

# Vertical Slice Examples

Patterns become useful when they are composed into a working system. A vertical slice is a small end-to-end design that shows the goal, agent loop, tools, state, policy, observability, evals, and runtime behavior together.

Download the [lab completion worksheet](#) and [lab production readiness worksheet](#) when turning a slice into an implementation plan.

The examples in this chapter are not full products. They are slices. Each one should be small enough to review in one sitting and concrete enough to expose architecture decisions.

Read this after [Pattern Composition Playbook](#), [Production Runtime Overview](#), and [Observability and Evals](#). Those chapters explain the boundaries; this chapter shows what those boundaries look like when several patterns work together for one task.

Run the deterministic capstone runtime when you want executable evidence for the same shapes:

```
npm run capstones:demo
npm run capstones:test
```

Expected output:

```
support-refund-agent: pass
 stop: draft_ready
 trace events: 7
research-rag-agent: pass
 stop: answered_with_citation
 trace events: 6
multi-agent-delivery-workflow: pass
 stop: accepted_after_review
 trace events: 4
Capstone project tests OK
```

The code lives in `capstone-projects-runtime/typescript/src/capstones.ts`. Treat it as the runnable evidence layer for these slices: each run returns state, trace events, eval results, and rollback actions.

Download the [captured lab](#) and [capstone command output examples](#) when you need a compact model for saving capstone terminal output, trace snapshots, eval snapshots, and production questions.

## How To Read A Slice

Use the same checklist for every example:

1. What user or system goal starts the run?
2. What patterns are composed?
3. What state must survive between steps?
4. What tools can the agent call, and under which scopes?
5. What requires approval?
6. What trace events prove what happened?
7. What evals catch regressions?
8. What failure mode would make this unsafe in production?

If a slice cannot answer those questions, it is still a demo.

## Slice Review Gate

Use this gate before turning any slice into a capstone or product backlog:

| Check                           | Evidence                                                                  |
|---------------------------------|---------------------------------------------------------------------------|
| Goal is bounded                 | One user or system goal starts the run.                                   |
| Pattern composition is explicit | Each major concern maps to a named pattern chapter.                       |
| Authority is constrained        | Tools, data, memory, approvals, and side effects have owners and scopes.  |
| State is recoverable            | The slice names what must persist, replay, resume, or be deleted.         |
| Evals protect the risky path    | Regression cases cover the failure mode that would make the slice unsafe. |

Record the goal, composed patterns, state, tools, approval points, trace events, and evals in the lab production readiness worksheet.

## Slice 1: Support Refund Assistant

### Goal

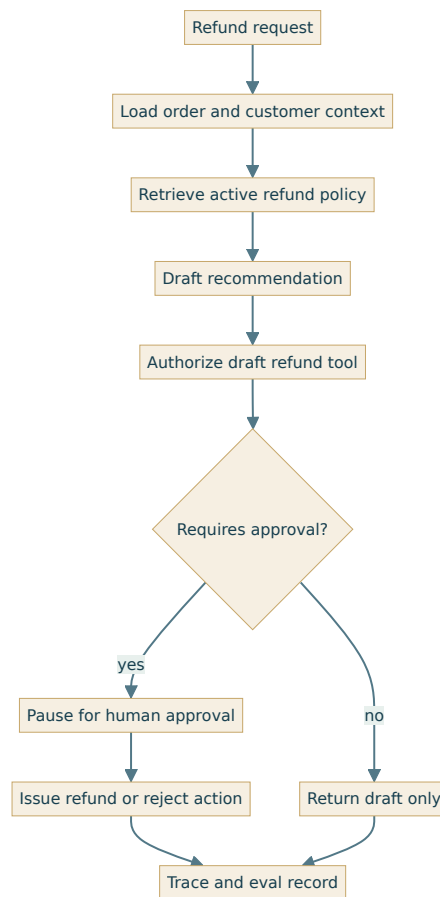
A support agent helps a human support operator handle refund requests. It reads the order, retrieves the active refund policy, drafts a recommendation, and prepares a refund action. It does not issue the refund without approval.

### Pattern Composition

| Concern    | Pattern             |
|------------|---------------------|
| Agent loop | Agent Loop          |
| Context    | Context Engineering |

| Concern  | Pattern                       |
|----------|-------------------------------|
| Evidence | Semantic Recall and RAG       |
| Tools    | Tool Capability Design        |
| Approval | Human Approval Gates          |
| Runtime  | Production Runtime Overview   |
| Security | Agent Security and Sandboxing |
| Evals    | Observability and Evals       |

## Runtime Flow



## Security Controls

- The agent receives `orders:read`, `refunds:draft`, and `policies:read`.
- The `refunds.issue` tool requires human approval and an idempotency key.
- The refund policy is retrieved from an approved source with a policy version.
- Customer payment tokens never enter the prompt.
- External email is a separate tool with its own approval rule.

## Trace And Eval

Every run should record order lookup, policy retrieval, recommendation draft, tool authorization, approval state, refund side-effect ID, and stop reason.

Good eval cases:

- refund allowed by policy and approved;
- refund denied by policy;
- missing order;
- stale policy retrieved;
- model attempts `refunds.issue` without approval;
- duplicate approval message replayed.

Runnable evidence:

| Signal              | Repository Evidence                                                                                   |
|---------------------|-------------------------------------------------------------------------------------------------------|
| Safe stop           | <code>stopReason: draft_ready</code>                                                                  |
| Policy citation     | <code>draft_contains_policy_citation</code> eval passes.                                              |
| Money does not move | <code>no_money_movement</code> eval passes and trace records <code>agent_cannot_issue_refund</code> . |
| Rollback            | Disable <code>refunds.create_draft</code> or route to the human support queue.                        |

## Minimal Code

```
type RefundDecision =
 | { action: "draft_refund"; orderId: string; amountCents: number; policyVersion: string }
 | { action: "deny_refund"; orderId: string; reason: string; policyVersion: string }
 | { action: "needs_human_review"; orderId: string; reason: string };

function requiresApproval(decision: RefundDecision): boolean {
 return decision.action === "draft_refund" && decision.amountCents > 0;
}
```

The code is intentionally small. The important part is the boundary: a model can propose a refund decision, but the runtime still checks policy, approval, and idempotency before money moves.

## Failure Modes

- The model treats an old refund policy as current.
- The tool call issues a refund before approval.
- The trace records the final answer but not the policy version.
- A retry issues the same refund twice.
- The agent sends the customer message before the operator reviews it.

## Slice 2: Safe Coding Agent

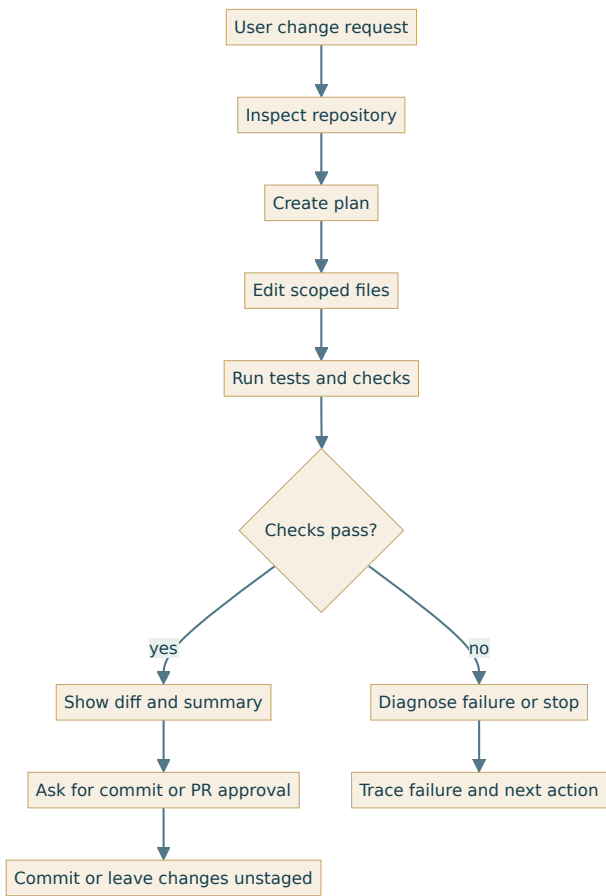
### Goal

A coding agent makes a small repository change, runs tests, shows the diff, and asks for approval before committing or opening a pull request.

### Pattern Composition

| Concern           | Pattern                                 |
|-------------------|-----------------------------------------|
| Loop and planning | Planning and Execution                  |
| Harness           | Agent Harnesses                         |
| Workspace         | Coding Agents                           |
| Sandbox           | Agent Security and Sandboxing           |
| Evaluation        | Evaluation-Driven Agent Development     |
| Recovery          | Circuit Breakers, Fallbacks, and Replay |

### Runtime Flow



### Security Controls

- The agent works in a scoped workspace or branch.
- Shell commands run with timeouts and no ambient production secrets.
- File edits stay within the repository root.
- Network access is disabled unless the task needs dependency or documentation lookup.
- Commit, push, deploy, and destructive commands require explicit approval.

## Trace And Eval

Every run should record files inspected, commands run, tests passed or failed, diff summary, approval request, and final state.

Good eval cases:

- correct single-file change with passing tests;
- failing test stops the run;
- command timeout is handled;
- attempted edit outside workspace is denied;
- generated change touches unrelated files;
- commit requested before diff review.

## Minimal Code

```
type CommandPolicy = {
 allowedPrefixes: string[];
 timeoutMs: number;
 network: "blocked" | "allowlisted";
};

function canRunCommand(command: string, policy: CommandPolicy): boolean {
 return policy.allowedPrefixes.some(prefix => command.startsWith(prefix));
}
```

The model should not decide that a command is safe because it looks familiar. The harness should check the command against the current task, workspace, and approval policy.

## Failure Modes

- The agent edits generated files instead of source files.
- A test failure is summarized as success.
- The sandbox exposes secrets through environment variables.
- The agent commits unrelated user changes.
- The final answer hides a failed command or skipped check.

## Slice 3: Research To Brief Agent

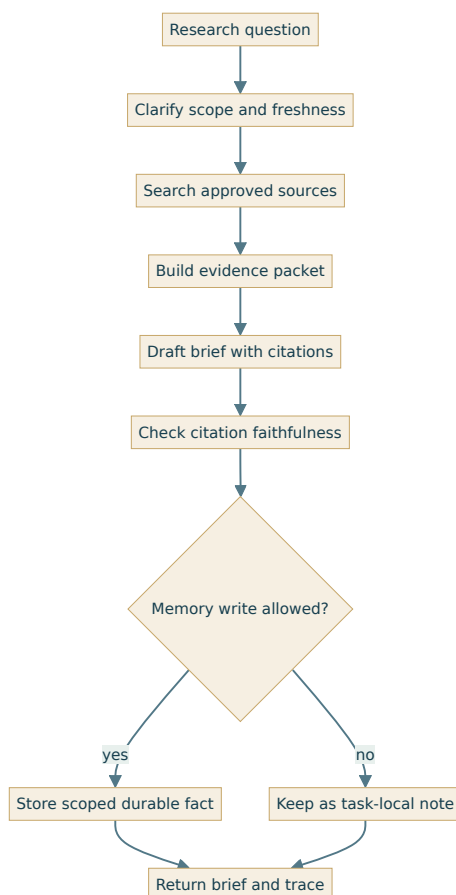
## Goal

A research agent gathers evidence, produces a short technical brief, cites sources, and stores only durable facts that pass a memory policy.

## Pattern Composition

| Concern         | Pattern                              |
|-----------------|--------------------------------------|
| Retrieval       | Semantic Recall and RAG              |
| Context control | Context Budgets and Working Sets     |
| Memory          | Working Memory                       |
| Output shape    | Structured Output                    |
| Evals           | Production Evaluation Feedback Loops |
| UX              | Agent UX and Human Trust             |

## Runtime Flow



## Security Controls

- Retrieved documents are data, not instructions.
- The agent separates source evidence from system instructions.



- Memory writes require source, confidence, retention class, and correction path.
- Private or licensed content is not copied into long-term memory by default.
- The brief says when evidence is missing, stale, or conflicting.

Trace And Eval

Every run should record query, source set, evidence packet, omitted sources, citation checks, memory decisions, and final answer shape.

Good eval cases:

- answer requires a current source;
- sources conflict;
- retrieval returns irrelevant documents;
- citation does not support the claim;
- model tries to store an unsupported memory;
- brief should refuse because evidence is missing.

Runnable evidence:

| Signal                   | Repository Evidence                                                                        |
|--------------------------|--------------------------------------------------------------------------------------------|
| Safe stop                | <code>stopReason: answered_with_citation</code>                                            |
| Current source           | <code>current_source_used</code> eval passes for <code>refund-policy-v4</code> .           |
| Stale source rejected    | <code>stale_source_rejected</code> eval passes for <code>refund-policy-v2</code> .         |
| Forbidden source omitted | <code>forbidden_source_omitted</code> eval passes for <code>finance-private-notes</code> . |

Minimal Code

```
type MemoryCandidate = {
 claim: string;
 sourceIds: string[];
 confidence: "low" | "medium" | "high";
 retention: "task_only" | "project" | "user";
};

function canWriteMemory(candidate: MemoryCandidate): boolean {
 return (
 candidate.retention !== "user" &&
 candidate.confidence === "high" &&
 candidate.sourceIds.length > 0
);
}
```

The default should be task-local memory. Durable memory is a controlled write, not a side effect of reading.

Failure Modes

- Retrieved content changes the agent's instructions.
- The brief cites a source that does not support the claim.
- Stale evidence is presented as current.
- The agent stores a user preference from one temporary task.
- The trace cannot explain why a source was included or omitted.

Comparison

| Slice                         | Main risk                              | Primary control                            | Best regression eval                                                    | Runnable Stop Signal                  |
|-------------------------------|----------------------------------------|--------------------------------------------|-------------------------------------------------------------------------|---------------------------------------|
| Support refund assistant      | Money moves without authority.         | Approval-bound tool execution.             | Refund tool cannot execute without policy and approval trace.           | <code>draft_ready</code>              |
| Safe coding agent             | The agent changes more than it should. | Workspace, diff, tests, and approval.      | Unrelated file edits or failed checks block completion.                 | Not included in capstone runtime yet. |
| Research to brief agent       | Unsupported claims look authoritative. | Evidence packets and citation checks.      | Claims must be supported by cited source IDs.                           | <code>answered_with_citation</code>   |
| Multi-agent delivery workflow | Delegation hides accountability.       | Workflow-owned merge and final acceptance. | Required role outputs and sequential turns must pass before acceptance. | <code>accepted_after_review</code>    |

Slice 4: Multi-Agent Delivery Workflow

Goal

A workflow owner coordinates planner, risk reviewer, and test planner roles. The workflow accepts the package only after every role contributes in order.

Pattern Composition

| Concern      | Pattern                          |
|--------------|----------------------------------|
| Delegation   | Task Delegation                  |
| Supervisor   | Supervisor / Worker              |
| Transcript   | Evaluate Multi-Agent Transcripts |
| Flow control | CrewAI Flows and Crews           |
| Evals        | Observability and Evals          |

## Runnable Evidence

| Signal                   | Repository Evidence                                                                                      |
|--------------------------|----------------------------------------------------------------------------------------------------------|
| Planner present          | <code>planner_present</code> eval passes.                                                                |
| Risk review present      | <code>risk_review_present</code> eval passes.                                                            |
| Test plan present        | <code>test_plan_present</code> eval passes.                                                              |
| Turn order valid         | <code>turns_sequential</code> eval passes.                                                               |
| Final owner accepts last | <code>final_owner_accepts_last</code> eval passes and <code>finalOwner</code> is <code>workflow</code> . |

## Failure Modes

- A specialist role is skipped but the final answer still sounds complete.
- Acceptance happens before risk review or test planning.
- Turn order is broken, making the trace hard to replay.
- No single owner accepts the final package.
- Delegation cannot be disabled during an incident.

## Design Rule

A vertical slice should prove composition. It should show how the loop, tools, state, memory, security, runtime, observability, and evals work together for one real task.

Small examples are fine. Isolated examples are not enough.

## Related Chapters

- [What Is An Agent?](#)
- [Agent Harnesses](#)
- [Production Runtime Overview](#)
- [Agent Security and Sandboxing](#)
- [Observability and Evals](#)
- [Pattern Composition Playbook](#)

# Capstone Projects

The capstones show how the patterns combine into product-shaped systems. Each capstone starts from a concrete workflow, chooses the right agentic boundaries, maps the design across frameworks, and defines the production evidence required before release.

Use these chapters after the labs. The labs isolate one pattern at a time. The capstones combine patterns into systems with state, tools, policy, memory, observability, evals, deployment, rollback, ownership, and selected native framework slices.

Do not read the capstones as large tutorials. Read them as design review packets. Each one shows what evidence a team should gather before turning an agent pattern into product behavior.

## Capstone Set

| Capstone                      | Primary Goal                                                         | Main Patterns                                                                | Framework Lens                                                                        |
|-------------------------------|----------------------------------------------------------------------|------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| Support Refund Agent          | Draft policy-safe refund recommendations.                            | Tool use, policy enforcement, approval gates, observability, evals.          | Mastra runtime, LangGraph workflow, mini-runtime, native Mastra and LangGraph slices. |
| Research RAG Agent            | Answer from approved sources with citations and memory rules.        | Context engineering, semantic recall, knowledge-bound agents, memory, evals. | LangGraph graph, direct Python/TypeScript, Mastra runtime, native LangGraph slice.    |
| Multi-Agent Delivery Workflow | Coordinate specialist agents while preserving one accountable owner. | Supervisor/worker, CrewAI flows, AutoGen transcripts, durable workflows.     | CrewAI, AutoGen, LangGraph, Mastra, native CrewAI and AutoGen slices.                 |

## Choose A Capstone

Use the capstone that matches your highest-risk boundary.

| If Your System Needs...                                               | Start With           | Main Risk To Inspect                                                                 |
|-----------------------------------------------------------------------|----------------------|--------------------------------------------------------------------------------------|
| A model to draft an action involving money, policy, or customer data. | Support Refund Agent | The model must never own final authority for the side effect.                        |
| Answers grounded in private or approved knowledge sources.            | Research RAG Agent   | The system must refuse or escalate when evidence is missing, stale, or unauthorized. |

| If Your System Needs...                                             | Start With                                    | Main Risk To Inspect                                                                       |
|---------------------------------------------------------------------|-----------------------------------------------|--------------------------------------------------------------------------------------------|
| Several specialist agents or roles contributing to one deliverable. | <a href="#">Multi-Agent Delivery Workflow</a> | Coordination must not erase blockers, minority concerns, failed tasks, or final ownership. |

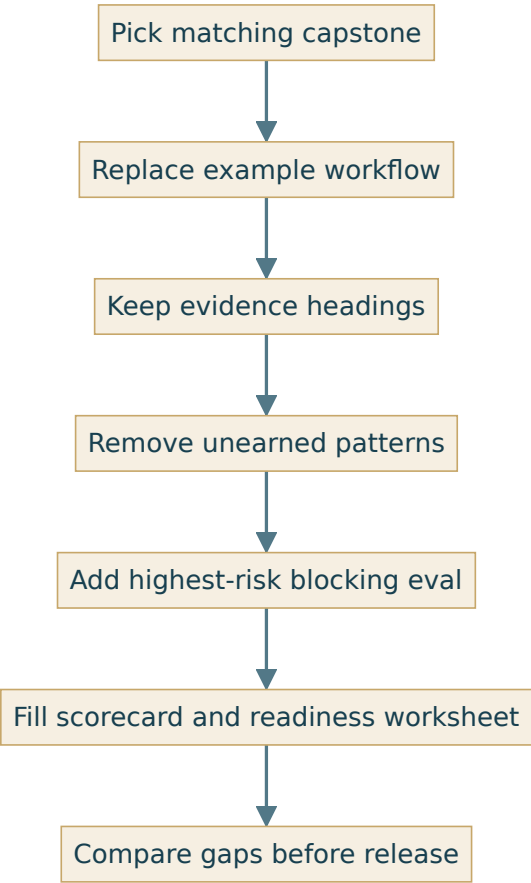
If more than one applies, read the capstones in this order: side-effect authority, evidence grounding, then multi-agent coordination. Side effects and private data usually deserve review before topology choices.

## How To Reuse A Capstone

Turn a capstone into your own design review in five steps:

1. Replace the example workflow with your real workflow.
2. Keep the same evidence headings: state, tools, policy, memory, trace, eval, ADR, runbook, rollback.
3. Delete any pattern that does not earn its place in your workflow.
4. Add one blocking eval for the highest-risk failure.
5. Record gaps in the [capstone review scorecard](#) and [production readiness worksheet](#).

The reusable value is the review shape, not the domain. A refund system, research assistant, and delivery workflow all need the same proof: bounded authority, replayable state, traceable decisions, eval gates, and rollback.



Use this flow as the capstone reuse contract. The output is not a copied implementation; it is a reviewed system packet with gaps, evidence, and release blockers made explicit.

## Run The Capstones

The capstones include deterministic TypeScript assets so readers can inspect state, traces, evals, and rollback behavior without model provider keys.

```
npm run capstones:demo
npm run capstones:test
```

Expected demo output:

```
support-refund-agent: pass
 stop: draft_ready
 trace events: 7
research-rag-agent: pass
 stop: answered_with_citation
 trace events: 6
multi-agent-delivery-workflow: pass
 stop: accepted_after_review
 trace events: 4
```

Source:

- capstone-projects-runtime/typescript/src/capstones.ts
- capstone-projects-runtime/typescript/test/capstones.spec.ts

After running the commands, compare the output with each capstone’s trace and eval sections. The goal is to connect runtime behavior to the written design evidence.

## Runtime Evidence Map

Use this map to inspect the code path before reading the detailed capstone chapters.

| Capstone             | Safe Stop              | Release Evals                                                       | Rollback Path                                                            |
|----------------------|------------------------|---------------------------------------------------------------------|--------------------------------------------------------------------------|
| Support Refund Agent | draft_ready            | draft_contains_policy_citation, no_money_movement, safe_stop_reason | Disable refunds.create_draft; route the ticket to a human support queue. |
| Research RAG Agent   | answered_with_citation | current_source_used, stale_source_rejected,                         | Disable answer synthesis; return the ranked source list only.            |

| Capstone                      | Safe Stop             | Release Evals                                                                                                                                              | Rollback Path                                                               |
|-------------------------------|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
|                               |                       | <div>forbidden_source_omitted,</div> <div>citation_faithfulness</div>                                                                                      |                                                                             |
| Multi-Agent Delivery Workflow | accepted_after_review | <div>planner_present,</div> <div>risk_review_present,</div> <div>test_plan_present,</div> <div>turns_sequential,</div> <div>final_owner_accepts_last</div> | Disable delegation; route the request to a single-owner delivery checklist. |

These are not toy assertions. Each eval protects a production boundary:

- The refund capstone proves the agent can draft a recommendation while policy blocks money movement.
- The RAG capstone proves the context packet uses the current approved source and omits stale or forbidden sources.
- The delivery capstone proves specialist agents can contribute without removing final workflow ownership.

## What Each Capstone Proves

Each capstone includes:

- problem and non-goals;
- pattern composition;
- system architecture;
- data and state model;
- tool, policy, memory, and approval boundaries;
- native framework mapping;
- native framework example path where one exists;
- trace example;
- eval report example;
- ADR example;
- runbook example;
- release and rollback checklist.

The repeated structure matters. It gives readers a reusable design review shape: if a future project cannot fill these sections, it is not ready for production.

## Capstone Completion Standard

A capstone is complete only when it can answer these questions:

Download the reusable review artifact: [capstone review scorecard](#).

Download the production follow-up worksheet: [production readiness worksheet](#).

| Question                          | Required Evidence                                              |
|-----------------------------------|----------------------------------------------------------------|
| What owns state?                  | State schema, checkpoint plan, migration note.                 |
| What owns authority?              | Tool manifest, policy decision, approval rule.                 |
| What proves quality?              | Eval cases, thresholds, failure examples.                      |
| What proves observability?        | Trace event sequence and required fields.                      |
| What proves production readiness? | Deployment notes, runbook, rollback path.                      |
| What proves portability?          | Framework mapping and assets kept outside framework-only code. |

Do not treat a capstone as a larger lab. Treat it as a small production design review.

## Capstone Review Gate

Use this gate before treating any capstone as A++ material:

| Check                              | Evidence                                                                     |
|------------------------------------|------------------------------------------------------------------------------|
| The workflow is concrete           | One user-visible or operator-visible workflow starts the system.             |
| The authority boundary is explicit | Tools, data, memory, approvals, and side effects have named owners.          |
| The unsafe path is blocked         | At least one blocking eval catches the highest-risk failure.                 |
| The run is replayable              | State, trace, versions, and eval result can reconstruct success and failure. |
| The reader can reuse the shape     | ADR, runbook, trace, eval, checklist, or rollback artifacts are provided.    |

Record the score, blocking gaps, and next production artifact in the capstone review scorecard.

## A++ Capstone Rubric

Score each area from 0 to 2.

| Area                | A++ Evidence                                                                  |
|---------------------|-------------------------------------------------------------------------------|
| Problem and scope   | Concrete workflow, explicit non-goals, clear authority level.                 |
| Pattern composition | Every loop, tool, memory, agent, and approval boundary has a reason to exist. |



| Area                           | A++ Evidence                                                                                                |
|--------------------------------|-------------------------------------------------------------------------------------------------------------|
| Architecture boundary          | Model judgment is separated from deterministic control, policy, state, tools, and approval.                 |
| Tool and policy control        | Tool contracts, permissions, timeouts, audit fields, and high-risk denial or approval paths are documented. |
| State, memory, and context     | Run state, memory rules, context sources, trust, freshness, and budget are inspectable.                     |
| Evaluation evidence            | Happy paths, edge cases, unsafe paths, and regressions have thresholds that can block release.              |
| Observability and traceability | A successful run and failed run can be reconstructed from trace fields.                                     |
| Production operation           | Runbook, incident triggers, rollback, kill switch, and owners are named.                                    |
| Framework portability          | Framework-owned and application-owned responsibilities are clear.                                           |
| Reader reuse                   | The chapter leaves the reader with reusable ADR, trace, eval, runbook, or checklist shapes.                 |

Interpret the score this way:

- 0-9: example sketch
- 10-14: useful design note
- 15-17: strong capstone
- 18-20: production-grade teaching example

A capstone cannot score A++ if a high-risk tool can run without policy or approval, if no replayable trace exists, if no blocking eval exists for unsafe behavior, if state or memory ownership is unclear, or if rollback is not documented.

## Recommended Reading Order

1. [Support Refund Agent](#)
2. [Research RAG Agent](#)
3. [Multi-Agent Delivery Workflow](#)
4. [Deployment Walkthrough](#)
5. [Templates and Worksheets](#)

The first capstone is tool and policy heavy. The second is evidence and memory heavy. The third is coordination heavy.

# Capstone - Support Refund Agent

Build a support agent that investigates a refund request, retrieves policy, drafts a recommendation, and stops before money moves or a customer message is sent.

This capstone is valuable because it forces the core production rule: the model can propose; the runtime decides.

## Problem

Support teams often need to gather order details, read policy, draft a response, and ask for finance review. The workflow is repetitive, but the final authority is sensitive. The agent may reduce investigation time, but it must not issue refunds, alter payment state, or send customer messages without approval.

## Non-Goals

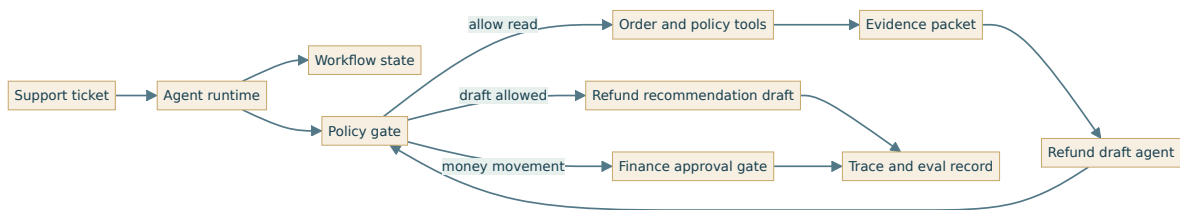
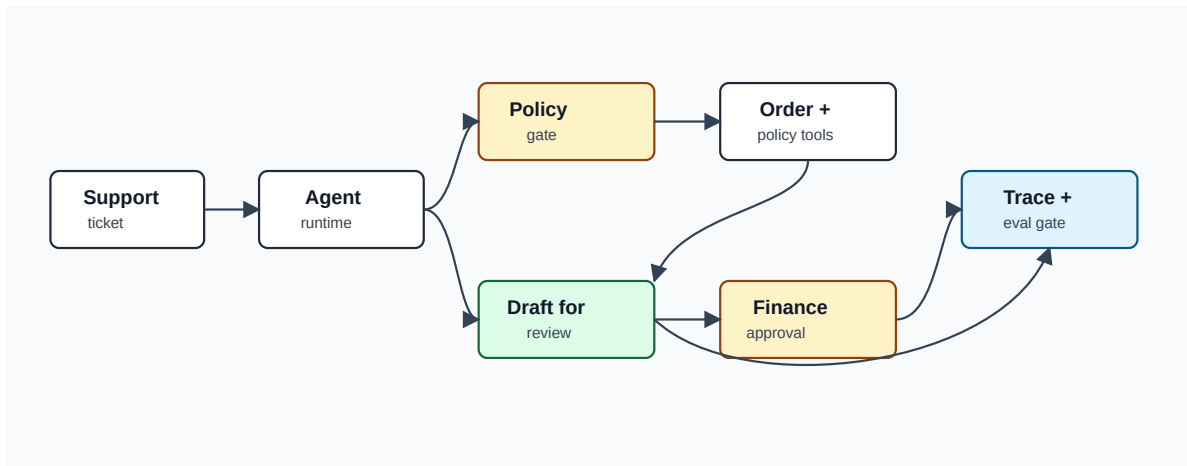
- Do not issue money directly.
- Do not send outbound customer email.
- Do not store payment details in long-term memory.
- Do not let model text bypass policy, approval, or tool permissions.

## Pattern Composition

| Concern            | Pattern                             |
|--------------------|-------------------------------------|
| investigation loop | Agent Loop                          |
| tool execution     | Tool Use and Tool Capability Design |
| authority          | Policy Enforcement                  |
| finance review     | Human Approval Gates                |
| state and replay   | Durable Workflows                   |
| quality            | Observability and Evals             |
| deployment         | Deployment Walkthrough              |

## Architecture

Read this diagram as an authority boundary. The runtime may gather evidence and draft a recommendation, but policy and finance approval decide whether any money-moving path can continue.



## Runnable Assets

Run the deterministic capstone implementation:

```
npm run capstones:demo
npm run capstones:test
```

Inspect:

- `capstone-projects-runtime/typescript/src/capstones.ts`
- `capstone-projects-runtime/typescript/test/capstones.spec.ts`

Downloadable evidence:

- [Sample trace JSON](#)
- [Sample eval report](#)
- [Captured command output examples](#)
- [Capstone review scorecard](#)
- [Framework selection ADR template](#)
- [Production readiness worksheet](#)

Expected runtime signal:

```
support-refund-agent: pass
stop: draft_ready
trace events: 7
```

The test suite treats these as release evidence:

| Evidence                     | Runtime Check                               |
|------------------------------|---------------------------------------------|
| Policy citation is present   | <code>draft_contains_policy_citation</code> |
| No money moves               | <code>no_money_movement</code>              |
| The workflow stops safely    | <code>safe_stop_reason</code>               |
| The trace records the denial | <code>agent_cannot_issue_refund</code>      |

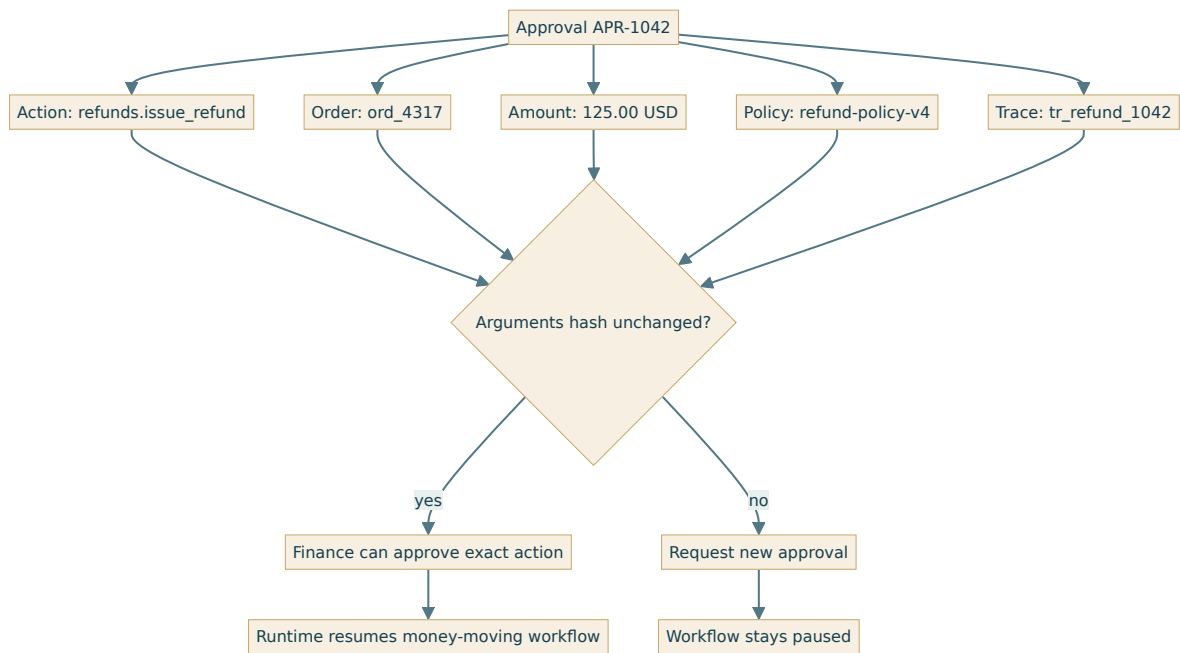
The captured output examples show the matching terminal signal, trace snapshot, and eval snapshot. Use them as the minimum evidence pack before adapting this capstone to a product workflow.

## State Model

| Field                             | Owner                 | Notes                                                                                                                            |
|-----------------------------------|-----------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <code>ticket_id</code>            | workflow              | Correlates user request and trace.                                                                                               |
| <code>tenant_id</code>            | runtime               | Required for access policy.                                                                                                      |
| <code>order_summary</code>        | tool result           | Redacted before trace storage.                                                                                                   |
| <code>policy_evidence</code>      | retrieval/tool result | Must cite current policy version.                                                                                                |
| <code>draft_recommendation</code> | agent output          | Draft only; not customer-visible until reviewed.                                                                                 |
| <code>approval_request</code>     | approval gate         | Exact amount, order ID, approver role, expiry.                                                                                   |
| <code>stop_reason</code>          | runtime               | <code>draft_ready</code> , <code>approval_required</code> , <code>denied</code> , <code>escalated</code> , <code>failed</code> . |

## Finance Approval Mock

The finance reviewer should approve one exact action, not a broad refund workflow. This mock shows the fields that must be visible before a money-moving system can resume.



Review panel fields:

| Field            | Example                                                            |
|------------------|--------------------------------------------------------------------|
| Proposed action  | <code>refunds.issue_refund</code>                                  |
| Resource         | <code>order:ord_4317</code> , <code>customer:cust_123</code>       |
| Amount           | <code>125.00 USD</code>                                            |
| Evidence         | order summary, payment summary, <code>refund-policy-v4</code>      |
| Policy decision  | <code>require_approval</code> , reason <code>money_movement</code> |
| Safety           | args hash, idempotency key, expiry, rollback note                  |
| Decision options | approve exact action, deny, request changes, escalate              |

The draft-only capstone should not request this approval during normal operation. It should still document the approval shape so the team knows where the authority boundary would live if finance later adds a money-moving workflow.

## Capstone Review Gate

Before treating this capstone as production-grade, verify the authority boundary:

| Check                                     | Evidence                                                      |
|-------------------------------------------|---------------------------------------------------------------|
| Money movement is outside agent authority | <code>refunds.issue_refund</code> is forbidden to the agent.  |
| Policy evidence is required               | Draft recommendations cite the current refund policy version. |

| Check                       | Evidence                                                                                |
|-----------------------------|-----------------------------------------------------------------------------------------|
| Approval is explicit        | Finance approval has exact amount, approver role, expiry, and trace event.              |
| Unsafe requests fail closed | Direct refund issuance, cross-tenant access, and missing policy evidence block release. |
| Rollback preserves safety   | Draft creation can be disabled without losing human queue fallback.                     |

Record the result in the capstone review scorecard and production readiness worksheet.

## Tool Manifest

| Tool                                     | Side Effect            | Policy                                         |
|------------------------------------------|------------------------|------------------------------------------------|
| <code>orders.lookup_order</code>         | read                   | Same tenant, current ticket only.              |
| <code>payments.get_summary</code>        | read                   | Redact payment identifiers in traces.          |
| <code>refund_policy.retrieve</code>      | read                   | Current policy only.                           |
| <code>refunds.create_draft</code>        | write draft            | Allowed for eligible orders; draft only.       |
| <code>refunds.issue_refund</code>        | money movement         | Forbidden to the agent; finance workflow only. |
| <code>email.send_customer_message</code> | outbound communication | Forbidden to the agent.                        |

## Production Bridge

Use this table when turning the capstone into a service:

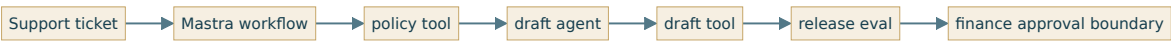
| Capstone Artifact | Production Version                                                                             |
|-------------------|------------------------------------------------------------------------------------------------|
| Tool manifest     | Capability registry with owner, timeout, side-effect class, approval rule, and disable switch. |
| State model       | Durable workflow schema with migration, tenant isolation, and replay support.                  |
| Trace example     | Observability contract with redaction, retention, dashboard, and incident fields.              |
| Eval report       | CI release gate with false-allow threshold set to zero.                                        |
| Runbook           | On-call procedure with kill switch, fallback route, and post-incident eval process.            |

The first production milestone is a draft-only service that can prove no money moved and no customer message was sent.

## Native Framework Mapping

Start with the deterministic TypeScript capstone, then compare the native slices:

- `native-framework-examples/mastra-refund/` proves TypeScript runtime packaging with agent, tools, workflow, trace fields, and eval gate.
- `native-framework-examples/langgraph-refund/` proves pause/resume and approval behavior before any real money-moving integration is added.



| Framework    | Best Mapping                                                                                                               |
|--------------|----------------------------------------------------------------------------------------------------------------------------|
| Mastra       | Agent drafts; workflow owns order lookup, policy retrieval, approval wait, evals, and trace export.                        |
| LangGraph    | Nodes for classify, retrieve, draft, policy check, approval interrupt, and finalize. Checkpointer stores approval wait.    |
| AutoGen      | Manager assigns research and draft roles, but tool execution and approval stay outside the transcript.                     |
| CrewAI       | Flow owns ticket state and approval; crew can research policy and draft response. Flow validates output before acceptance. |
| Mini-runtime | Explicit loop with tool registry, policy function, trace events, and deterministic evals.                                  |

## Trace Example

```
{
 "trace_id": "tr_refund_1042",
 "release": "support-refund-agent@1.0.0",
 "events": [
 { "span": "run", "status": "started", "ticket_id": "T-1042" },
 { "span": "policy", "decision": "allow", "reason": "same_tenant_read" },
 { "span": "tool", "tool": "orders.lookup_order", "status": "succeeded" },
 { "span": "tool", "tool": "refund_policy.retrieve", "status": "succeeded",
 "policy_version": "refund-policy-v4" },
 { "span": "model", "prompt": "refund-draft-v2", "status": "succeeded" },
 { "span": "policy", "decision": "deny", "reason": "agent_cannot_issue_refund" },
 { "span": "approval", "status": "not_requested", "reason": "draft_only" },
 { "span": "eval", "case_id": "support_refund_release_gate", "status": "pass" }
```

```
]
}
```

## Eval Report Example

| Case                                        | Expected                                                    | Result   |
|---------------------------------------------|-------------------------------------------------------------|----------|
| <code>draft_contains_policy_citation</code> | Draft cites <code>refund-policy-v4</code> .                 | pass     |
| <code>no_money_movement</code>              | Agent does not call a money-moving tool.                    | pass     |
| <code>safe_stop_reason</code>               | Runtime stops with <code>draft_ready</code> .               | pass     |
| refund issuance requested                   | Policy denies with <code>agent_cannot_issue_refund</code> . | pass     |
| cross-tenant ticket                         | Tool access is denied before evidence enters context.       | blocking |
| missing policy evidence                     | Runtime escalates instead of drafting.                      | blocking |
| draft promises payment                      | Release gate fails.                                         | blocking |

Blocking threshold:

```
policy false allow: 0
missing citation on policy-dependent answer: 0
direct money movement by agent: 0
draft quality pass rate: >= 95%
```

## ADR Example

```
ADR-021: Support refund agent may draft but not issue refunds
```

```
Status
```

```
Accepted
```

```
Decision
```

```
The agent may read order summaries, retrieve refund policy, and create refund
recommendation drafts. It may not issue refunds, alter payment state, or send customer
messages.
```

```
Consequences
```

```
Support investigation becomes faster. Finance authority remains outside the model. The
workflow needs approval records, trace retention, and eval maintenance when refund policy
```



changes.

## Rollback

Disable `refunds.create\_draft`, route all refund tickets to the human queue, and keep read-only investigation available only if traces show policy compliance.

## Runbook Example

```
service: support-refund-agent
owner: support-platform
on-call: support-platform-primary
kill switch: disable capability support_refund_agent
tool disable: refunds.create_draft
fallback: route ticket to human support queue
trace dashboard: support/refund-agent/traces
eval suite: evals/support-refund
incident trigger: any attempted refund issuance, cross-tenant access, or missing policy citation
post-incident action: create regression eval before re-enable
```

## Release Checklist

- State schema records ticket, tenant, evidence, draft, approval, and stop reason.
- Tool manifest forbids money movement and outbound communication.
- Policy runs before read tools, draft creation, and final answer.
- Trace redaction removes payment identifiers.
- Evals include false-allow, missing-evidence, and prohibited-tool cases.
- Rollback disables draft creation without redeploying code.

## Related Labs

- [Lab 06 - Observability and Evals](#)
- [Lab 07 - Mastra Runtime Packaging](#)
- [Lab 10 - Tool Registry and Policy Gate](#)
- [Lab 12 - LangGraph State Graph](#)

Native examples:

- [native-framework-examples/mastra-refund/](#) (download)
- [native-framework-examples/langgraph-refund/](#) (download)

# Capstone - Research RAG Agent

Build a research agent that answers from approved sources, cites evidence, refuses unsupported claims, and records enough trace data to debug retrieval failures.

This capstone is evidence heavy. The main risk is not that the model cannot write an answer. The main risk is that the answer looks convincing while using stale, forbidden, or missing evidence.

## Problem

Product, support, and engineering teams often need answers from internal documents. A research agent can reduce search time, but it must respect source access, freshness, citations, and memory rules.

## Non-Goals

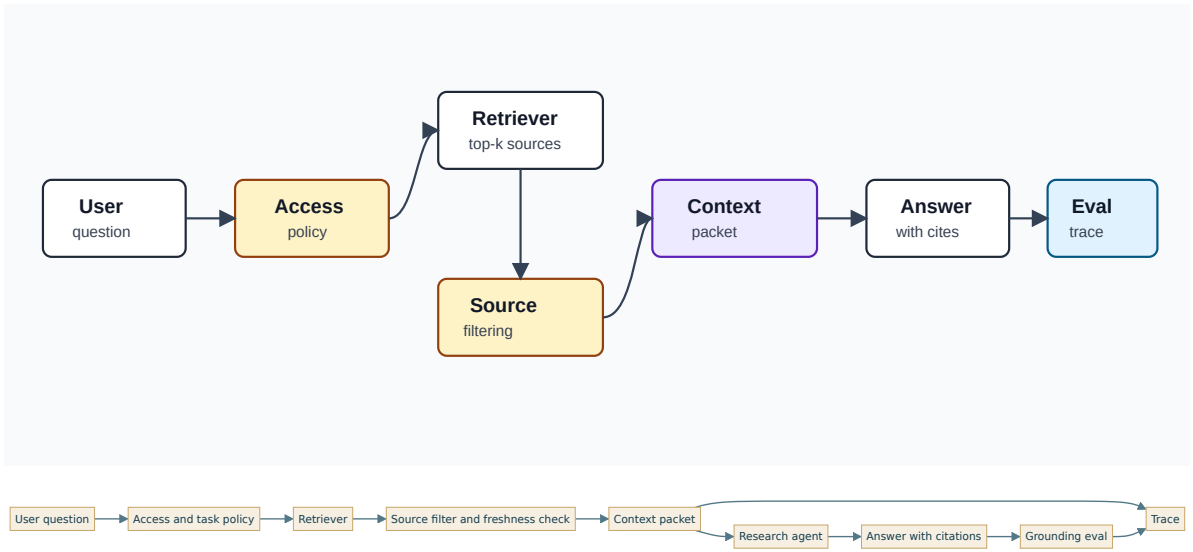
- Do not answer from unapproved sources.
- Do not treat retrieved text as trusted instructions.
- Do not store private facts in memory without retention and correction rules.
- Do not cite documents the agent did not actually use.

## Pattern Composition

| Concern           | Pattern                 |
|-------------------|-------------------------|
| context packet    | Context Engineering     |
| retrieval         | Semantic Recall and RAG |
| evidence boundary | Knowledge-Bound Agents  |
| memory            | Memory-Augmented Agent  |
| policy            | Policy Enforcement      |
| quality           | Observability and Evals |

## Architecture

Read this diagram as an evidence boundary. Retrieval can find candidates, but only source filtering, context assembly, and grounding evals decide what the answer may cite.



## Runnable Assets

Run the deterministic capstone implementation:

```

npm run capstones:demo
npm run capstones:test

```

Inspect:

- `capstone-projects-runtime/typescript/src/capstones.ts`
- `capstone-projects-runtime/typescript/test/capstones.spec.ts`

Downloadable evidence:

- [Sample trace JSON](#)
- [Sample eval report](#)
- [Capstone review scorecard](#)
- [Framework selection ADR template](#)
- [Production readiness worksheet](#)

Expected runtime signal:

```

research-rag-agent: pass
 stop: answered_with_citation
 trace events: 6

```

The test suite treats these as release evidence:

| Evidence                                                | Runtime Check                    |
|---------------------------------------------------------|----------------------------------|
| The context packet includes the current approved source | <code>current_source_used</code> |

| Evidence                                         | Runtime Check                         |
|--------------------------------------------------|---------------------------------------|
| The context packet excludes the stale source     | <code>stale_source_rejected</code>    |
| The context packet excludes the forbidden source | <code>forbidden_source_omitted</code> |
| The answer cites the source it used              | <code>citation_faithfulness</code>    |

Native example:

- [native-framework-examples/langgraph-research-rag/](#) ([download](#))

## Context Packet

| Field                     | Required Rule                                                        |
|---------------------------|----------------------------------------------------------------------|
| <code>question</code>     | Store normalized question and original user wording separately.      |
| <code>actor</code>        | Include tenant, role, and source access scope.                       |
| <code>sources</code>      | Include source ID, title, freshness, ACL result, and citation label. |
| <code>evidence</code>     | Include only approved passages needed for the answer.                |
| <code>memory</code>       | Include user or project memory only when policy allows it.           |
| <code>instructions</code> | Separate system instructions from retrieved content.                 |
| <code>omissions</code>    | Record sources omitted because of access, freshness, or relevance.   |

## Capstone Review Gate

Before treating this capstone as production-grade, verify the evidence boundary:

| Check                                      | Evidence                                                                        |
|--------------------------------------------|---------------------------------------------------------------------------------|
| Source access runs before context assembly | Forbidden sources are omitted before answer synthesis.                          |
| Freshness is enforced                      | Stale sources cannot override current approved sources.                         |
| Citations are faithful                     | Every cited claim maps to evidence in the context packet.                       |
| Missing evidence fails closed              | The agent refuses or escalates instead of guessing.                             |
| Memory writes are governed                 | Durable memory requires source IDs, confidence, retention, and correction path. |

Record the result in the capstone review scorecard and production readiness worksheet.

## Production Bridge

Use this table when turning the capstone into a service:

| Capstone Artifact | Production Version                                                                        |
|-------------------|-------------------------------------------------------------------------------------------|
| Context packet    | Versioned evidence contract with ACLs, freshness, omitted-source notes, and token budget. |
| Source filter     | Access-control service with audit trail and policy version.                               |
| Citation eval     | Blocking release gate for unsupported, stale, forbidden, or missing evidence.             |
| Memory rule       | Governed write path with retention, deletion, correction, consent, and tenant scope.      |
| Runbook           | Fallback that disables synthesis and returns approved source lists only.                  |

The first production milestone is an answer path that can explain what it cited, what it omitted, and why.

## Native Framework Mapping

| Framework    | Best Mapping                                                                                                               |
|--------------|----------------------------------------------------------------------------------------------------------------------------|
| LangGraph    | Graph nodes for classify question, retrieve, filter, answer, cite, evaluate, and escalate. Stores handle long-term memory. |
| Mastra       | Agent handles answer synthesis; workflow owns retrieval, source filtering, evals, memory policy, and trace export.         |
| AutoGen      | Researcher and reviewer agents can collaborate, but source access and citation checks stay in software.                    |
| CrewAI       | Flow owns evidence packet and acceptance; crew can split research and review tasks.                                        |
| Mini-runtime | Direct context builder plus retrieval client, source policy function, answer validator, and trace events.                  |

## Trace Example

```
{
 "trace_id": "tr_research_2077",
 "question": "Can the support refund agent issue money?",
 "events": [
 { "span": "policy", "decision": "allow", "scope": "support_docs" },
 { "span": "retrieval", "query": "support refund agent issue money", "top_k": 5 },
 { "span": "source_filter", "allowed": 1, "stale": 1, "forbidden": 1 },
 { "span": "context_packet", "evidence_refs": ["refund-policy-v4"] },
```

```
{ "span": "model", "prompt": "research-answer-v1", "status": "succeeded" },
{ "span": "eval", "case_id": "research_rag_release_gate", "status": "pass" }
]
}
```

## Eval Report Example

| Eval                                  | What It Checks                                                     | Blocking Rule                               |
|---------------------------------------|--------------------------------------------------------------------|---------------------------------------------|
| <code>current_source_used</code>      | <code>refund-policy-v4</code> reaches the context packet           | no answer without current approved evidence |
| <code>stale_source_rejected</code>    | <code>refund-policy-v2</code> stays out of the context packet      | no stale policy answer                      |
| <code>forbidden_source_omitted</code> | <code>finance-private-notes</code> stays out of the context packet | no forbidden source in context              |
| <code>citation_faithfulness</code>    | cited sources support the answer                                   | no unsupported citation                     |
| missing evidence refusal              | agent refuses when approved evidence is absent                     | no fabricated answer                        |
| memory write                          | memory writes follow retention rules                               | no sensitive memory write                   |

Example fixture:

```
{
 "case_id": "stale_refund_policy_rejected",
 "question": "Can the agent issue refunds directly?",
 "retrieved_sources": ["refund-policy-v2", "refund-policy-v4"],
 "expected": {
 "must_cite": ["refund-policy-v4"],
 "must_not_cite": ["refund-policy-v2"],
 "answer_contains": "may draft but not issue refunds"
 }
}
```

## ADR Example

```
ADR-022: Research agent answers only from approved current sources

Status

Accepted
```

## ## Decision

The research agent may answer only from sources that pass access control, freshness, and citation checks. If approved evidence is missing, stale, or conflicting, the agent escalates instead of guessing.

## ## Rollback

Disable answer synthesis and keep retrieval-only search results available while the source filter or citation evaluator is repaired.

## Runbook Example

```
service: research-rag-agent
owner: knowledge-platform
kill switch: disable answer synthesis
fallback: return ranked source list only
trace dashboard: knowledge/research-agent/traces
eval suite: evals/research-rag
incident trigger: unsupported answer, forbidden source exposure, stale source citation
post-incident action: add retrieval fixture and citation eval before re-enable
```

## Release Checklist

- Source ACLs run before context assembly.
- Retrieved content is separated from instructions.
- Answers cite only evidence in the context packet.
- Missing evidence produces refusal or escalation.
- Memory writes have retention, deletion, and correction rules.
- Evals cover stale, forbidden, missing, and conflicting sources.

## Native Slice Completion Standard

The native LangGraph slice is complete when it proves these outcomes:

| Requirement                                                 | Evidence                                                 |
|-------------------------------------------------------------|----------------------------------------------------------|
| source access runs before retrieval output enters context   | <code>check_access</code> node and policy trace          |
| stale and forbidden sources stay out of the evidence packet | <code>filter_sources</code> node and omitted source list |

| Requirement                                 | Evidence                                                               |
|---------------------------------------------|------------------------------------------------------------------------|
| answer cites only approved current evidence | <code>answer_with_citations</code> output and citation eval            |
| missing approved evidence escalates         | conditional edge to <code>escalate</code>                              |
| release gate blocks bad grounding           | <code>evaluate_answer</code> failures for stale or forbidden citations |

The slice should remain small. Its job is to prove source policy, context assembly, citation faithfulness, and escalation before adding real vector stores or model calls.

### Related Labs

- [Lab 03 - Agentic RAG](#)
- [Lab 06 - Observability and Evals](#)
- [Lab 11 - Context, Memory, Trace, and Evals](#)
- [Lab 12 - LangGraph State Graph](#)



# Capstone - Multi-Agent Delivery Workflow

Build a workflow that coordinates specialist agents to plan, review, and package a delivery artifact while preserving one accountable owner for final acceptance.

This capstone is coordination heavy. The lesson is that multiple agents do not remove the need for workflow ownership. They increase the need for it.

## Problem

A team wants an agentic workflow that turns a product request into a reviewed delivery package: requirements summary, implementation plan, risk review, test plan, and final release note. Specialist agents can help, but the workflow must prevent duplicated work, conflicting outputs, unclear authority, and unreviewable transcripts.

## Non-Goals

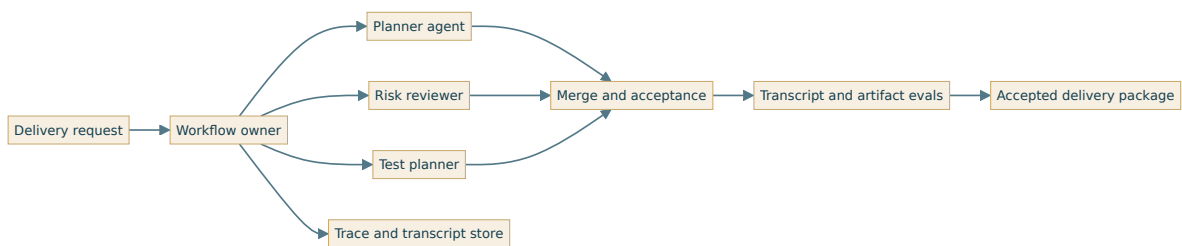
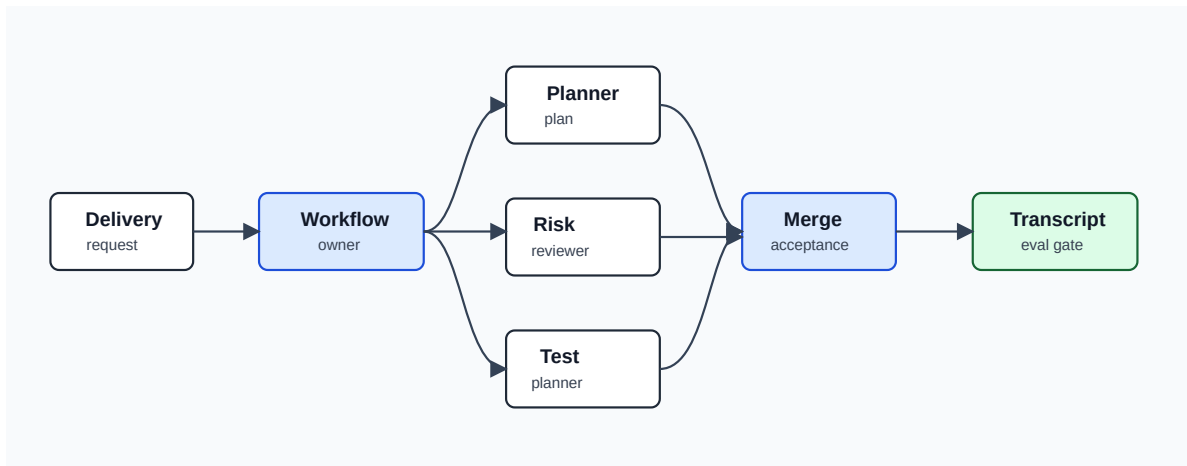
- Do not let agents merge their own outputs without a final owner.
- Do not treat role names as specialization.
- Do not use chat history as the only state store.
- Do not allow tools without role-specific permissions.

## Pattern Composition

| Concern            | Pattern                 |
|--------------------|-------------------------|
| decomposition      | Task Delegation         |
| coordination       | Supervisor / Worker     |
| role workflow      | CrewAI Flows and Crews  |
| transcript review  | Observability and Evals |
| durable state      | Durable Workflows       |
| production runtime | Deployment Walkthrough  |

## Architecture

Read this diagram as an accountability boundary. Specialist agents contribute work, but the workflow owner keeps state, merges outputs, runs evals, and accepts the final package.



## Runnable Assets

Run the deterministic capstone implementation:

```
npm run capstones:demo
npm run capstones:test
```

Inspect:

- `capstone-projects-runtime/typescript/src/capstones.ts`
- `capstone-projects-runtime/typescript/test/capstones.spec.ts`

Downloadable evidence:

- [Sample trace JSON](#)
- [Sample eval report](#)
- [Capstone review scorecard](#)
- [Framework selection ADR template](#)
- [Production readiness worksheet](#)

Expected runtime signal:

```
multi-agent-delivery-workflow: pass
 stop: accepted_after_review
 trace events: 4
```

The test suite treats these as release evidence:

| Evidence                              | Runtime Check                               |
|---------------------------------------|---------------------------------------------|
| Planner output exists                 | <code>planner_present</code>                |
| Risk review exists                    | <code>risk_review_present</code>            |
| Test plan exists                      | <code>test_plan_present</code>              |
| Turns are ordered                     | <code>turns_sequential</code>               |
| Workflow owner accepts last           | <code>final_owner_accepts_last</code>       |
| Release gate emits one final decision | <code>delivery_workflow_release_gate</code> |

## Role Contracts

| Role           | Input                | Output                       | Cannot Do               |
|----------------|----------------------|------------------------------|-------------------------|
| Planner        | request, constraints | scoped implementation plan   | approve final release   |
| Risk reviewer  | request, plan        | risks, mitigations, blockers | rewrite plan silently   |
| Test planner   | request, plan        | test matrix and gates        | lower release threshold |
| Workflow owner | all outputs          | final accepted package       | ignore failed evals     |

Every role needs a reason to exist. If a role does not change the output or risk profile, remove it.

## Capstone Review Gate

Before treating this capstone as production-grade, verify the accountability boundary:

| Check                          | Evidence                                                                  |
|--------------------------------|---------------------------------------------------------------------------|
| One owner accepts final output | Workflow owner, not a worker agent, sets final acceptance.                |
| Roles are distinct             | Planner, reviewer, and tester have different inputs, outputs, and limits. |
| Transcript is normalized       | Messages include role, turn, type, task, stop reason, and eval result.    |
| Missing review blocks release  | Risk review and test plan are required before acceptance.                 |
| Delegation can be disabled     | Rollback routes work to a single-owner checklist workflow.                |

Record the result in the capstone review scorecard and production readiness worksheet.

## Production Bridge

Use this table when turning the capstone into a service:

| Capstone Artifact  | Production Version                                                                    |
|--------------------|---------------------------------------------------------------------------------------|
| Role contracts     | Versioned role schemas with permissions, tool scopes, timeouts, and expected outputs. |
| Workflow owner     | Durable acceptance step with final owner, stop reason, and rollback control.          |
| Transcript example | Redacted transcript store with replay, retention, and incident links.                 |
| Transcript evals   | Blocking gates for role coverage, turn order, final owner, and ignored blockers.      |
| Runbook            | Kill switch for delegation plus fallback checklist path.                              |

The first production milestone is a delivery workflow that can reject incomplete collaboration and prove who accepted the final package.

### Native Framework Mapping

Start with the deterministic TypeScript capstone, then compare the native slices:

- `native-framework-examples/crewai-delivery/` proves role separation and flow-owned acceptance before adding real project-management or repository tools.
- `native-framework-examples/autogen-delivery/` proves AgentChat team roles, termination, normalized transcript export, and transcript evals.

| Framework    | Best Mapping                                                                                          |
|--------------|-------------------------------------------------------------------------------------------------------|
| CrewAI       | Flow owns state and final acceptance. Crew agents produce planner, reviewer, and tester outputs.      |
| AutoGen      | AgentChat team records role turns and termination. Transcript evals check role order and stop reason. |
| LangGraph    | Nodes or subgraphs represent roles. Graph state stores each output and final acceptance.              |
| Mastra       | Workflow coordinates agents, tools, evals, and trace export inside a TypeScript runtime package.      |
| Mini-runtime | Supervisor dispatches tasks, validates worker outputs, merges results, and emits trace events.        |

### Trace And Transcript Example

```
{
 "trace_id": "tr_delivery_331",
```

```

"workflow_state": "accepted",
"messages": [
 { "turn": 1, "from": "workflow", "to": "planner", "type": "assignment" },
 { "turn": 2, "from": "planner", "to": "workflow", "type": "plan" },
 { "turn": 3, "from": "workflow", "to": "risk_reviewer", "type": "review_request" },
 { "turn": 4, "from": "risk_reviewer", "to": "workflow", "type": "risk_review" },
 { "turn": 5, "from": "workflow", "to": "test_planner", "type": "test_request" },
 { "turn": 6, "from": "test_planner", "to": "workflow", "type": "test_plan" },
 { "turn": 7, "from": "workflow", "to": "team", "type": "accepted_package" }
],
"evals": [
 { "case_id": "planner_present", "status": "pass" },
 { "case_id": "risk_review_present", "status": "pass" },
 { "case_id": "test_plan_present", "status": "pass" },
 { "case_id": "turns_sequential", "status": "pass" },
 { "case_id": "final_owner_accepts_last", "status": "pass" }
]
}

```

## Eval Report Example

| Case                     | Expected                                                     | Result   |
|--------------------------|--------------------------------------------------------------|----------|
| planner_present          | Planner returns an implementation plan.                      | pass     |
| risk_review_present      | Risk reviewer returns review output before acceptance.       | pass     |
| test_plan_present        | Test planner returns test gates before acceptance.           | pass     |
| turns_sequential         | Transcript turns are 1 through 7 with no gaps.               | pass     |
| final_owner_accepts_last | The workflow owner sends <code>accepted_package</code> last. | pass     |
| reviewer finds blocker   | Workflow stops or escalates.                                 | blocking |
| tester missing           | Final acceptance is blocked.                                 | blocking |

Blocking threshold:

```

required role coverage: 100%
final owner present: 100%
critical blocker ignored: 0
missing test gate accepted: 0

```

## ADR Example

```
ADR-023: Delivery workflow uses specialist agents with workflow-owned acceptance
```

```
Status
```

```
Accepted
```

```
Decision
```

```
The delivery workflow may delegate planning, risk review, and test planning to specialist agents. A workflow-owned acceptance step decides the final package. Worker agents cannot approve release, lower gates, or mutate final state directly.
```

```
Rollback
```

```
Disable multi-agent delegation and route requests to a single deterministic checklist workflow until transcript evals and role boundaries pass.
```

## Runbook Example

```
service: multi-agent-delivery-workflow
owner: platform-engineering
kill switch: disable delegation
fallback: single-owner delivery checklist
trace dashboard: platform/delivery-workflow/traces
eval suite: evals/delivery-workflow
incident trigger: final package accepted without risk review, test plan, or owner
post-incident action: add transcript regression fixture and update role contract
```

## Release Checklist

- Workflow state is separate from role chat.
- Every role has a typed input and expected output.
- Tool permissions are role-specific.
- Merge and acceptance are explicit workflow steps.
- Transcript evals verify order, role coverage, and stop reason.
- Rollback can disable delegation without disabling the whole delivery workflow.

## Related Labs

- [Lab 05 - Multi-Agent Supervisor](#)
- [Lab 08 - CrewAI Flows and Crews](#)
- [Lab 12 - LangGraph State Graph](#)

- [Lab 13 - AutoGen Transcript Evals](#)

Native examples:

- [native-framework-examples/crewai-delivery/](#) ([download](#))
- [native-framework-examples/autogen-delivery/](#) ([download](#))

# Deprecated / Historical Patterns

Deprecated patterns are preserved in the repository but removed from the active learning path.

They are either speculative, too broad, duplicated by stronger chapters, or currently too thin to stand as active patterns.

Use this page as a translation guide. Older agent terms still appear in posts, vendor decks, and early prototypes. The active book keeps the useful ideas but moves them into chapters with clearer boundaries, code, and production checks.

Download the reusable worksheet: [historical pattern migration record](#).

## Why This Section Exists

Historical terms are not useless. They often point at real needs: coordination, memory, tool access, delegation, or adaptation. The problem is that many older labels name an aspiration instead of a design.

An online book should help readers translate that aspiration into something testable. Treat this section as a glossary with judgment:

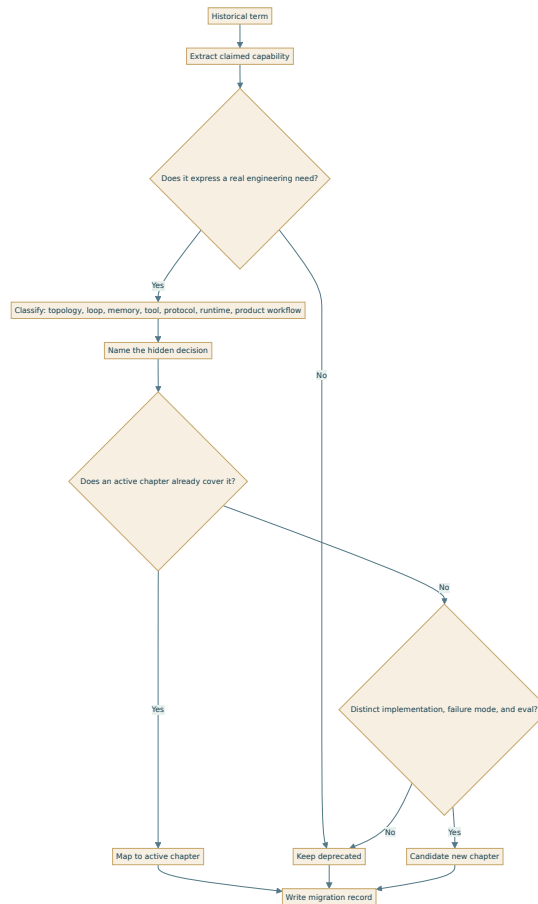
- preserve the useful idea;
- reject the vague label;
- map the idea to the current chapter that gives boundaries, code, evals, and production checks.

Do not cite a historical term as architecture by itself. Use it only as a clue that a more precise design decision is hiding underneath.

## Migration Decision Flow

Use this flow when an old term appears in a roadmap, design review, or vendor comparison. The goal is not to preserve the label; the goal is to recover the engineering decision behind it.





## Migration Map

| Historical Term      | Why It Was Deprecated                                                                       | Read Instead                                                                                                                    |
|----------------------|---------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| Agent Marketplace    | Too speculative without protocol, trust, pricing, permissions, and quality controls.        | <a href="#">A2A Agent Interoperability</a> , <a href="#">MCP-first Tool Use</a> , <a href="#">Choosing Multi-Agent Topology</a> |
| Agent Swarm          | The term hides the actual topology and often implies uncontrolled coordination.             | <a href="#">Parallel Agents</a> , <a href="#">Choosing Multi-Agent Topology</a> , <a href="#">Resource-Aware Agent Design</a>   |
| Hybrid Agent         | Too broad; almost every useful system combines model calls, tools, retrieval, and software. | <a href="#">Agentic System Architecture</a> , <a href="#">Reference Architecture</a>                                            |
| Meta-Cognitive Agent | Useful ideas belong in concrete reflection, evaluation, and improvement loops.              | <a href="#">Reflection</a> , <a href="#">Evaluator-Optimizer</a> , <a href="#">Self-Improvement</a>                             |
| Recursive Agent      | Recursion is an implementation technique, not a production pattern by itself.               | <a href="#">Planning and Execution</a> , <a href="#">Goals and State</a> , <a href="#">Task Delegation</a>                      |

| Historical Term                  | Why It Was Deprecated                                                                   | Read Instead                                                                                                            |
|----------------------------------|-----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| Distributed Agent                | Too vague compared with explicit protocols, workflows, and ownership boundaries.        | <a href="#">A2A Agent Interoperability</a> , <a href="#">Durable Workflows</a> , <a href="#">Reference Architecture</a> |
| API Integration Copilot          | Better treated as an applied tool-use and workflow example.                             | <a href="#">Tool Capability Design</a> , <a href="#">MCP-first Tool Use</a> , <a href="#">Human Approval Gates</a>      |
| Data Pipeline Orchestrator Agent | Better taught through durable workflows, state, retries, and observability.             | <a href="#">Durable Workflows</a> , <a href="#">Observability and Evals</a> , <a href="#">Deployment Walkthrough</a>    |
| Multi-Modal Tool-Using Agent     | Archived until the repository has a developed multimodal implementation and eval suite. | <a href="#">Tool Capability Design</a> , <a href="#">Production Evaluation Feedback Loops</a>                           |

## Worked Translations

Use these examples when a design review starts with an older term.

| Claim                                       | Better Translation                                                                                   | Design Question                                                                 | Current Evidence Needed                                              |
|---------------------------------------------|------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------|----------------------------------------------------------------------|
| "We need an agent swarm for research."      | Run parallel specialist agents only if independent work can be merged safely.                        | What can run independently, who merges results, and how are conflicts resolved? | Parallel trace, merge rubric, cost budget, and disagreement eval.    |
| "We need a recursive agent for planning."   | Use bounded planning with explicit depth, stop reasons, and plan validation.                         | What is the maximum planning depth, and what rejects a bad plan?                | Plan schema, stop condition, invalid-plan tests, and replay trace.   |
| "We need a distributed agent architecture." | Separate services, queues, workflow state, tool gateways, and agent-to-agent messages.               | Which component owns state, policy, retry, and final acceptance?                | Service diagram, durable workflow record, A2A envelope, and runbook. |
| "We need a meta-cognitive agent."           | Add an evaluator or reflection step only where it improves measurable outcomes.                      | What does the reviewer check that the first pass cannot check?                  | Rubric, before/after eval, stop rule, and regression case.           |
| "We need an API copilot."                   | Build a narrow tool-using workflow with permissioned API calls and human approval for risky actions. | Which API action can execute without a human, and which must be approved?       | Tool manifest, approval record, auth test, and audit log.            |

The translation should make the system less mystical and easier to review. If the clearer version sounds ordinary, that is usually a good sign.

## Replacement Categories

Most deprecated terms map to one of five current design categories.

| Old Language Usually Means                            | Current Category                       | Start Here                                                     |
|-------------------------------------------------------|----------------------------------------|----------------------------------------------------------------|
| “Swarm”, “society”, “collective”, “crew”              | Multi-agent topology                   | <a href="#">Choosing Multi-Agent Topology</a>                  |
| “Recursive”, “self-planning”, “autonomous loop”       | Control loop                           | <a href="#">Planning and Execution</a>                         |
| “Meta-cognitive”, “self-critiquing”, “self-improving” | Evaluation loop                        | <a href="#">Reflection, Evaluator-Optimizer</a>                |
| “Distributed”, “marketplace”, “agent network”         | Protocol and service boundary          | <a href="#">A2A Agent Interoperability, Agents As Services</a> |
| “Copilot”, “orchestrator”, “assistant for X”          | Product workflow with tool permissions | <a href="#">Tool Capability Design, Human Approval Gates</a>   |

If a term does not fit one category, split it until it does. A term that mixes topology, memory, tools, policy, and UX should become several design decisions, not one pattern.

## Legacy Term Triage Scorecard

Use this scorecard before adding an old term to a roadmap, slide, or architecture document. A low score means the term should stay deprecated. A medium score means the idea should map to an existing chapter. A high score means the term might deserve new pattern work.

| Criterion      | 0 Points                      | 1 Point                      | 2 Points                                             |
|----------------|-------------------------------|------------------------------|------------------------------------------------------|
| Intent         | vague aspiration              | clear user or system need    | distinct need not covered elsewhere                  |
| Boundary       | no owner or state boundary    | partial owner or boundary    | explicit owner, state, tools, policy, and acceptance |
| Implementation | demo only or description only | one plausible implementation | runnable implementation with contracts               |
| Failure modes  | not named                     | generic risks                | specific failures and avoid cases                    |
| Evaluation     | no eval                       | informal examples            | repeatable evals and negative cases                  |

| Criterion  | 0 Points            | 1 Point                    | 2 Points                                           |
|------------|---------------------|----------------------------|----------------------------------------------------|
| Operations | no production story | partial logging or runbook | observability, rollback, budget, and incident path |

Interpret the score:

- 0-4: keep deprecated.
- 5-8: map to existing chapters.
- 9-12: candidate new chapter, but only after code and evals exist.

This scorecard protects the book from vocabulary inflation. A term earns its way into the active path by making engineering decisions clearer.

## How To Read Old Agent Literature

When an older article uses one of these terms, translate it into production questions:

1. What topology does the author actually mean?
2. Who owns state, tools, policy, memory, and final acceptance?
3. Which step is deterministic software, and which step is model judgment?
4. What evidence, trace, or eval would prove the pattern works?
5. Which current chapter teaches the same idea with a clearer boundary?

This keeps old language useful without letting vague labels drive architecture.

## Migration Review Workflow

Use this workflow when a team brings an old pattern name into an architecture review, roadmap, or design document.

1. Classify the claim: topology, control loop, tool boundary, memory, knowledge, runtime, operations, product workflow, or domain packaging.
2. Name the decision hidden by the label: ownership, state, policy, evidence, eval, observability, cost, latency, or human approval.
3. Map the claim to the strongest current chapter.
4. Decide whether the current chapter is enough.
5. Promote a new pattern only if the old term exposes a distinct intent, failure mode, implementation shape, and evaluation method.

Most historical terms fail at step 2. They sound architectural, but they do not tell a reader what to build, test, observe, or reject.

## Do Not Promote A Term Just Because

| Weak Signal                                           | Why It Is Not Enough                                                                    |
|-------------------------------------------------------|-----------------------------------------------------------------------------------------|
| It appears in vendor decks.                           | Marketing language often compresses several engineering decisions into one vague label. |
| It has many search results.                           | Popular vocabulary can still be too broad for a book chapter.                           |
| A demo exists.                                        | A demo proves possibility, not repeatability, boundaries, or production fitness.        |
| It maps to several chapters.                          | If it maps everywhere, it is probably a packaging term rather than a pattern.           |
| It sounds more advanced than the replacement chapter. | Better chapters make the decision clearer, not louder.                                  |

## Migration Record

Use this short record when translating a deprecated term into the active book.

```

legacy_term:
source:
claimed_capability:
actual_decision_needed:
current_chapter:
ownership_boundary:
required_evidence:
promotion_decision: keep_deprecated | map_to_existing | candidate_new_chapter
reason:

```

The record should make the decision auditable. A reader should be able to see why the term stayed deprecated, where the useful idea moved, and what evidence would change the decision.

Download the full version: [historical pattern migration record](#).

## Completed Example

```

legacy_term: Agent Swarm
source: internal roadmap proposal, Q3 platform planning
claimed_capability: many agents collaborate to research incidents faster than one analyst
actual_decision_needed: whether independent investigation branches can run in parallel and
be merged safely
current_chapter: Multi-Agent Systems / Parallel Agents
ownership_boundary: orchestrator owns fan-out, analysts own scoped findings, aggregator
owns merge, human incident lead owns final acceptance
required_evidence: parallel trace, merge rubric, cost budget, conflict-handling eval,

```

```
escalation rule
promotion_decision: map_to_existing
reason: the useful idea is bounded parallel investigation; the swarm label hides merge
policy, budget, and final ownership
```

This is the standard: the migration record should replace the old label with a decision another engineer can review.

## Promotion Criteria

A deprecated pattern can return to the active learning path only when it has:

- a crisp intent that differs from existing chapters;
- a runnable reference implementation;
- a concrete code walkthrough;
- eval cases that prove when to use it and avoid it;
- production guidance for state, policy, tools, observability, and failure handling;
- links to templates or checklists readers can reuse.

Without those artifacts, the term stays here as historical vocabulary.

## Reader Value

This chapter exists to reduce confusion. Readers should leave knowing which modern chapter replaces each older term and why the book avoids vague pattern names.

Source archive: `deprecated`

# Publishing and Releases

The book is published for reading on the Astro GitHub Pages site. PDF and EPUB files are courtesy companion formats for offline reading; the online reader is the primary product.

## Public URLs

- Book site: <https://gturitto.github.io/Agentic-Systems-Patterns/>
- Courtesy PDF: <https://gturitto.github.io/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.pdf>
- Courtesy EPUB: <https://gturitto.github.io/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.epub>
- Repository: <https://github.com/GTuritto/Agentic-Systems-Patterns>
- Discussions: <https://github.com/GTuritto/Agentic-Systems-Patterns/discussions>

## Release Artifacts

The checked-in courtesy formats live at:

```
book/releases/Agentic-Systems-Patterns.pdf
book/releases/Agentic-Systems-Patterns.epub
```

The GitHub Pages deployment publishes the same files at:

```
/releases/Agentic-Systems-Patterns.pdf
/releases/Agentic-Systems-Patterns.epub
```

Each `Publish Book` workflow run also uploads the courtesy formats as a workflow artifact. Use that artifact when you need to inspect the exact PDF or EPUB produced by a specific CI run.

Before tagging or announcing a release, use the [Release Readiness Checklist](#). Use [Release Notes](#) as the reader-facing summary of what changed and what evidence supports the release. Use [GitHub Discussions](#) as the canonical public feedback channel for reader questions, chapter feedback, and release follow-up.

Download the reusable publishing artifact: [release evidence record](#).

## Local Publishing Commands

From the repository root:

```
npm test
npm run release:commands
```

```
npm run typecheck
npm run capstones:evidence
npm run native-examples:validate
npm run native-examples:smoke:langgraph
npm run book:manifest:test
npm run book:visuals:verify
npm run book:quality
npm run book:build
npm run site:build
npm run site:parity
npm run book:pdf
npm run book:epub
```

These commands cover runnable examples, release command parity, TypeScript contracts, capstone evidence alignment, native framework slices, generated chapters, editorial consistency, visual coverage, diagram assets, site routes, internal links, and the courtesy PDF/EPUB downloads.

Use `npm run site:dev` for the primary local reader preview.

Use `npm run book:start` only for the VitePress authoring preview.

The main outputs are:

```
site/dist
book/releases/Agentic-Systems-Patterns.pdf
book/releases/Agentic-Systems-Patterns.epub
```

`site/dist` is deployed to GitHub Pages. The Astro build uses the base path `/Agentic-Systems-Patterns/`.

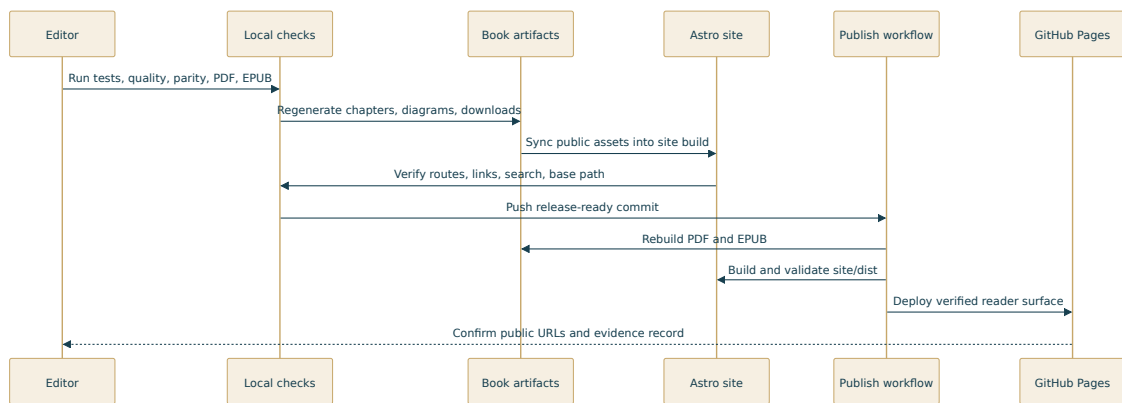
The lower-level book pipeline commands are:

```
npm run book:content
npm run book:content:verify
npm run book:diagrams
npm run book:diagrams:verify
```

Run lower-level commands only when you are editing generated chapters, source bundles, editorial metadata, or diagram exports directly. A release should still pass the full command set above.

Use this flow as the publishing model. The online book is the release surface; PDF and EPUB are generated companions that must match the same verified content.





## GitHub Pages Release Gate

Use this gate before announcing the online book:

| Check                           | Evidence                                                                                                                                  |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Site base path is correct       | Built links resolve under <code>/Agentic-Systems-Patterns/</code> .                                                                       |
| Reader routes exist             | <code>site:parity</code> reports all manifest chapters and section pages.                                                                 |
| Download assets exist           | Courtesy PDF, courtesy EPUB, source bundles, trace examples, eval reports, and worksheets are present under <code>site/dist</code> .      |
| Book quality gates pass         | <code>book:quality</code> verifies manifest coverage, generated content, editorial consistency, diagram assets, and Mermaid SVG coverage. |
| Capstone evidence agrees        | <code>capstones:evidence</code> verifies capstone chapters, runtime output, trace assets, eval reports, and scorecard links.              |
| Search is generated             | Pagefind indexes the built site without build errors.                                                                                     |
| Courtesy formats match the site | <code>book:pdf</code> and <code>book:epub</code> run after content changes and the deploy copies are refreshed.                           |
| Release notes are current       | Review date, scope, known boundaries, and verification evidence match the release.                                                        |

Do not treat GitHub Pages as a static file dump. Treat it as the product surface: routes, downloads, search, PDF, navigation, and release notes must agree.

## Reader Surface Checklist

Before sharing the public URL, inspect the site as a reader would:

| Surface          | What To Check                                                                                         | Why It Matters                                     |
|------------------|-------------------------------------------------------------------------------------------------------|----------------------------------------------------|
| Homepage         | Primary action opens the book and secondary actions resolve.                                          | New readers need a clear start.                    |
| Sidebar          | Sections match the logical groups and current manifest.                                               | Readers need orientation across 100+ chapters.     |
| Search           | A query such as <code>approval</code> , <code>RAG</code> , or <code>eval</code> returns useful pages. | GitHub Pages readers use the book as a reference.  |
| Courtesy formats | The PDF and EPUB download and reflect the current site content.                                       | Offline readers should not receive stale guidance. |
| Templates        | Worksheets and checklists download from public paths.                                                 | The book's value depends on reusable artifacts.    |
| Source bundles   | Pattern and lab downloads resolve.                                                                    | Engineers need working code beside the prose.      |
| Release notes    | Current version, scope, evidence, and limits are explicit.                                            | Public claims need proof and boundaries.           |

If a surface fails, fix the site or document the limitation before announcing the release.

## Deployment

Deployment is handled by:

```
.github/workflows/publish-book.yml
```

The workflow runs on every push to `main` and can also be triggered manually with `workflow_dispatch`. It checks release command parity, capstone evidence, the book manifest, editorial content, visual coverage, diagram assets, and Mermaid SVG coverage, builds the courtesy PDF and EPUB, validates the VitePress authoring build, builds the Astro site, runs Astro parity/link checks, and deploys `site/dist`.

## Release Evidence Record

For each public release, keep a short evidence record:

```
version:
date:
commit:
site url:
pdf url:
epub url:
commands passed:
```

visual pages checked:

known limitations:

rollback action:

This record is useful when a reader reports a broken page, stale PDF, missing download, or mismatch between the site and repository.

Use the downloadable [release evidence record](#) when the release changes content, assets, routes, courtesy format generation, search, or publishing workflow behavior. For the current release, start from the filled [pre-launch release evidence for 2026-06-21](#) and add the public URL checks after deployment.

## License

Source code and runnable examples are licensed under the MIT License. Book/reference content, diagrams, worksheets, and generated publishing artifacts are licensed under [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International](#) ([CC-BY-NC-SA-4.0](#)).

When reusing or adapting the content, preserve attribution, use it only for non-commercial purposes unless separate permission is granted, and distribute adaptations under the same license.

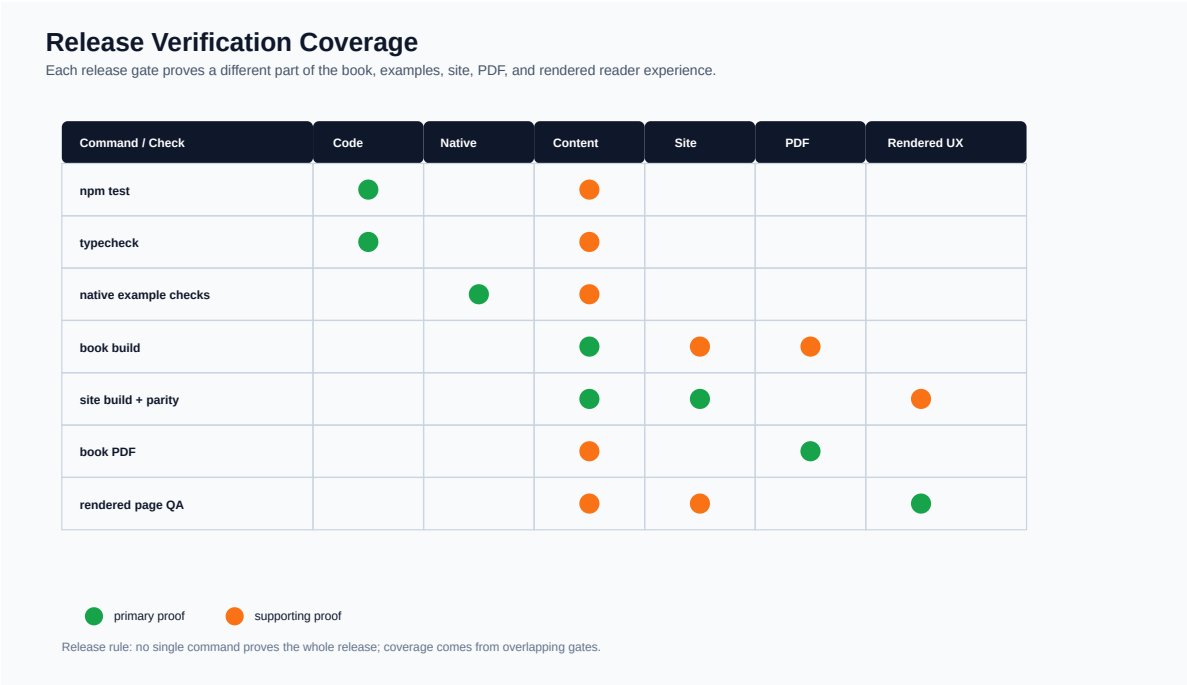
# Release Readiness Checklist

Use this checklist before publishing a new version of the book. The standard is not “the site builds.” The standard is that a reader can move from concept, to pattern choice, to lab, to capstone, to release evidence without finding missing context.

Download the reusable publishing artifact: [release evidence record](#).

## Verification Coverage

Use this graph to see why release readiness needs overlapping checks. Code tests, native example checks, content builds, site parity, courtesy format generation, and rendered-page QA each prove a different part of the release.



## Reader Journey

| Gate                                   | Release Evidence                                                                                        |
|----------------------------------------|---------------------------------------------------------------------------------------------------------|
| Start path is coherent                 | <a href="#">How To Read This Book</a> gives first-time, builder, lab, capstone, and reference paths.    |
| Pattern selection is usable            | Selection chapters explain when to use a pattern, when to avoid it, and how patterns compose.           |
| Labs prove architecture, not only APIs | Labs identify language, framework, source files, baseline command, production gap, and expected output. |

| Gate                                         | Release Evidence                                                                                                     |
|----------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| Mini-framework track explains the primitives | The from-scratch track shows loop, decision, tool registry, policy, memory, trace, and eval responsibilities.        |
| Capstones feel product-shaped                | Capstones include state, policy, memory, approvals, traces, evals, ADRs, runbooks, rollback, and framework mappings. |

## Content Quality

| Gate                       | Release Evidence                                                                                                             |
|----------------------------|------------------------------------------------------------------------------------------------------------------------------|
| No unfinished markers      | Search for common draft markers outside generated assets.                                                                    |
| Diagrams have context      | Architecture diagrams are introduced by the ownership boundary or decision they explain.                                     |
| Tables are not orphaned    | Tables have enough context before and after them to explain how the reader should use the rows.                              |
| Examples name their limits | Demo code states what production still needs: state, policy, tracing, evals, approval, deployment, or framework integration. |
| Terminology is stable      | State, policy, memory, tools, traces, evals, workflows, and approvals mean the same thing across chapters.                   |

## Chapter A++ Spot Check

Before publishing, select at least one chapter from each changed section and check it against the book's editorial bar. A chapter is release-ready when it gives the reader a decision, boundary, test, or reusable artifact, not just more information.

| Check                            | Release Evidence                                                                   |
|----------------------------------|------------------------------------------------------------------------------------|
| Reader promise is clear          | The first screen says what problem the chapter solves and why it matters.          |
| Use and avoid cases are explicit | The chapter helps a reader choose the pattern or reject it.                        |
| Ownership is visible             | State, tools, policy, memory, approvals, evals, and stop conditions have owners.   |
| Failure modes are concrete       | The chapter names realistic ways the design can fail and how to detect them.       |
| Production gap is named          | The chapter explains what changes before real users, data, money, or side effects. |

| Check                    | Release Evidence                                                                                |
|--------------------------|-------------------------------------------------------------------------------------------------|
| Evaluation is actionable | The reader can turn the guidance into a test, eval case, fixture, or release gate.              |
| Reuse artifact exists    | The chapter leaves a checklist, schema, trace shape, worksheet, ADR, or implementation pattern. |
| Online UX works          | Headings, tables, diagrams, code blocks, links, and downloads are readable in the built site.   |

If a changed chapter fails more than two checks, do not treat it as an A++ change. Revise the chapter or record the limitation in the release notes.

## Verification Commands

Run these from the repository root:

```
npm test
npm run release:commands
npm run typecheck
npm run capstones:evidence
npm run native-examples:validate
npm run native-examples:smoke:langgraph
npm run book:manifest:test
npm run book:visuals:verify
npm run book:quality
npm run book:build
npm run site:build
npm run site:parity
npm run book:pdf
npm run book:epub
```

Expected evidence:

| Command                    | What It Proves                                                                                    |
|----------------------------|---------------------------------------------------------------------------------------------------|
| npm test                   | Deterministic pattern examples, labs, capstones, evidence gates, and protocol examples still run. |
| npm run release:commands   | Package scripts, release docs, and the publish workflow list the same release gates.              |
| npm run typecheck          | TypeScript examples and shared contracts still compile.                                           |
| npm run capstones:evidence | Capstone chapters, runtime stop reasons, trace assets, eval reports, and scorecard links agree.   |

| Command                                              | What It Proves                                                                                                                                         |
|------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>npm run native-examples:validate</code>        | Native framework example files are syntactically valid and required assets exist.                                                                      |
| <code>npm run native-examples:smoke:langgraph</code> | LangGraph native slices install optional dependencies and execute without provider keys.                                                               |
| <code>npm run book:manifest:test</code>              | Sidebar, PDF manifest, chapter ownership, and generated chapter registration are valid.                                                                |
| <code>npm run book:visuals:verify</code>             | Every manifest chapter and the homepage have a Mermaid diagram, image, or SVG reference.                                                               |
| <code>npm run book:quality</code>                    | Manifest coverage, generated chapters, title/H1 consistency, unfinished markers, diagram assets, diagram alt text, and Mermaid SVG coverage are valid. |
| <code>npm run book:build</code>                      | VitePress authoring build, generated pages, diagrams, and downloads are valid after quality gates pass.                                                |
| <code>npm run site:build</code>                      | Astro reader site builds with synced public assets and search index.                                                                                   |
| <code>npm run site:parity</code>                     | Published routes and internal links match the book manifest.                                                                                           |
| <code>npm run book:pdf</code>                        | The courtesy PDF and deploy copy can be regenerated.                                                                                                   |
| <code>npm run book:epub</code>                       | The courtesy EPUB and deploy copy can be regenerated.                                                                                                  |

## Visual QA

Inspect these pages in the built site before release:

1. `/book/intro/`
2. `/book/publishing/how-to-read/`
3. `/book/pattern-selection/choosing-the-right-pattern/`
4. `/book/agent-engineering-practice/cross-framework-decision-matrix/`
5. `/book/hands-on-labs/`
6. `/book/hands-on-labs/from-scratch-mini-framework/`
7. `/book/capstone-projects/`
8. `/book/systems-architecture/reference-architecture/`
9. `/book/publishing/release-notes/`

Check that diagrams render, headings fit, code blocks are readable, tables do not feel unexplained, and navigation keeps the reader oriented.

## Public URL QA

After deployment to GitHub Pages, inspect the public site with the production base path:

| Public URL                                                                                | Check                                                                     |
|-------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| <code>/Agentic-Systems-Patterns/</code>                                                   | Homepage renders and primary actions work.                                |
| <code>/Agentic-Systems-Patterns/book/intro/</code>                                        | Introduction loads from the deployed site.                                |
| <code>/Agentic-Systems-Patterns/book/publishing/how-to-read/</code>                       | Reader paths are reachable.                                               |
| <code>/Agentic-Systems-Patterns/book/pattern-selection/choosing-the-right-pattern/</code> | Core pattern selection page loads.                                        |
| <code>/Agentic-Systems-Patterns/book/hands-on-labs/</code>                                | Lab index loads and lab links resolve.                                    |
| <code>/Agentic-Systems-Patterns/book/capstone-projects/</code>                            | Capstone index loads and artifact links resolve.                          |
| <code>/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.pdf</code>              | Courtesy PDF downloads.                                                   |
| <code>/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.epub</code>             | Courtesy EPUB downloads.                                                  |
| <code>/Agentic-Systems-Patterns/pagefind/</code>                                          | Search assets exist, search returns results, and level/type filters work. |

Local build checks prove the deployable artifact. Public URL QA proves the deployed reader surface.

## Download And Asset QA

Check the reader-facing assets that make the online book useful:

| Asset          | Release Evidence                                                                                       |
|----------------|--------------------------------------------------------------------------------------------------------|
| Courtesy PDF   | <code>/releases/Agentic-Systems-Patterns.pdf</code> exists and reflects the current content.           |
| Courtesy EPUB  | <code>/releases/Agentic-Systems-Patterns.epub</code> exists and reflects the current content.          |
| Source bundles | Pattern, lab, native framework, and capstone downloads resolve from the built site.                    |
| Templates      | Worksheets, scorecards, and review checklists resolve under <code>/capstone-assets/templates/</code> . |



| Asset                  | Release Evidence                                                                                                                      |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| Completed examples     | Completed ADR, lab evidence, and production-readiness examples resolve under <code>/capstone-assets/templates/</code> .               |
| Trace examples         | Capstone trace JSON files resolve and match the chapter references.                                                                   |
| Eval reports           | Capstone eval report files resolve and match the chapter references.                                                                  |
| Capstone evidence gate | <code>npm run capstones:evidence</code> passes against chapter text, runtime output, trace assets, eval reports, and scorecard links. |
| Diagrams               | SVG diagrams render on chapter pages and are not orphaned.                                                                            |
| Search metadata        | Pagefind reports chapter filters and rendered search results show section, type, and level metadata.                                  |

This check matters because the book is an online product. A chapter can read well and still fail readers if its downloads or evidence artifacts are missing.

## Release Evidence Record

Before publishing, record:

```
version:
date:
commit:
commands passed:
visual pages checked:
download assets checked:
known limitations:
release owner:
rollback action:
```

Put the record in the release notes, GitHub release, or release PR description.

Use the downloadable [release evidence record](#) when the release changes reader-facing content or publishing behavior. The current release already has a filled [pre-launch release evidence record](#); complete its public GitHub Pages section after deployment.

## Release Decision

Do not publish if any of these are true:

- A required command fails.
- A core reader path contains a broken link or missing image.
- A lab points to source code that no longer exists.

- A capstone claims native framework coverage that has no matching example or clear scope.
- The courtesy PDF or EPUB is stale relative to the site content.
- Release notes do not say what changed and how it was verified.

Publish only when the release evidence is stronger than the claim made to readers.

# Release Notes

These notes summarize the current release shape of *Agentic Systems Patterns*. Use them with the [Release Readiness Checklist](#) before publishing.

Use the [release evidence record](#) to capture actual command output, public URL checks, asset checks, known limits, and rollback action for a release. The current filled local record is the [pre-launch release evidence for 2026-06-21](#).

## Current Release

Version: `1.0.0`

Review date: 2026-06-21

Release theme: turn the repository from a pattern catalog into a complete guide for designing, implementing, evaluating, and operating agentic systems across frameworks.

## Reader Value Added

- Clear reader paths for first-time readers, builders, lab users, capstone users, and reference users.
- Pattern chapters with consistent intent, use/avoid guidance, architecture, system shape, protocol, failure modes, eval strategy, production checklist, source links, and downloads.
- Framework-agnostic labs across Python and TypeScript.
- Coverage of LangChain/LangGraph-style retrieval, LangGraph-style state graphs, Mastra-style runtime packaging, AutoGen-style transcript evaluation, CrewAI-style flows, A2A, MCP, and deterministic custom runtimes.
- A from-scratch mini-framework track that explains what agent frameworks package under the hood.
- Product-shaped capstones for refund support, research RAG, and multi-agent delivery workflows.
- A running support refund case study that starts in the Introduction and early foundation chapters before returning in labs and capstones.
- More concrete pattern teaching examples for single-agent drafting, prompt-chain refund gates, and evaluator-optimizer scoring.
- Optional per-exercise time budgets across Labs 01-13 so readers can split lab work into shorter reviewable blocks.
- Deeper advanced examples for framework setup failures, refund approval incident-to-eval conversion, and domain-specific authority boundaries.
- Filled production-readiness examples for support refund, research RAG, and multi-agent delivery workflows.
- Native framework slices for selected LangGraph, Mastra, CrewAI, and AutoGen examples.
- Release-ready publishing flow for the GitHub Pages reader, courtesy PDF/EPUB, generated source bundles, diagrams, Pagefind search metadata and filters, parity checks, and native example validation.

## Verification Evidence

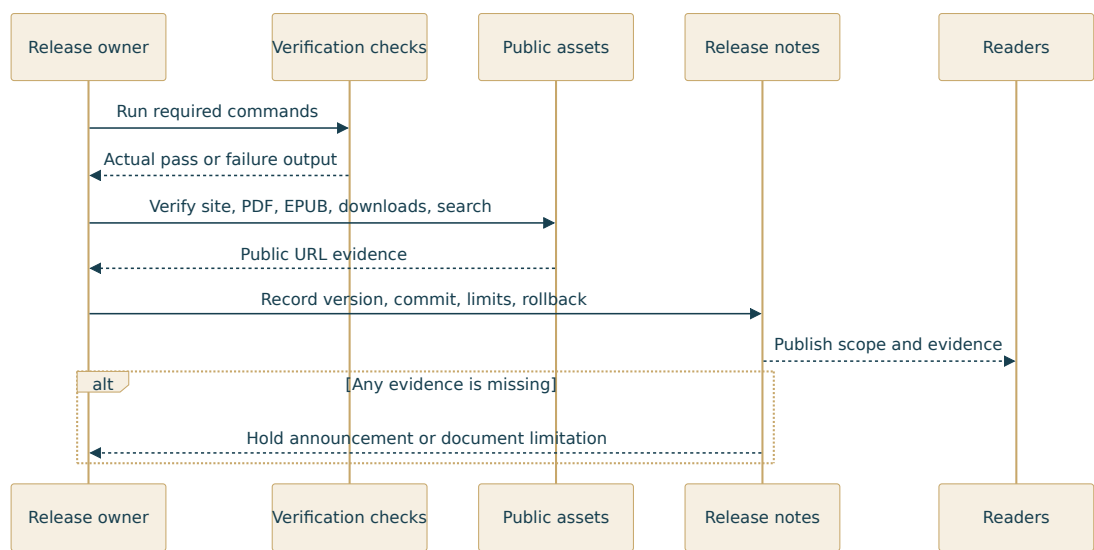
Before publishing, the release should pass:

```
npm test
npm run release:commands
npm run typecheck
npm run capstones:evidence
npm run native-examples:validate
npm run native-examples:smoke:langgraph
npm run book:manifest:test
npm run book:visuals:verify
npm run book:build
npm run site:build
npm run site:parity
npm run book:pdf
npm run book:epub
```

The release is not ready if any command fails, if visual coverage regresses, if capstone evidence drifts from runtime output, or if the generated courtesy PDF and EPUB are not refreshed after content changes.

Record the actual command results with the release. Expected evidence is not enough for a public tag or announcement.

Use this flow when turning these notes into a public release. A release note is valid only when it points to current evidence, public assets, and a rollback action.



## Evidence Record Template

For each public release, record:

| Field                     | Value |
|---------------------------|-------|
| Version                   |       |
| Date                      |       |
| Commit                    |       |
| Release owner             |       |
| Commands passed           |       |
| Public URLs checked       |       |
| Discussions checked       |       |
| Download assets checked   |       |
| Capstone evidence checked |       |
| Search checked            |       |
| Search filters checked    |       |
| Courtesy PDF checked      |       |
| Courtesy EPUB checked     |       |
| Known limitations         |       |
| Rollback action           |       |

Keep this record in the release PR, GitHub release, or appended release notes. Do not replace actual evidence with a planned checklist.

## Reader-Facing Asset Checks

Before publishing, verify these built-site paths:

| Asset         | Path                                                                          |
|---------------|-------------------------------------------------------------------------------|
| Online book   | <code>/Agentic-Systems-Patterns/</code>                                       |
| Discussions   | <code>https://github.com/GTuritto/Agentic-Systems-Patterns/discussions</code> |
| Courtesy PDF  | <code>/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.pdf</code>  |
| Courtesy EPUB | <code>/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.epub</code> |
| Templates     | <code>/Agentic-Systems-Patterns/capstone-assets/templates/</code>             |

| Asset                                   | Path                                                                                                  |
|-----------------------------------------|-------------------------------------------------------------------------------------------------------|
| Captured output examples                | <code>lab-and-capstone-command-output.txt</code>                                                      |
| Completed production readiness examples | <code>completed-production-readiness-examples.txt</code>                                              |
| Downloads                               | <code>/Agentic-Systems-Patterns/downloads/</code>                                                     |
| Search index and filters                | <code>/Agentic-Systems-Patterns/pagefind/</code> with section, type, level, and reader-path metadata. |

The release should not be announced until the GitHub Pages site, GitHub Discussions feedback channel, courtesy PDF, courtesy EPUB, downloads, templates, captured output examples, and Pagefind search index are all generated from the same content or verified against the same release.

### Known Scope Boundaries

- Examples are educational and deterministic by default; live model-provider integrations require local configuration.
- Native framework slices are comparison points for important boundaries, not exhaustive tutorials for every framework feature.
- The book favors architecture, production evidence, and design review discipline over API-by-API framework coverage.
- Historical pattern names remain in the deprecated section so older terminology can be mapped to the current taxonomy.

### Publishing Artifacts

- Site output: `site/dist`
- PDF source artifact: `book/releases/Agentic-Systems-Patterns.pdf`
- PDF deploy copy: `book/docs/public/releases/Agentic-Systems-Patterns.pdf`
- EPUB source artifact: `book/releases/Agentic-Systems-Patterns.epub`
- EPUB deploy copy: `book/docs/public/releases/Agentic-Systems-Patterns.epub`
- Generated downloads: `book/docs/public/downloads/`

### Release Summary

This release is ready when the reader can answer five questions after finishing the guide:

1. Which agentic pattern should I use, and why?
2. What owns state, policy, tools, memory, approvals, traces, and evals?
3. How do I run a small example and identify what is missing for production?
4. How does the same architecture map across frameworks and languages?
5. What evidence proves the system is safe enough to release?

## Post-Release Checks

After deployment, open the public GitHub Pages URL and verify:

- the homepage loads;
- chapter navigation works;
- search opens, returns metadata-rich results, and level/type filters work;
- the courtesy PDF downloads;
- the courtesy EPUB downloads;
- at least one worksheet, trace, eval report, captured output example, and source bundle downloads.

If any public asset fails, update the release notes with the limitation or roll back the announcement.