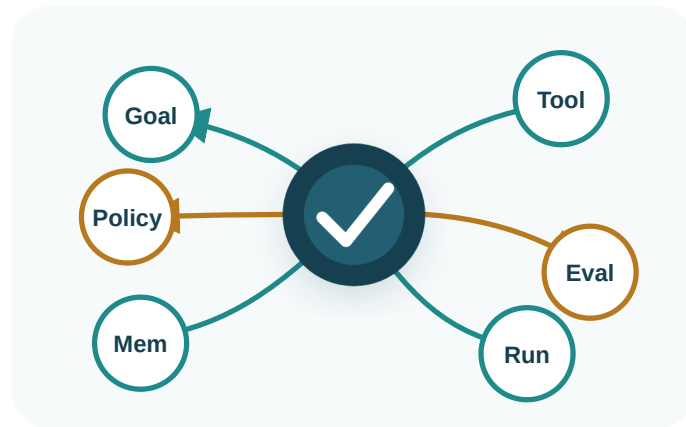




Agentic Systems Patterns

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/es/>

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

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

La mayoría de los agentic systems no fallan porque el model no sea lo suficientemente inteligente. Fallan porque la arquitectura que rodea al model es débil.

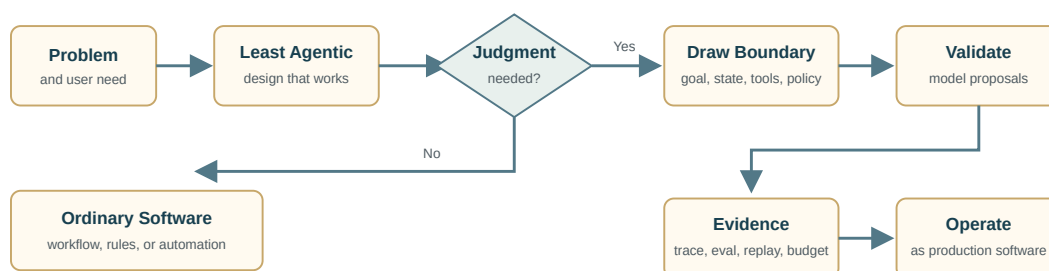
El goal es vago. El loop no tiene una condición real de detención. Los tools son demasiado poderosos. El state está oculto en una transcripción de chat. Nadie puede replay lo que ocurrió. El eval suite revisa la respuesta final pero no el camino que la produjo. La observability llega después del primer incidente. La autonomía se agrega antes de que el sistema la haya merecido.

Este libro trata de arreglar eso.

Está escrito para ingenieros de software, líderes técnicos, arquitectos y constructores que necesitan decidir qué debe ser responsabilidad de un agent, qué debe ser responsabilidad del software tradicional y qué nunca debe dejarse a un model. No es una lista de todos los nombres de agent patterns que pude encontrar. Es una guía práctica para elegir, componer, probar, asegurar y operar agentic patterns sin perder el control de ingeniería.

Un agent útil no es un prompt con ambición. Es software con límites.

Argumento de Arquitectura en Resumen

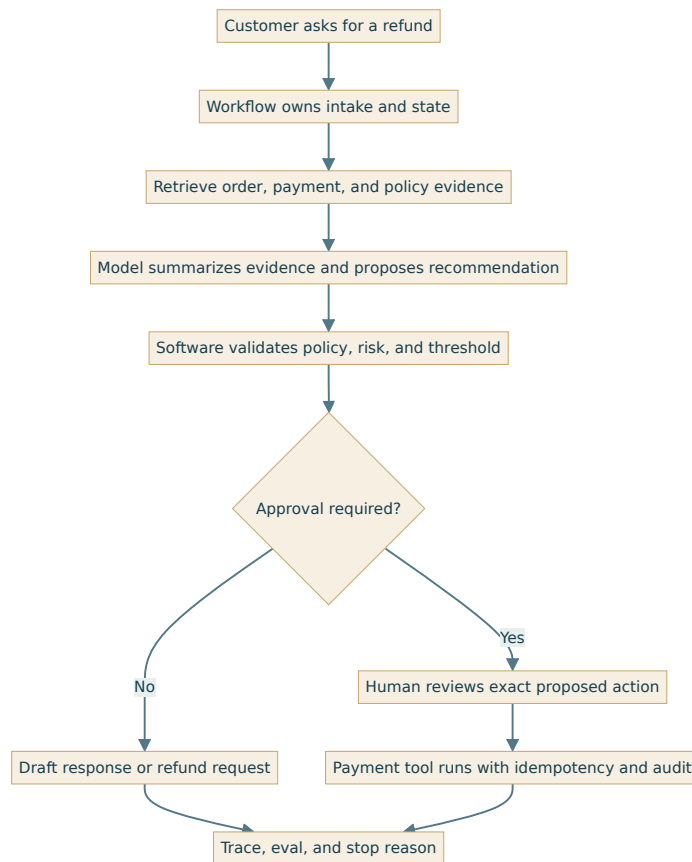


Usa este diagrama como la prueba central del libro: si un diseño omite los pasos de boundary, validación o evidencia, no está listo para producción.

Caso de Estudio en Ejecución

Varios capítulos regresan a un ejemplo con forma de producto: un asistente de reembolsos de soporte. Es lo suficientemente pequeño para entenderlo, pero lo suficientemente riesgoso para forzar una arquitectura real. Un refund workflow toca datos de clientes, evidencia de policy, payment tools, umbrales de aprobación, registros de auditoría y comunicación con el usuario. Eso lo convierte en una prueba útil para todo el libro.

El punto no es hacer que los reembolsos sean especiales. El punto es observar cómo un sistema pasa de ser software simple, a juicio acotado de model, a investigación agentic, a efectos secundarios con aprobación, hasta evidencia de producción.



Usa este caso como una pregunta recurrente: ¿qué debe decidir el model, qué debe ser responsabilidad del software y qué evidencia permitiría a un operador replay la ejecución después de que algo salga mal?

Contrato con el Lector

Cada capítulo debe ayudarte a tomar o revisar una decisión de ingeniería. Un capítulo sólido en este libro debe darte al menos uno de estos:

- un boundary que puedas definir;
- un pattern que puedas elegir o rechazar;
- un contrato que puedas implementar;
- un failure mode que puedas probar;
- una checklist que puedas usar en una revisión;
- un artifact que puedas reutilizar en un diseño, laboratorio, proyecto final o lanzamiento.

Si un capítulo solo enseña vocabulario, debe conectar ese vocabulario con una decisión. Si un capítulo muestra código, debe nombrar los controles de producción que aún faltan. Si un capítulo recomienda un pattern, también debe decir cuándo no usarlo.

Ese es el estándar de calidad para el libro en línea: los lectores deben irse con decisiones, evidencia y artifacts reutilizables, no solo con familiaridad con la terminología agent.

El Argumento

Mi argumento es simple:

1. Comienza con la arquitectura menos agentic que pueda cumplir el requerimiento.
2. Agrega juicio de model solo donde el software determinista no sea suficiente.
3. Trata las salidas del model como propuestas hasta que el software las valide.
4. Mantén goals, state, tools, memory y policy fuera del model, donde puedan ser inspeccionados.
5. Evalúa trayectorias, no solo respuestas finales.
6. Opera agents como sistemas de producción, con traces, presupuestos, reintentos, aprobaciones y revisión de incidentes.

El diseño agentic no se trata de hacer todo autónomo. Se trata de decidir exactamente dónde la autonomía ayuda, dónde crea riesgo y dónde el software tradicional debe seguir a cargo.

Qué Deberías Poder Hacer

Después de leer los capítulos principales, deberías poder:

1. Decidir si un problema necesita un prompt chain, workflow, single agent o multi-agent system.
2. Dibujar el boundary entre el juicio del model y el control determinista.
3. Definir el goal, state, tools, memory, policy, presupuesto y condiciones de detención para una ejecución agentic.
4. Revisar una superficie de tool para detectar autoridad excesiva antes de exponerla a un model.
5. Diseñar evals que prueben trayectorias, no solo respuestas finales.
6. Explicar cómo un sistema hará trace, replay, aprobará, reintentará, hará rollback y se recuperará.
7. Convertir un demo en un diseño de producción con controles explícitos de runtime.

Si un capítulo no te ayuda a tomar una de esas decisiones, no está haciendo suficiente trabajo.

Qué Cubre Este Libro

El libro sigue las decisiones que los ingenieros toman al construir sistemas reales:

- fundamentos: single agents, loops, goals, state, tools, structured outputs y context;
- selección de patterns: cuándo usar chains, routing, workflows, agents o multi-agent systems;
- práctica de ingeniería: ciclo de vida, elección de framework, seguridad, evaluación y confianza del usuario;
- control loops: planning, ReAct, reflection, evaluator-optimizer y recovery loops;
- memory y conocimiento: working memory, episodic memory, retrieval y boundaries de evidencia;
- tools, skills y protocolos: tool contracts, MCP, A2A, comunicación segura y approval gates;
- multi-agent systems: delegación, supervisión, debate, ejecución paralela y sistemas definidos por framework;
- arquitectura de sistemas: cómo los patterns se componen en productos desplegables;
- producción y runtime: durable workflows, observability, evaluación, policy, eventos y operaciones.

Los capítulos de patterns son intencionalmente consistentes. Están pensados para ser consultados durante el trabajo de diseño. Los capítulos que los rodean llevan el argumento: por qué un pattern pertenece a un sistema, qué cuesta y cómo saber si está funcionando.

Qué No Es Este Libro

Este no es un recetario de prompt engineering, un reporte de comparación de proveedores ni una afirmación de que los agents deban reemplazar al software tradicional. No asume que más autonomía es mejor. Tampoco trata los valores predeterminados de framework como arquitectura.

Los frameworks son útiles, pero no eliminan la necesidad de decidir dónde vive el state, qué tools están permitidas, cómo funcionan las aprobaciones, qué se tracea o cómo los fallos se convierten en pruebas de regresión. Esas son decisiones de producto y arquitectura.

Cómo Usarlo

Si eres nuevo en agentic systems, comienza con [Cómo Leer Este Libro](#), luego lee [¿Qué es un Agent?](#), [Arquitectura Antes que Autonomía](#), [Eligiendo el Pattern Correcto](#) y [De Patterns a Sistemas](#).

Si estás revisando un diseño de producción, comienza con los capítulos de selección y de ingeniería antes de leer las páginas de patterns individuales.

Si estás implementando, usa los laboratorios prácticos después de entender la arquitectura. Los ejemplos son deliberadamente pequeños. Las notas de producción muestran lo que debe cambiar antes de que esos ejemplos se conviertan en sistemas que puedan manejar state, policy, evals y observability.

Como este es un libro en línea, úsalo tanto como texto guiado como referencia. Sigue los caminos de lectura cuando estés aprendiendo el material. Usa la barra lateral, búsqueda, páginas de patterns, checklists, diagramas, laboratorios y proyectos finales cuando estés diseñando o revisando un sistema real.

Licencia

El código fuente y los ejemplos ejecutables están licenciados bajo la Licencia MIT. El contenido del libro/referencia, diagramas, hojas de trabajo y artifacts generados para publicación están licenciados bajo [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International](#) (CC-BY-NC-SA-4.0).

Cómo leer este libro

Este libro tiene dos objetivos.

Primero, plantea un argumento: los agentic systems necesitan arquitectura antes que autonomía. Segundo, te ofrece una referencia de patterns que puedes usar durante el trabajo de diseño.

No necesitas leer todos los capítulos en orden. Pero sí debes leer los capítulos de selección antes de agregar loops, tools, memory o múltiples agents a un sistema.

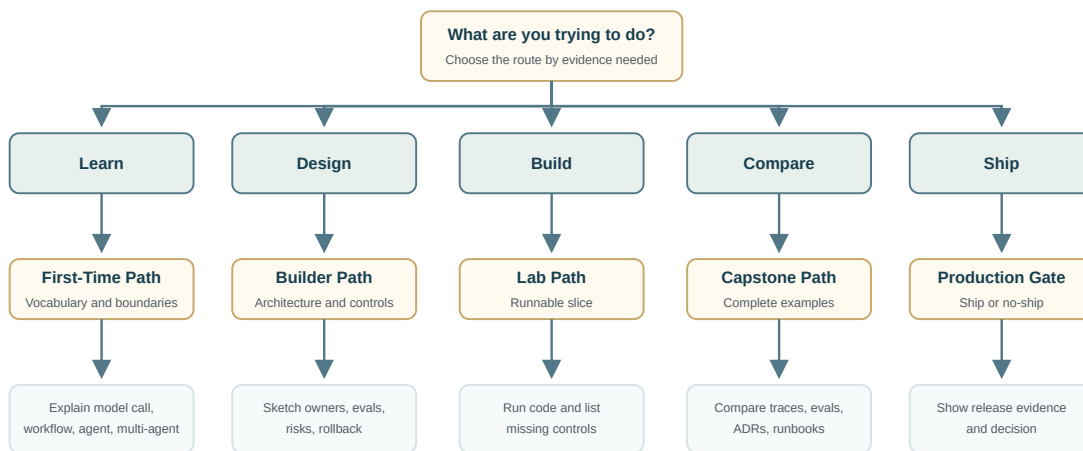
Elige tu ruta

Usa la ruta que coincida con la decisión que tienes enfrente.

Si eres...	Comienza con	Objetivo
Nuevo en agentic systems	First-Time Path	Aprende el vocabulario central y los límites.
Diseñando o revisando un sistema	Builder Path	Elige patterns, controles y responsabilidades de producción.
Ejecutando código	Lab Path	Construye pequeños ejemplos y conéctalos con la arquitectura.
Buscando ejemplos tipo producto	Capstone Path	Ve patterns combinados en sistemas completos.
Aprendiendo visualmente o integrando un equipo	Visual Architecture Route	Usa diagramas para entender límites, propiedad y controles de producción.
Consultando un pattern durante el trabajo	Reference Path	Revisa casos de uso, modos de falla, evals y listas de verificación de producción.
Eligiendo una sección para leer	Logical Groups	Entiende por qué el libro está agrupado así y dónde cada grupo aporta valor.
Atorado con la terminología	Glossary and Acronyms	Descifra agent, eval, protocol, security y términos de producción.
Preparando un release	10/10 Production Gate	Verifica si el sistema es revisable, testeable, observable y reversible.

La ruta más útil no es la lista más corta de páginas. Es la que te lleva a una decisión de diseño con suficiente context para evitar una mala abstracción.

Flujo de decisión de ruta de lectura



La ruta termina solo cuando produce la evidencia de la derecha.

Guía de tiempo y dificultad

Consulta esta tabla antes de elegir una ruta. Las estimaciones asumen un ingeniero de software que puede hojear material familiar y detenerse en patterns, código o controles de producción desconocidos.

Ruta	Mejor para	Dificultad	Tiempo	Evidencia que debes producir
First-Time Path	Aprender el vocabulario y los límites.	Principiante a intermedio	2-4 horas	Una breve explicación de agent, workflow, tool, state, memory, eval y condición de stop.
Builder Path	Diseñar o revisar un sistema real.	Intermedio a avanzado	1-2 días	Boceto de arquitectura, pattern seleccionado, alternativas rechazadas, owners, evals y ruta de rollback.
Lab Path	Ejecutar ejemplos y ver límites de implementación.	Intermedio	1-3 días	Pruebas aprobadas, salida de trace, lista de controles faltantes y una tarea de hardening por lab.
Capstone Path	Comparar sistemas completos tipo producto.	Avanzado	4-8 horas	Análisis de brechas contra traces, evals, ADRs, runbooks y controles de release.
Reference Path	Consultar un pattern durante el diseño.	Intermedio	10-30 minutos por capítulo	Decisión de fit/avoid, modo de falla, caso de eval y checklist de producción.
Release Path	Decidir si un sistema puede lanzarse.	Avanzado	2-6 horas	Scorecard completado, registro de evidencia de release, límites conocidos y decisión de ship/no-ship.

No optimices por velocidad. Optimiza por evidencia. Una lectura rápida que no cambia el diseño no es útil.

Criterios de salida de ruta

Usa estos chequeos para decidir si una ruta ya te sirvió. Si no puedes responder la pregunta de salida, continúa con la ruta enlazada antes de elegir un pattern o lanzar un sistema.

Ruta	Terminas cuando puedes...	Si no, continúa con...
First-Time Path	Explicar la diferencia entre una llamada a model, prompt chain, workflow, single agent y multi-agent system.	Pattern Selection and Composition
Builder Path	Dibujar el límite del sistema, nombrar el owner de state, tools, memory, policy, evals y rollback, y defender por qué cada pattern pertenece.	Reference Architecture
Lab Path	Ejecutar una pequeña implementación, identificar los controles de producción faltantes y nombrar el siguiente paso de hardening.	Lab Production Readiness Checklist
Capstone Path	Comparar un ejemplo completo con tu propio sistema y listar las brechas en traces, evals, ADRs, runbooks y controles de rollout.	Capstone Projects
Reference Path	Decidir si el pattern encaja, su costo, cómo falla y qué eval prueba que funciona.	Choosing the Right Pattern
Release Path	Mostrar evidencia de que el sistema es revisable, testeable, observable, reversible y tiene owner.	10/10 Production Gate

No consideres una ruta como completa solo porque leíste las páginas. Considérala completa cuando cambia la decisión de diseño que tienes enfrente.

Grupos lógicos

La barra lateral está organizada como una ruta de diseño, no como un catálogo plano.

Grupo	Propósito	Lee cuando
Start Here	Orientate, elige una ruta de lectura y define qué cuenta como agent.	Eres nuevo en el libro o necesitas la entrada más corta.
Pattern Selection and Composition	Decide si usar un prompt, chain, router, loop, multi-agent system o reliability pattern.	Estás eligiendo o revisando una arquitectura.
Agent Runtime Foundations	Aprende los primitivos centrales de runtime: single agent, loop, state, tools y structured output.	Necesitas el vocabulario y los límites de control.
Engineering Practice and Frameworks	Pasa de demo a sistema diseñado con lifecycle, harnesses, elecciones de framework y hojas de trabajo.	Estás planeando la implementación.

Grupo	Propósito	Lee cuando
Evaluation, Security, and Trust	Agrega eval gates, threat modeling, sandboxing y controles de confianza humana.	El sistema afecta usuarios, datos o acciones externas.
Control Loops	Agrega planning, reflection, evaluator-optimizer, self-improvement y patterns de self-healing.	El agent necesita control iterativo.
Context, Memory, and Knowledge	Diseña working sets, context packets, memory, semantic recall, RAG y límites de knowledge.	El sistema depende de evidencia o state recordado.
Tools, Skills, and Protocols	Diseña tool capabilities, skills, MCP, A2A, aprobaciones y comunicación segura.	El agent necesita capacidades externas.
Multi-Agent Systems	Elige y opera topologías para delegación, supervisión, debate y trabajo en paralelo.	Un agent ya no es suficiente.
Systems Architecture	Compón sistemas completos: servicios, sistemas RAG, coding agents, computer-use agents, dominios, ADRs y referencias.	Estás diseñando todo el límite del sistema.
Production Runtime	Ejecuta agents con durabilidad, observability, feedback loops, policy, budgets, eventos y controles de rollout.	Estás preparando para producción.
Hands-On Labs	Construye los patterns en ejercicios pequeños y tracks de framework.	Quieres práctica de implementación.
Capstone Projects	Estudia ejemplos completos tipo producto con traces, evals, ADRs, runbooks y rollback.	Quieres ver patterns combinados.

Recomendación para la Primera Lectura

Empieza aquí si quieres que el libro se lea como un libro, no como un catálogo:

- Lector: ingeniero o líder técnico que busca el argumento completo.
- Resultado: comprender la postura de arquitectura-primero del libro antes de usar el catálogo.
- Profundidad aproximada: media-larga; lo ideal es leerlo en varias sesiones.

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](#)

Esta ruta te da la tesis antes de los detalles del catálogo.

Ruta para Primeros Lectores

Empieza aquí si eres nuevo en agentic systems:

- Lector: ingeniero de software aprendiendo el vocabulario.
- Resultado: saber qué hace diferente a un agent de una llamada a un model, chain o workflow.
- Profundidad aproximada: corta-media; léelo antes de los 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](#)

Esta ruta te da el vocabulario central antes de los capítulos de production runtime.

Ruta para Builders

Usa esta ruta cuando estés diseñando o revisando un sistema:

- Lector: builder, staff engineer, arquitecto, revisor o líder técnico.
- Resultado: producir un diseño defendible con state, tools, memory, policy, evals y límites de runtime explícitos.
- Profundidad aproximada: media-larga; úsala junto con un diseño real.

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](#)

Esta ruta es ideal para trabajo de arquitectura, revisiones de diseño y verificaciones de preparación para producción.

Ruta de Labs

Usa esta ruta cuando quieras ejecutar código:

- Lector: ingeniero enfocado en la implementación.
- Resultado: ejecutar ejemplos pequeños, inspeccionar límites y entender qué debe cambiar antes de producción.
- Profundidad aproximada: práctica; espera tiempo de configuración y depuración.

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](#)

Cada lab enlaza de regreso a los capítulos de patterns y a los paquetes de código fuente descargables. Los labs se mueven intencionalmente entre Python, TypeScript, código neutral a frameworks, recuperación estilo LangChain/LangGraph, state graphs estilo LangGraph, supervisión y transcript evals estilo AutoGen, runtime packaging estilo Mastra, orquestación de flows estilo CrewAI, código A2A orientado a protocolos y evals basados en pruebas para que puedas ver la arquitectura debajo de la API.

Capstone Path

Usa esta ruta cuando quieras ejemplos completos con forma de producto:

- Lector: ingeniero que ha terminado los capítulos principales o los labs.
- Resultado: conectar patterns, traces, evals, ADRs, runbooks y rollout controls en sistemas completos.
- Profundidad aproximada: media; se recomienda como material para revisión de diseño.

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](#)

Esta ruta es ideal después de los labs. Muestra cómo los patterns, frameworks, evals, traces, ADRs y runbooks encajan entre sí.

Reference Path

Usa la barra lateral o la búsqueda cuando necesites un pattern específico. Cada capítulo generado de pattern sigue la misma estructura:

- Lector: ingeniero que toma una decisión local sobre un pattern.
- Resultado: decidir si un pattern es adecuado, cuál es su costo y cómo probarlo.
- Profundidad aproximada: consulta rápida; salta entre capítulos relacionados.

Las páginas de patterns incluyen:

- cuándo usarlo
- cuándo evitarlo
- arquitectura
- system shape
- core protocol
- notas de implementación
- modos de falla
- estrategia de evaluación
- checklist de producción
- código fuente y descargas

La estructura repetida facilita escanear los capítulos durante el trabajo de diseño. No leas esas páginas como una novela. Usa los capítulos redactados para el argumento y las páginas de patterns para tomar decisiones.

Logical Groups

Este libro está organizado según el orden en que normalmente aparecen las decisiones de ingeniería. Comienza con el problema y la elección del pattern, luego agrega los elementos primitivos del runtime, prácticas de ingeniería, controles de riesgo, arquitectura, operaciones en producción, laboratorios y ejemplos completos.

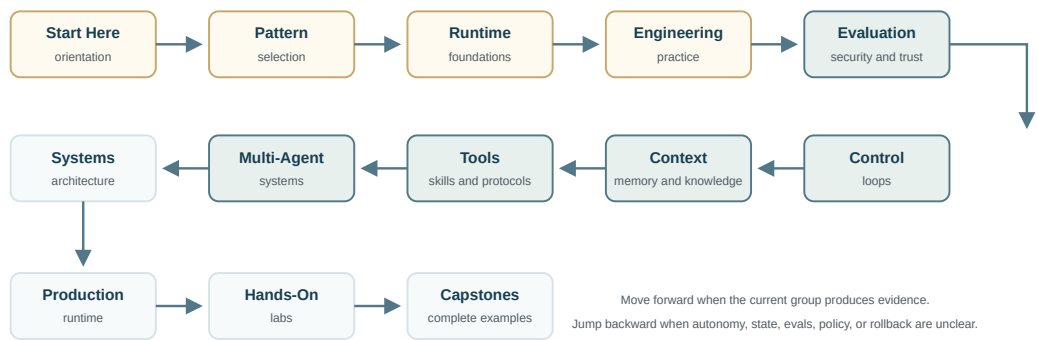
Usa esta página cuando la barra lateral parezca extensa. Cada grupo responde a una pregunta diferente y le da al lector un punto claro de partida.

Mapa de Grupos

Grupo	Pregunta Principal	Beneficio para el Lector
Start Here	¿Qué es este libro y qué cuenta como un agent?	Obtienes el vocabulario, rutas para el lector, glosario y barra de producción.
Pattern Selection and Composition	¿Qué pattern debe usar este sistema?	Evitas agregar autonomía donde un prompt, chain, router o workflow es suficiente.
Agent Runtime Foundations	¿Qué elementos primitivos hacen funcionar un agent?	Comprendes loops, state, tools, structured output, context y límites de control.
Engineering Practice and Frameworks	¿Cómo construimos esto como software y no solo como un demo?	Obtienes orientación sobre lifecycle, harness, framework, setup y worksheet.
Evaluation, Security, and Trust	¿Cómo sabemos que esto es seguro y útil?	Conectas evals, threat models, sandboxing y user trust con decisiones de lanzamiento.
Control Loops	¿Cuándo debe el sistema planear, reflexionar, optimizar o recuperarse?	Eliges control iterativo sin ocultar fallas o costos.
Context, Memory, and Knowledge	¿Qué debe saber, recordar o recuperar el agent?	Diseñas context packets, memory, RAG y límites de knowledge de manera deliberada.
Tools, Skills, and Protocols	¿Qué puede hacer el agent fuera del model?	Defines tool contracts, skills, approvals, MCP, A2A y seguridad en la comunicación.
Multi-Agent Systems	¿Cuándo un solo agent ya no es suficiente?	Comparas delegación, supervisión, debate, paralelismo y workflows tipo crew.
Systems Architecture	¿Cómo se convierten estos patterns en un sistema completo?	Ves servicios, sistemas RAG, coding agents, personal agents, arquitecturas de dominio, ADRs y referencias.
Production Runtime	¿Cómo sobrevive el sistema en operación real?	Agregas durabilidad, observability, feedback loops, budgets, policy, eventos, rollout y rollback.

Grupo	Pregunta Principal	Beneficio para el Lector
Hands-On Labs	¿Cómo se ven estas ideas en código?	Construyes pequeños vertical slices en Python, TypeScript, runtimes personalizados y frameworks comunes de agent.
Capstone Projects	¿Cómo se ve un agentic system completo?	Estudias sistemas con forma de producto con traces, evals, ADRs, runbooks y planes de rollback.
Historical Patterns	¿Qué términos antiguos siguen siendo relevantes?	Puedes comparar el lenguaje de arquitectura actual con nombres de patterns antiguos.
Publishing Appendix	¿Cómo se mantiene y publica el libro?	Obtienes notas de lanzamiento, checklist y publicación para el libro en línea.

Design Pipeline



Lee de izquierda a derecha cuando quieras que el libro construya un solo argumento de diseño. Salta al grupo con la evidencia faltante cuando estés revisando un sistema real.

Mapa de Problemas

Usa este mapa cuando conoces el problema pero no el grupo correcto. Comienza con la sección indicada y luego sigue sus capítulos relacionados.

Si tu problema es...	Comienza con	Terminas con
El equipo quiere agregar un agent pero no ha justificado la autonomía.	Pattern Selection and Composition	Un diseño más pequeño o una razón clara para el comportamiento agentic.
La implementación oculta el state en prompts o historial de chat.	Agent Runtime Foundations	Goals explícitos, state, tools, structured output y condiciones de parada.
Un demo funciona, pero nadie sabe cómo probarlo u operarlo.	Engineering Practice and Frameworks	Un lifecycle, harness, límite de framework y rastro de worksheet.

Si tu problema es...	Comienza con	Terminas con
El sistema puede afectar usuarios, datos, dinero o sistemas externos.	Evaluation, Security, and Trust	Evals, threat model, sandbox, approval y controles de trust.
El agent sigue en loop, reintentando o autocorrigiéndose sin evidencia clara.	Control Loops	Un diseño de loop con condiciones de parada, budgets y estados de falla revisables.
Las respuestas dependen de documentos, memory o ensamblaje de context.	Context, Memory, and Knowledge	Policy de origen, context packet, retrieval, memory y reglas de frescura.
El agent necesita tools, approvals, MCP, A2A o mensajes seguros.	Tools, Skills, and Protocols	Tool contracts restringidos, sobres de protocolo, permisos y registros de auditoría.
Un agent está sobrecargado o necesita colaboradores especialistas.	Multi-Agent Systems	Una topología, límite de roles, transcript, merge policy y regla de parada.
Varios patterns deben convertirse en un solo producto desplegable.	Systems Architecture	Un límite de sistema, ADRs, forma de servicio y arquitectura de referencia.
El diseño está cerca de producción.	Production Runtime	Durabilidad, observability, policy, budgets, rollout y evidencia de rollback.
Quieres prueba en código antes de comprometerte con un diseño.	Hands-On Labs	Un pequeño vertical slice ejecutable y una lista de controles de producción faltantes.
Quieres comparar con ejemplos completos.	Capstone Projects	Una lista de brechas para traces, evals, ADRs, runbooks y controles de lanzamiento.

Orden Recomendado

Lee estos grupos en orden cuando quieras que el libro construya un solo argumento:

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

Este orden mantiene al lector dentro de un solo argumento de diseño: elige el pattern, define el runtime, agrega control de loop solo donde sea necesario, da al agent evidencia y capabilities, decide si un agent es suficiente y luego convierte el diseño en un sistema operado.

Ruta de Arquitectura Visual

Usa esta ruta cuando quieras que los diagramas lleven la primera pasada. Es útil para revisiones de diseño, onboarding de equipos y lectura en PDF.

Paso	Capítulo	Qué debe aclarar el diagrama
1	Agent Loop	Cómo el state, las acciones, las observaciones y las condiciones de detención forman una run.
2	Planning and Execution	Quién es responsable del plan, la validación, la ejecución y el progreso.
3	Memory-Augmented Agent	Cómo se gobiernan la recuperación, inyección de context, almacenamiento y corrección.
4	Tool Capability Design	Dónde se ubican los permisos, policy, llamadas a tools y registros de auditoría.
5	Secure Agent Communication	Qué puertas de autoridad protegen el intercambio agent-to-agent o con herramientas remotas.
6	Choosing Multi-Agent Topology	Cuándo delegar, supervisar, debatir, paralelizar o permanecer como single-agent.
7	Agentic System Architecture	Cómo los patterns se componen en servicios, state, policy, evals y operaciones.
8	Production Runtime Overview	Qué debe existir alrededor del agent antes de su uso en producción.
9	Event-Triggered Agents	Cómo los triggers desatendidos se mantienen idempotentes, observables, reproducibles y seguros.

Después de esta ruta, usa los capítulos de patterns como referencia. Cada diagrama debe responder una pregunta concreta de ownership, no solo decorar la página.

Contratos de Grupo

Cada grupo debe entregar al lector un artifact o decisión durable. Usa esta tabla para saber cuándo continuar, pausar o saltar.

Grupo	Entra cuando	Sale con
Start Here	Necesitas orientación, vocabulario o el estándar de producción.	Una ruta de lectura y una definición compartida de agentic system.
Pattern Selection and Composition	Estás decidiendo si la autonomía pertenece al diseño.	Un pattern seleccionado, alternativas rechazadas y criterios de aceptación.
Agent Runtime Foundations	Necesitas hacer explícita la run.	Goal, state, tool, output, context y contratos de condiciones de detención.
Engineering Practice and Frameworks	Estás convirtiendo un diseño en software.	Harness, elección de framework, worksheet y registro de revisión.
Evaluation, Security, and Trust	El sistema afecta usuarios reales, datos, dinero o decisiones.	Eval set, threat model, sandbox boundary y controles de confianza.
Control Loops	El sistema debe planear, revisar, evaluar o recuperarse.	Loop policy, presupuesto, evaluator, estados de falla y reglas de detención.
Context, Memory, and Knowledge	La respuesta depende de evidencia, historial o recuperación.	Context packet, source policy, regla de memory y límite de frescura.
Tools, Skills, and Protocols	El agent necesita capacidades externas o intercambio agent-to-agent.	Contratos de tool, sobre de permisos, puerta de aprobación y registro de auditoría.
Multi-Agent Systems	Un agent está sobrecargado o múltiples roles mejoran el trabajo.	Topología, contratos de rol, merge policy, transcript y ruta de escalamiento.
Systems Architecture	Los patterns deben convertirse en un producto desplegable.	Límite de servicio, ADRs, flujo de datos/control y arquitectura de referencia.
Production Runtime	El sistema está cerca de su lanzamiento o ya está operando.	Durabilidad, traces, feedback de eval, presupuestos, rollout y plan de rollback.
Hands-On Labs	Necesitas prueba en código.	Slice ejecutable, evidencia de pruebas y lista de controles de producción faltantes.
Capstone Projects	Quieres un punto de comparación end-to-end.	Análisis de brechas contra un sistema con forma de producto.

Reglas de Handoff

Avanza solo cuando el grupo actual haya producido evidencia. Una elección de pattern sin criterios de aceptación no está lista para implementación. Un diseño de runtime sin state y condiciones de detención no está listo para tools. Una superficie de tool sin límites de aprobación y auditoría no está lista para producción. Un laboratorio sin lista de controles faltantes no es prueba de preparación.

Retrocede cuando falte evidencia:

Evidencia faltante	Regresa a
El equipo no puede explicar por qué el sistema necesita autonomía.	Pattern Selection and Composition
State, tools, memory o policy viven solo dentro de prompts.	Agent Runtime Foundations
No hay eval set o threat model.	Evaluation, Security, and Trust
El sistema no tiene trace, replay, presupuesto o ruta de rollback.	Production Runtime
La arquitectura no tiene registro escrito de decisiones.	Systems Architecture

Estos handoffs evitan que el libro se convierta en un recorrido de conceptos. Cada sección debe producir una decisión, exponer una brecha o enviar al lector a la sección que puede cerrarla.

Prueba de Valor

Cada grupo debe ayudar al lector a tomar una mejor decisión de ingeniería:

- elegir un diseño más pequeño cuando la autonomía no es necesaria
- hacer visible el state, tools, memory y context
- agregar evals antes de confiar en el comportamiento
- separar el juicio del model del control determinista
- hacer que el riesgo sea revisable por otro ingeniero
- conectar ejemplos con responsabilidades de producción

Si un grupo no ayuda con uno de esos resultados, debe ser revisado, fusionado o eliminado.

Conclusión

El libro no está organizado por nombres de tendencias. Está organizado por decisiones: elegir el pattern, definir el runtime, controlar el riesgo, componer el sistema, operarlo y probarlo con ejemplos.

What Is An Agent?

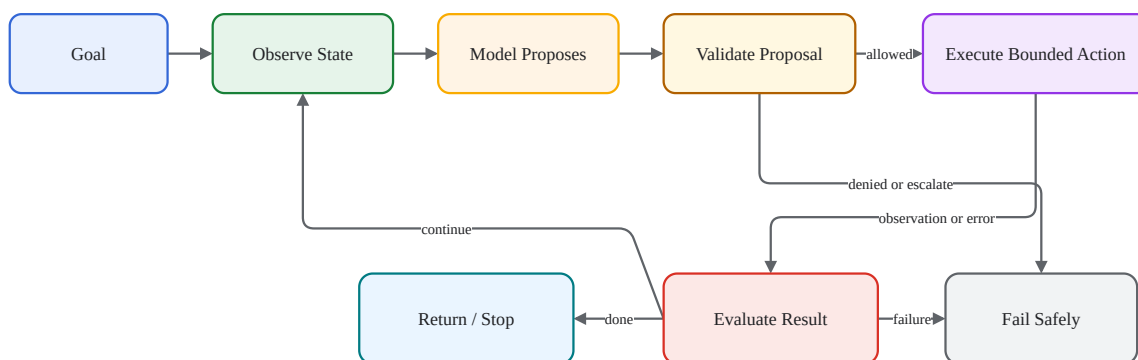
Un agent no es magia. Es un loop con acceso a un model, algo de state, algunas decisiones y, a veces, tools. El loop es lo que importa; todo lo demás es envoltura.

Un framework puede llamarlo graph, crew, swarm, harness, assistant o runtime. Si quitas los nombres, casi siempre encuentras la misma forma debajo: observar, decidir, actuar, volver a observar y, finalmente, detenerse.

Este capítulo define mecánicamente qué es un agent. No define un framework, interfaz de producto, sistema de memory o production runtime. Esas capas rodean el loop, pero no lo reemplazan.

Lee esto antes del catálogo de patterns si quieres primero el modelo básico: un agent es un loop controlado, no una personalidad, una ventana de chat ni una categoría de producto. Después de esto, [Architecture Before Autonomy](#) explica cuándo vale la pena usar ese loop.

Agent Loop



The Minimal Agent

El agent más pequeño y útil necesita un goal o input, instrucciones, context, acceso a un model, un paso de decisión, state, una regla para lo que sucede después y una condición de parada. Las tools son opcionales, pero normalmente son lo que hace que valga la pena construir el agent.

En términos simples, eso es:

```

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

La mayor parte de agent engineering consiste en hacer que cada una de esas flechas sea explícita y segura. El diagrama es sencillo. La disciplina está en decidir qué puede hacer el model en cada paso y qué código se reserva para sí mismo.

El loop útil más pequeño se puede esquematizar así:

```
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' };
}
```

La parte importante no es la sintaxis. El model propone una decisión, el software la valida antes de ejecutarla, el state registra la observación y el loop se detiene por una razón explícita. La versión completa y ejecutable aparece en [Agent Loop](#).

A Model Call Is Not Yet An Agent

Un model call toma un input y devuelve texto o structured output. Para tareas de resumen, extracción, clasificación o reescritura, eso suele ser suficiente.

Un agent agrega control de flujo alrededor de la llamada. Puede decidir si se necesita más trabajo, si usar una tool, si preguntar a un humano, si reintentar o si detenerse. La diferencia no es inteligencia. Es la forma del runtime.

System	What It Can Do	Main Risk
Model call	Produce una respuesta a partir del context proporcionado.	Sin acción ni recuperación.
Prompt chain	Avanza por pasos conocidos.	Puertas frágiles y latencia innecesaria.
Workflow with LLM steps	El código controla el camino mientras los models manejan el juicio.	Sobreajuste a un proceso fijo.
Agent loop	Elige el siguiente paso según las observaciones.	Loops, mal uso de tools y drift oculto del state.

Así que ten cuidado con la palabra. No toda función respaldada por un model es un agent. Reserva el término para sistemas que pueden tomar al menos algunas decisiones en runtime sobre qué hacer después.

A Quick Classification Test

Cuando un equipo dice “agent”, clasifica el sistema antes de debatir el framework.

Example	Better Name	Why
Un botón que reescribe un párrafo una vez.	Model call	No hay loop, uso de tools ni decisión en runtime.
Un generador de respuestas de soporte en tres pasos: clasificar, recuperar policy, redactar respuesta.	Prompt chain o workflow	El camino se conoce antes de la ejecución.

Example	Better Name	Why
Un flujo de reembolso donde el código revisa la policy, el model resume la evidencia y la aprobación libera el pago.	Workflow with LLM steps	El código controla el camino y los efectos secundarios.
Un asistente de investigación que busca, lee resultados, decide si necesita más retrieval y se detiene cuando la evidencia es suficiente.	Agent loop	El siguiente paso depende de las observaciones del run.
Un sistema de entregas con planner, dispatcher, worker de comunicación con el conductor y escalation policy.	Multi-agent system o workflow with agents	El trabajo se divide en roles con context y autoridad separados.

La clasificación no es cuestión de prestigio. Un model call no es peor que un agent. Un workflow no es menos moderno que un loop. El nombre correcto te dice qué debe probarse, trazarse, asegurarse y operarse.

Running Example: Refund Support

El asistente de reembolsos de soporte usado en este libro comienza como un workflow, no un agent. El camino conocido es claro: clasificar la solicitud, cargar el pedido, recuperar la refund policy actual, resumir la evidencia, validar la recomendación y solicitar aprobación cuando se puede mover dinero. El código debe controlar ese camino.

La parte agentic aparece solo cuando el workflow enfrenta incertidumbre acotada. Tal vez los datos del pedido entran en conflicto con el registro de entrega. Tal vez el cliente reclama una excepción de policy. Tal vez falta evidencia y el sistema debe decidir si buscar otra fuente, hacer una pregunta aclaratoria o detenerse. Esa investigación limitada puede justificar un loop.

Part of the refund system	Better shape	Owner
Intake, búsqueda de cuenta y selección de versión de policy	Deterministic workflow	Application code
Resumen de evidencia y borrador de recomendación	Model call o paso de structured-output	El model propone, el código valida
Investigación de evidencia faltante o conflictiva	Agent loop con máximo de pasos	Loop runtime
Umbral de reembolso, exception policy y requisito de aprobación	Policy gate	Software
Ejecución de pago	Tool call después de la aprobación	Tool gateway

Esta es la disciplina detrás de la palabra “agent”. No preguntes si el asistente de reembolsos suena autónomo. Pregunta qué decisión en runtime necesita autonomía, qué efectos secundarios requieren autoridad y qué razón de parada registrará el sistema.

Decision Trees Come Before Autonomy

Muchos agents útiles son en su mayoría decision trees con model calls dentro. Eso no es una debilidad y, a menudo, es el diseño correcto:

```
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
```

Esto sigue siendo un agentic system mientras el model tome decisiones acotadas dentro del flujo. Lo que lo mantiene seguro es que el código controla la policy, el state y las transiciones de alto riesgo. Agrega autonomía donde el decision tree se vuelve demasiado grande para mantener, o donde el siguiente paso realmente depende de algo que el run descubre en el camino. Agregarla en otro lugar solo te trae riesgos innecesarios.

Tools Make The Loop Consequential

Sin tools, un agent solo puede producir texto. Con tools, puede buscar, calcular, recuperar, escribir archivos, navegar, llamar APIs, ejecutar código, enviar mensajes, actualizar tickets o cambiar datos de clientes. Ahí es donde el sistema se vuelve útil, y también donde se vuelve peligroso.

Nunca trates una tool call como “el model lo hizo”. El model propuso la llamada; el software la ejecutó. Ese límite es donde vive la mayor parte de tu seguridad, así que debe ser real: los nombres de tools se conocen de antemano, los argumentos tienen tipo, los permisos se revisan antes de ejecutar, existen timeouts, los resultados se registran, los efectos secundarios son auditables y las acciones riesgosas esperan aprobación. Cuanto más poderosa la tool, menos confianza debes dejar en el prompt.

El State Hace Coherente el Loop

Un agent necesita state porque el loop requiere una memory de su propia ejecución. Como mínimo, el state debe poder responder cuál es el goal activo, qué context se usó, qué ha sucedido, qué tools se invocaron, qué evidence se encontró, qué errores ocurrieron, cuánto budget queda y por qué se detuvo la ejecución.

El historial de conversación no cubre esto. El historial es evidencia de lo que se dijo; no es un modelo de lo que el sistema sabe y ha hecho. El state real es lo que permite que el sistema se reanude después de una interrupción, evite repetir trabajo, explique una decisión después del hecho y convierta un incidente de producción en un caso de eval.

Las Condiciones de Parada Son Parte del Agent

Un loop sin una condición de parada no es un agent. Es un generador de costos. Las condiciones de parada comunes incluyen que se cumplan los criterios de éxito, que falte un campo requerido, una falla de tool no recuperable, un bloqueo de policy, una aprobación humana pendiente, un límite máximo de iteraciones, un budget de costo o latencia agotado, o que un usuario cancele la ejecución.

Sea cual sea la razón por la que termina la ejecución, el motivo de parada debe ser explícito. “Listo” no es suficiente. Un sistema que vale la pena operar registra si el agent completó, rechazó, escaló, falló, expiró por tiempo o se detuvo porque se agotó el budget, ya que cada uno de esos casos requiere una respuesta diferente de las personas que lo supervisan.

Qué Aportan los Frameworks

Los frameworks no eliminan el loop. Lo empaquetan y ahorran trabajo real: ejecución de grafos, enrutamiento, registros de tools, adaptadores de model, almacenes de memory, skills, subagents, checkpoints durables, streaming, interrupciones human-in-the-loop, tracing y hooks de despliegue. Nada de eso es esfuerzo desperdiciado, y reconstruir estos elementos básicos a mano rara vez es un buen uso del tiempo del equipo.

Pero el framework no cambia la pregunta que debes responder: ¿qué observa el loop, qué puede decidir, qué puede hacer y cuándo se detiene? Si tu equipo no puede responder eso sin antes nombrar un framework, aún no entiende su propio agent.

Qué Hace Bueno a un Agent

Un buen agent no es el que tiene más autonomía. Es aquel cuya autonomía es útil y está delimitada. En la práctica, eso significa goals claros, tools limitadas, state explícito, outputs tipados, decisiones visibles, loops acotados, evals que realmente detectan regresiones, modos de falla seguros y escalamiento humano para los casos riesgosos. Los malos agents esconden todo esto dentro de un prompt y esperan que el model se comporte.

La regla de diseño se desprende de todo lo anterior: si no puedes dibujar el loop, los tools, el state y la condición de parada, aún no entiendes el agent y no estás listo para darle autonomía.

Antes de llamar agent a algo, escribe cuatro cosas:

1. La decisión en runtime que se le permite tomar.
2. El state que lee y escribe fuera del context del model.
3. Los tools o efectos secundarios que puede solicitar.
4. Las razones exactas de parada que el sistema registra.

Si falta alguna de esas, probablemente tienes una feature, chain o workflow respaldado por un model que necesita límites más claros antes de necesitar más autonomía.

La siguiente pregunta no es cómo agregar más autonomía. Es si la task necesita un loop en absoluto. Continúa con [Architecture Before Autonomy](#), luego regresa a [Agent Loop](#) cuando las decisiones en runtime estén justificadas.

Capítulos Relacionados

- Architecture Before Autonomy
- Single Agent
- Agent Loop
- Goals and State
- Tool Use
- Choosing the Right Pattern

Glosario y Acrónimos

Usa este capítulo cuando un término bloquee el avance. Las definiciones son breves a propósito. Cada término explica qué significa la palabra en este libro y por qué importa al diseñar, probar u operar un agentic system.

Lee el glosario como un vocabulario de control. En este libro, los términos solo son útiles cuando ayudan al equipo a decidir sobre ownership, riesgo, evidencia o preparación para release.

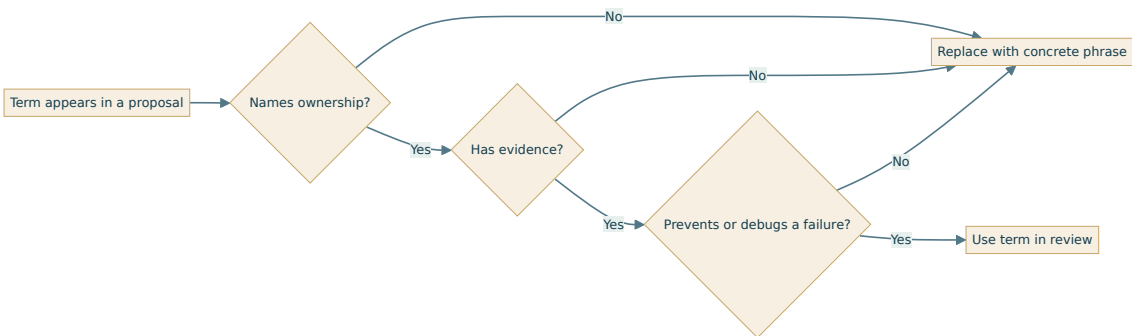
Cómo Usar Este Glosario

Usa un término en una revisión de diseño solo si el equipo puede responder tres preguntas:

- 1. ¿Qué posee este término?
- 2. ¿Qué evidencia prueba que funciona?
- 3. ¿Qué falla nos ayuda a prevenir o depurar?

Si un término no responde esas preguntas, reemplázalo por una frase más concreta. Por ejemplo, “agentic workflow” es demasiado amplio a menos que puedas nombrar el goal, state, límite de tool, puerta de policy y condición de stop.

Usa este flujo cuando un término suene útil pero vago. Conserva el término solo cuando clarifique ownership, evidencia y análisis de fallas.



Mapa de Revisión de Términos

Usa este mapa durante revisiones de diseño. Cuando un término aparezca en una propuesta, haz la pregunta de revisión antes de aceptarlo.

Cuando alguien dice...	Pregunta...	Ir a
Agent	¿Qué decisión en runtime toma el model que el código no puede tomar con suficiente seguridad?	¿Qué es un Agent?
Autonomous	¿Qué decisiones, tools, presupuestos y condiciones de stop se delegan?	Arquitectura antes de Autonomía

Cuando alguien dice...	Pregunta...	Ir a
Memory	¿Es context local de task, memory durable de usuario, historial episódico o knowledge recuperado?	Context Engineering
Tool	¿Qué autoridad otorga y qué revisiones de policy se ejecutan antes de efectos secundarios?	Tool Capability Design
Eval	¿Qué comportamiento, trayectoria, decisión de policy o modo de falla detecta?	Evaluation-Driven Agent Development
Human-in-the-loop	¿La persona aprueba una acción exacta, revisa un resultado o solo recibe una notificación?	Human Approval Gates
Multi-agent	¿Qué context, permiso, skill, latencia o necesidad de revisión justifica otro agent?	Choosing Multi-Agent Topology
Production-ready	¿Dónde está la evidencia de traces, puertas de eval, rollback, ownership y respuesta a incidentes?	10/10 Production Gate

Términos Clave

Término	Significado	Por qué importa
Agent	Un sistema que persigue un goal mediante un loop, usa tools o context, rastrea state y se detiene por regla.	Separa la arquitectura real de agent de una simple llamada a un model.
Agent loop	El ciclo de observar, decidir, actuar, evaluar y detenerse alrededor de un agent.	Aquí deben vivir los presupuestos, policy, state y manejo de fallas.
Autonomy	La cantidad de autoridad de decisión delegada al sistema.	Más autonomy requiere límites, pruebas, aprobaciones y rollback más fuertes.
Boundary	La línea entre la elección del model y el control determinista del sistema.	Límites débiles convierten la incertidumbre del model en riesgo para el sistema.
Capability	Algo que el sistema puede hacer, usualmente mediante un tool, skill, workflow o servicio.	Las capabilities necesitan ownership, autorización, auditoría y pruebas.
Context	La información pasada a un model o paso de agent.	El context controla lo que el model puede usar, confundir, filtrar o ignorar.
Goal	El resultado explícito que el sistema intenta alcanzar.	Goals ambiguos producen comportamientos ambiguos y evals débiles.
Policy	Una regla que restringe lo que el agent puede hacer.	La policy debe aplicarse fuera del prompt cuando las acciones implican riesgo.

Término	Significado	Por qué importa
State	Datos que registran la ejecución actual, memory, progreso, decisiones o efectos externos.	State oculto dificulta el replay, la depuración y las revisiones de seguridad.
Stop condition	Una regla que termina un loop o transfiere el control a un humano/sistema.	Sin stop conditions, los agents desperdician presupuesto o repiten acciones inseguras.
Tool	Una capability invocable expuesta a un agent.	El diseño de tools es un plano de control, no solo un wrapper de API.

Confusiones Comunes

Estos pares causan muchos diseños débiles. Usa el término más preciso en las revisiones.

Términos confundidos	Distinción
Model call vs agent	Un model call retorna una respuesta. Un agent loop puede decidir qué hacer después.
Workflow vs agent	Un workflow sigue un camino definido por código. Un agent elige al menos algunos pasos a partir de observaciones.
Memory vs context	Context es lo que el model ve ahora. Memory es state almacenado con reglas de retención, corrección y eliminación.
Tool vs skill	Un tool ejecuta una capability. Un skill empaqueta procedimientos, referencias, scripts, plantillas y ejemplos.
Policy vs prompt instruction	La policy la aplica el software o un límite de autoridad. Una prompt instruction es una guía para el model.
Eval vs test	Los tests suelen revisar comportamiento determinista. Los evals revisan outputs, trayectorias o juicios influenciados por el model.
Trace vs log	Los logs registran eventos. Los traces conectan la ejecución: goal, context, decisiones, tools, outputs, costos y razón de stop.
Approval vs notification	Approval bloquea una acción exacta hasta ser aceptada. Notification solo informa lo que ocurrió.

En caso de duda, elige el término que haga más visible el ownership y la evidencia.

Términos de Arquitectura

Término	Significado	Por qué importa
Architecture decision record	Un registro breve de una decisión de arquitectura, context, alternativas y	Los ADRs hacen que el diseño de agent sea revisable después de la fase

Término	Significado	Por qué importa
(ADR)	consecuencias.	de demo.
Circuit breaker	Un control que detiene o degrada la ejecución cuando las fallas superan un umbral.	Previene que fallas repetidas de model/tool se conviertan en daños visibles para el usuario.
Deterministic boundary	Código, schema, policy o lógica de workflow que no depende del juicio del model.	Mantiene las decisiones de alto riesgo fuera de la generación probabilística de texto.
Durable workflow	Un workflow que puede persistir, reanudarse, reintentarse y recuperarse tras una interrupción.	Los agents en producción necesitan continuidad ante caídas, colas y timeouts.
Fallback	Un camino alternativo más seguro usado cuando el camino preferido falla o se vuelve muy riesgoso.	Los fallbacks preservan la calidad del servicio sin fingir que la autonomy siempre funciona.
Handoff	Una transferencia tipada de trabajo entre componentes, agents, tools o humanos.	Los handoffs necesitan context, state, ownership y criterios de aceptación.
Idempotency	La propiedad de que repetir una acción produce el mismo efecto deseado.	Los reintentos y replay son inseguros sin acciones externas idempotentes.
Replay	Volver a ejecutar un trace, paso de workflow o camino de decisión para depuración o evaluación.	El replay convierte incidentes en pruebas reproducibles.
Routing	Elegir qué camino, tool, model, workflow o agent debe manejar una task.	Un mal routing agrega costo, latencia y modos de falla.
Service boundary	El contrato alrededor de un componente desplegable del sistema.	Los agents se vuelven operables cuando se tratan como servicios con APIs y SLOs.

Términos de Evaluación

Término	Significado	Por qué importa
Eval case	Un input concreto, comportamiento esperado y regla de puntuación.	Convierte una afirmación de calidad vaga en una revisión repetible.
Eval gate	Un umbral de evaluación requerido antes de release o despliegue.	Previene que cambios en prompt o policy se publiquen solo por confianza.
Fixture	Un input de prueba controlado, conjunto de documentos, respuesta de tool o trace.	Los fixtures hacen reproducible el comportamiento de un agent.

Término	Significado	Por qué importa
Grader	Código, rúbrica o model que puntúa un output o trayectoria.	Los graders deben probarse, o se vuelven otro agent poco confiable.
Regression	Un comportamiento previamente correcto que falla tras un cambio.	Los agent systems necesitan suites de regression porque pequeños cambios en el prompt pueden alterar el comportamiento.
Trace	Un registro estructurado de prompts, llamadas a tools, decisiones, observaciones, costos y resultados.	Los traces explican lo que ocurrió cuando los outputs no son suficientes.
Trajectory	El camino completo que un agent sigue a través de pasos, tools, state y decisiones.	Algunas fallas solo aparecen a lo largo del camino, no en la respuesta final.

Términos de Memory y Knowledge

Término	Significado	Por qué importa
Context budget	El límite de tokens, latencia, costo y atención para el context.	Más context no siempre es mejor; puede distraer o filtrar información.
Context packet	Un paquete ensamblado deliberadamente de instrucciones, hechos, state, tools y evidencia.	Permite que la selección de context sea inspeccionable.
Episodic memory	Registros almacenados de eventos, interacciones o ejecuciones.	Ayuda a la personalización y continuidad, pero requiere reglas de retención y corrección.
Knowledge boundary	Las fuentes confiables que un agent puede usar para afirmaciones.	Reduce la alucinación y permite rechazar cuando falta evidencia.
Retrieval-augmented generation (RAG)	Un diseño donde la evidencia recuperada se agrega al context del model.	RAG solo ayuda cuando la calidad de retrieval, las citas y la frescura se prueban.
Semantic recall	Recuperación basada en significado en vez de coincidencia exacta de palabras clave.	Puede encontrar evidencia útil, pero también puede recuperar material erróneo pero plausible.
Working memory	State local de la task usado durante una ejecución o sesión.	Mantiene el trabajo de varios pasos coherente sin convertir cada hecho en memory durable.

Términos de Seguridad y Confianza

Término	Significado	Por qué importa
Approval gate	Un punto donde un humano o un servicio de policy debe aprobar una acción exacta.	Los gates reducen el riesgo para dinero, credenciales, datos de usuario y cambios irreversibles.
Audit log	Un registro de decisiones, aprobaciones, llamadas a tools y efectos externos.	Los audit logs apoyan la revisión de incidentes y la rendición de cuentas.
Credential boundary	La regla sobre qué identidad o secreto puede usar una llamada a tool.	Los agents no deben heredar credenciales amplias por defecto.
Least privilege	Dar al sistema solo el acceso necesario para la task.	Un acceso amplio convierte errores del model en incidentes de seguridad.
Sandbox	Un entorno restringido para acciones de código, navegador, archivos o red.	Los sandboxes limitan el daño cuando las acciones generadas fallan o son atacadas.
Threat model	Un análisis estructurado de lo que puede salir mal, quién puede causarlo y cómo reducir el riesgo.	Los threat models de agents deben incluir tools, memory, prompts, usuarios y datos externos.
Trust contract	Las promesas que el producto hace sobre lo que el agent puede hacer, mostrar, cambiar o decidir.	Los usuarios necesitan control claro cuando los agents actúan en su nombre.

Términos de Protocolos y Frameworks

Término	Significado	Por qué importa
A2A	Comunicación agent-to-agent mediante mensajes tipados, identidad y autorización.	A2A necesita contratos; el chat libre entre agents es difícil de asegurar o depurar.
MCP	Model Context Protocol, un protocolo para exponer tools y recursos a clientes de model.	MCP puede estandarizar el acceso a tools, pero los servidores aún requieren auth, policy y control de ciclo de vida.
Skill	Una capability procedimental empaquetada con instrucciones, archivos, scripts o ejemplos.	Las skills preservan el know-how probado y reducen el exceso de prompts repetidos.
Supervisor	Un componente o agent que asigna, revisa o coordina agents trabajadores.	Los supervisors necesitan autoridad, reglas de detención y manejo de fallos.
Worker	Un agent o componente acotado responsable de una subtask.	Los workers deben tener entradas, salidas, tools y criterios de éxito limitados.

Acrónimos

Acrónimo	Expansión	Nota
ADR	Architecture Decision Record	Usar para decisiones de diseño y consecuencias.
A2A	Agent-to-Agent	Usar para comunicación tipada entre agents.
API	Application Programming Interface	Tratar agent tools como APIs con policy.
CI	Continuous Integration	Usar CI para eval gates y revisiones de regresión.
KPI	Key Performance Indicator	Usar con cuidado; los KPIs de agents requieren métricas de calidad y riesgo.
MCP	Model Context Protocol	Usar para interoperabilidad de tools/recursos.
PII	Personally Identifiable Information	Requiere minimización, enmascaramiento, retención y reglas de acceso.
RAG	Retrieval-Augmented Generation	Requiere evals de retrieval y controles de fuente.
SLA	Service Level Agreement	Compromiso externo sobre el comportamiento del servicio.
SLO	Service Level Objective	Objetivo interno para confiabilidad, latencia o calidad.
UX	User Experience	Incluye confianza, visibilidad, corrección y control.

Regla de Revisión

Cuando un capítulo, ADR o nota de lanzamiento use un acrónimo, defínelo en su primer uso a menos que la sección circundante ya lo haga. Esto importa para un libro en línea porque los lectores a menudo llegan a una sola página desde una búsqueda en vez de leer los capítulos en orden.

Capítulos Relacionados

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

10/10 Production Gate

Este libro no está terminado cuando puedes describir agent patterns. Está terminado cuando puedes diseñar un sistema que un equipo pueda revisar, probar, lanzar, observar y hacer rollback.

Usa este gate antes de declarar un agentic system listo para producción. Un puntaje menor a 8 significa que el diseño sigue siendo un demo o prototipo. Un puntaje de 10 significa que el equipo tiene evidencia, no solo confianza.

Descarga la hoja de trabajo reutilizable: [10/10 production gate scorecard](#).

Cuándo Usar Este Gate

Usa este gate en tres momentos:

Momento	Propósito	Resultado Esperado
Antes de la implementación	Encontrar falta de ownership antes de que el código haga costoso cambiar el diseño.	Cambios de arquitectura, no aprobación de lanzamiento.
Antes del piloto	Decidir si usuarios limitados, datos o efectos secundarios son aceptables.	Piloto interno, piloto controlado o bloqueado.
Antes de producción	Verificar evidencia para cada camino de alto riesgo.	Lanzar, lanzar con límites documentados o no lanzar.

No esperes hasta la semana de lanzamiento. El gate es más útil cuando cambia el diseño temprano.

Qué Prueba Un Sistema 10/10

Área	Un Sistema 10/10 Demuestra	Evidencia
Goal	El sistema tiene un goal estrecho, non-goals explícitos y criterios de éxito claros.	Problem statement, user story, acceptance tests.
Boundary	El diseño separa el juicio del model del control determinista.	Architecture diagram, service contracts, policy boundary.
State	El sistema nombra quién es dueño del run state, durable memory, user data y external effects.	State schema, retention rules, replay plan.
Tools	Cada tool tiene un contrato tipado, regla de autorización, timeout y registro de auditoría.	Tool manifest, test fixtures, audit log example.
Context	El context se arma deliberadamente y lleva información de origen, confianza, frescura y presupuesto.	Context packet example, retrieval trace, context budget.
Evaluation	El equipo puede correr evals para happy paths, edge cases, regresiones y fallas conocidas.	Eval dataset, grader rubric, CI gate output.

Área	Un Sistema 10/10 Demuestra	Evidencia
Security	El threat model cubre tools, memory, prompts, retrieved content, credentials y users.	Threat model, mitigations, sandbox tests.
Human control	Los usuarios pueden inspeccionar, aprobar, corregir, detener o escalar comportamientos riesgosos.	Approval UI, correction flow, audit trail.
Runtime	El sistema maneja retries, timeouts, idempotency, degradación y rollback.	Runbook, retry policy, rollback plan.
Observability	Los operadores pueden inspeccionar traces, costos, latencia, llamadas a tools, eval drift e incidentes.	Dashboard, trace sample, alert thresholds.

Scorecard

Califica cada fila del 0 al 2.

Puntaje	Significado
0	Falta o solo descrito en prosa.
1	Parcialmente implementado pero no probado o no revisable.
2	Implementado, probado, documentado y asignado a un owner.

Gate	Puntaje
Goal y non-goals son explícitos.	0-2
Architecture boundary es revisable.	0-2
State ownership está documentado.	0-2
Tool contracts y permisos están probados.	0-2
Context packet es inspeccionable.	0-2
Eval gate bloquea lanzamientos inseguros.	0-2
Threat model tiene mitigaciones.	0-2
Human control existe para acciones riesgosas.	0-2
Runtime puede reintentar, reanudar, degradar y hacer rollback.	0-2
Observability explica incidentes.	0-2

Puntaje total:

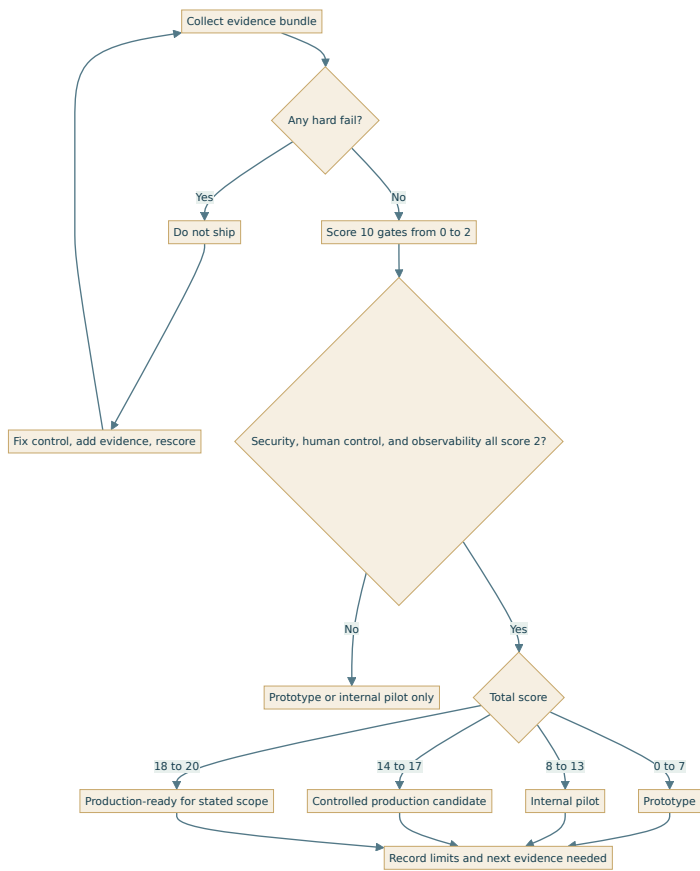
- 0-7: prototipo
- 8-13: piloto interno
- 14-17: candidato a producción controlada

- 18-20: listo para producción

No promedies la ausencia de un control de seguridad. Un sistema que toca dinero, credenciales, datos privados, infraestructura o acciones visibles para el cliente debe tener 2 en security, human control y observability.

Release Decision Flow

Usa este flujo después de calificar. Evita que fallas críticas se oculten con un puntaje total alto.



Reglas de Calificación

Califica la evidencia, no la intención:

Afirmación	Puntaje 0	Puntaje 1	Puntaje 2
“Tenemos evals.”	No existen eval fixtures.	Existen evals pero no bloquean el lanzamiento ni cubren fallas.	Evals cubren caminos esperados e inseguros, corren en CI y bloquean el lanzamiento.
“Tenemos observability.”	Existen logs pero no pueden reconstruir una ejecución.	Existen traces solo para caminos exitosos.	Los operadores pueden inspeccionar ejecuciones exitosas, fallidas, rechazadas, escaladas y con timeout.

Afirmación	Puntaje 0	Puntaje 1	Puntaje 2
“Tenemos approval.”	Los humanos son notificados después de la acción.	Approval existe pero autoriza comportamientos vagos.	Approval vincula una acción exacta, versión de policy, expiración, approver y trace ID.
“Podemos hacer rollback.”	Rollback significa volver a desplegar o borrar la función.	Existe un camino de rollback pero no está probado.	Model, prompt, policy, tool, workflow o agent behavior pueden deshabilitarse de forma independiente.

Si la evidencia no es inspeccionable por otro ingeniero, no la califiques como 2.

Release Modes

Usa el puntaje para elegir un modo de lanzamiento:

Modo	Alcance Permitido	Evidencia Requerida
Prototipo	Demos locales, datos falsos, sin efectos secundarios reales.	Etiqueta clara de que no es producción.
Piloto interno	Usuarios internos, datos limitados, acciones solo de lectura o muy acotadas.	Owner, logs, evals básicos, rollback y límites conocidos.
Candidato a producción controlada	Usuarios o datos reales con despliegue limitado y monitoreo activo.	Scorecard completa, approval para acciones riesgosas, traces, eval gate, runbook.
Listo para producción	Operación normal para el alcance previsto.	Puntaje 18-20, sin fallas críticas, rollback probado, incident-to-eval loop.

El modo de lanzamiento debe aparecer en el ADR, release notes o launch plan. Un sistema puede estar listo para producción para una tarea específica y seguir siendo prototipo para tareas adyacentes.

Hard Fails

Cualquier hard fail bloquea producción, sin importar el puntaje total.

Hard Fail	Por Qué Bloquea
El agent puede llamar tools de alto riesgo sin policy o approval.	El model puede convertir ambigüedad en acción irreversible.
Las credenciales se heredan ampliamente del proceso host.	Un bug en prompt o tool puede convertirse en escalamiento de privilegios.
El equipo no puede hacer replay de una ejecución fallida.	Los incidentes no pueden convertirse en pruebas de regresión.

Hard Fail	Por Qué Bloquea
Evals solo miden la calidad de la respuesta final.	El mal uso de tools, state oculto y trayectorias inseguras permanecen invisibles.
Las fuentes de context no tienen procedencia ni regla de frescura.	El sistema puede actuar con confianza sobre información obsoleta o no confiable.
No hay plan de rollback para cambios en prompt, policy, model o tool.	Un lanzamiento seguro requiere reversión segura.

Evidence Bundle

Una revisión de diseño seria debe adjuntar estos artifacts:

1. Architecture diagram con model, tools, state, policy, memory y user boundary.
2. ADR explicando el pattern elegido y alternativas rechazadas.
3. Tool manifest con permisos, clase de riesgo, timeouts y campos de auditoría.
4. Context packet example para una task real.
5. Trace de una ejecución exitosa y una fallida.
6. Eval dataset con al menos happy path, edge case, adversarial y regression cases.
7. Threat model con mitigaciones y evidencia de pruebas.
8. Approval o escalation flow para acciones riesgosas.
9. Runbook con alertas, triage de incidentes, rollback y owner.
10. Umbrales de costo, latencia y presupuesto.

Preguntas Mínimas Para Revisión de Producción

Haz estas preguntas antes del lanzamiento:

1. ¿Qué acción exacta puede tomar el sistema sin intervención humana?
2. ¿Qué state cambia si el model se equivoca?
3. ¿Qué llamada a tool sería la más costosa, riesgosa o irreversible?
4. ¿Qué evidencia probaría que la respuesta o acción está fundamentada?
5. ¿Qué pasa si retrieval devuelve contenido obsoleto u hostil?
6. ¿Qué pasa si un tool hace timeout después de que el efecto externo tuvo éxito?
7. ¿Qué eval fallaría si el sistema retrocede mañana?
8. ¿Quién recibe la alerta y qué puede hacer rollback?
9. ¿Qué ve el usuario cuando el agent está inseguro?
10. ¿Qué haría que el equipo apague el agent?

Dónde aprender sobre cada gate

Área de Gate	Leer
Elección de pattern	Choosing the Right Pattern
Límite de arquitectura	Architecture Before Autonomy, Agentic System Architecture

Área de Gate	Leer
Control de state y loop	Agent Loop, Goals and State
Contratos de tool	Tool Use, Tool Capability Design
Context y memory	Context Engineering, Context Budgets and Working Sets
Evals	Evaluation-Driven Agent Development, Observability and Evals
Seguridad	Agent Threat Model, Agent Security and Sandboxing
Control humano	Human Approval Gates, Agent UX and Human Trust
Runtime	Production Runtime Overview, Deployment Walkthrough
Ejemplos completos	Capstone Projects

Conclusión

Un agentic system 10/10 no es el sistema más autónomo. Es el sistema cuya autonomía es limitada, probada, observable, reversible y que justifica el costo operativo.

Arquitectura antes que autonomía

No comiences con el framework. Comienza con el límite: ¿qué puede decidir el model?

Los agents tienen sentido cuando el software no puede conocer cada paso de antemano. Se vuelven un riesgo cuando el sistema permite que el model tome decisiones que deberían pertenecer al código, como verificaciones de permisos, condiciones de detención, límites de presupuesto, transiciones de state, registros de auditoría y efectos secundarios irreversibles. Una buena arquitectura agentic separa el juicio de la autoridad. El model propone la siguiente acción útil; el sistema decide si esa acción es válida, permitida, asequible, observable y segura de ejecutar.

El model no es el sistema

Un LLM puede resumir context, proponer planes, elegir entre tools, criticar resultados y explicar compensaciones. Nada de eso lo convierte en el control plane. El control plane es dueño del goal activo, los tools permitidos, el state actual, el presupuesto, la condición de detención, el policy context, los requisitos de aprobación, las reglas de retry y fallback, y el trace de lo que ocurrió.

Cuando esas responsabilidades viven solo en un prompt, el sistema se vuelve difícil de probar y aún más difícil de operar. Un prompt puede influir en el comportamiento. No puede reemplazar un límite durable.

Un límite práctico

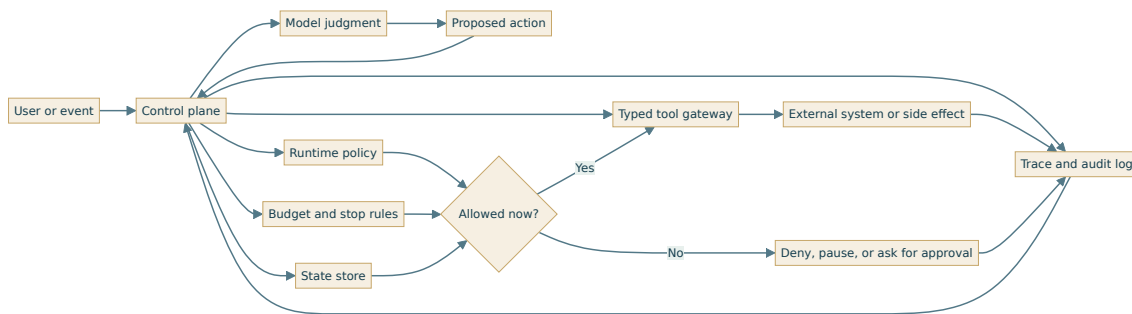
Usa esta división de responsabilidades como predeterminada:

Preocupación	Propiedad del software	Propuesta por el model
Goal	El contrato del task y los criterios de éxito	Preguntas de aclaración o subgoals
State	Almacenamiento durable, motor de workflow o servicio de aplicación	Resúmenes o escrituras de memory candidatas
Tools	Schemas tipados, permisos, timeouts y logs de auditoría	Elección de tool y argumentos
Policy	Verificaciones en runtime y puertas de aprobación	Explicación de riesgos o solicitud de escalamiento
Evaluation	Tests, rúbricas, traces y workflow de revisión	Autocrítica o score candidato
Stopping	Regla explícita de éxito, falla, presupuesto o cancelación	Afirmación de que el goal parece completo

Esto no hace que el sistema sea menos agentic. Hace que la autonomía sea legible, lo que permite operarla.

Mapa de límites

Usa este mapa al revisar un diseño. El model puede proponer trabajo que requiera juicio, pero el software debe ser dueño de la autoridad, la ejecución y la evidencia.



Este capítulo nombra el límite. [Tool Capability Design](#) y [Human Approval Gates](#) definen los contratos concretos de tool y aprobación detrás de él.

En código, el límite puede ser pequeño y explícito:

```
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 };
}
```

El model propone `ProposedAction`. El runtime decide si se niega, pausa, ejecuta o detiene.

Autonomía prematura

La autonomía prematura aparece cuando un equipo recurre a un agent loop antes de responder preguntas más simples: ¿Podría ser esto un workflow determinista con pasos asistidos por model? ¿Podría resolverse con un prompt chain y validación? ¿Podría el routing aislar los diferentes tipos de tasks? ¿Podría un human approval gate manejar los casos de riesgo? ¿Ya puede el sistema probar por qué se tomó una acción y puede un run fallido ser replayed?

Cuando la respuesta a esas preguntas es no, agregar más agents tiende a ocultar la debilidad en lugar de corregirla. Los sistemas multi-agent amplifican goals poco claros, state débil y poca observability mucho más confiablemente de lo que los resuelven.

Una reescritura común

Un diseño débil dice:

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

Eso suena eficiente, pero falta el límite. El model es dueño de demasiado: interpretación de policy, selección de evidencia, clasificación de riesgos, aprobación, autoridad sobre tools y condiciones de detención.

Un mejor diseño dice:

```
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.
```

El segundo diseño sigue usando el juicio del model, pero el model ya no es dueño de la autoridad. Propone una recomendación dentro de un sistema que puede ser revisado, probado, pausado, replayed y operado.

La prueba de ingeniería

Antes de agregar autonomía, pregunta si un operador podría inspeccionar un run fallido y responder:

1. ¿Qué goal estaba activo?
2. ¿Qué state creía el sistema?
3. ¿Qué evidencia estaba disponible?
4. ¿Qué propuso el model?
5. ¿Qué validó el software?
6. ¿Qué llamadas a tool se ejecutaron?
7. ¿Qué verificaciones de policy pasaron o fallaron?
8. ¿Por qué se detuvo el run?
9. ¿Qué cambió en el mundo exterior?

Si esas respuestas no están disponibles, lo siguiente que debes construir no es otro agent. Es state, policy, evaluation u observability.

Lista de revisión para autonomía

Usa esta lista antes de aprobar un diseño agentic:

Pregunta	Condición para aprobar
¿Qué decisión en runtime toma el model?	La decisión está nombrada y es más específica que “hacer el task”.
¿Qué es propiedad del software?	Goal, state, policy, tools, presupuesto, razones de detención y registros de auditoría están fuera del model.
¿Qué efectos secundarios pueden ocurrir?	Cada escritura, envío, pago, cambio de permisos o escritura de memory tiene un dueño y una regla de idempotencia.
¿Qué requiere aprobación?	Los umbrales de riesgo y roles de aprobación están definidos antes de la ejecución.
¿Qué se persiste?	State, evidencia, propuestas, decisiones de validación, resultados de tools y razón de detención pueden ser inspeccionados.
¿Qué se prueba?	Los evals cubren acciones permitidas, acciones denegadas, evidencia faltante, tools fallidas y detenciones por presupuesto.
¿Qué ocurre en caso de falla?	El sistema puede detenerse, reintentar, fallback, escalar o replay sin repetir efectos secundarios inseguros.

Si la lista parece demasiado pesada, probablemente el diseño necesita menos autonomía, no más.

Regla de diseño

La autonomía es un presupuesto. Úsala solo donde logre un mejor resultado que el software determinista, y rodéala de límites que hagan visible el fallo.

Capítulos relacionados

- [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](#)

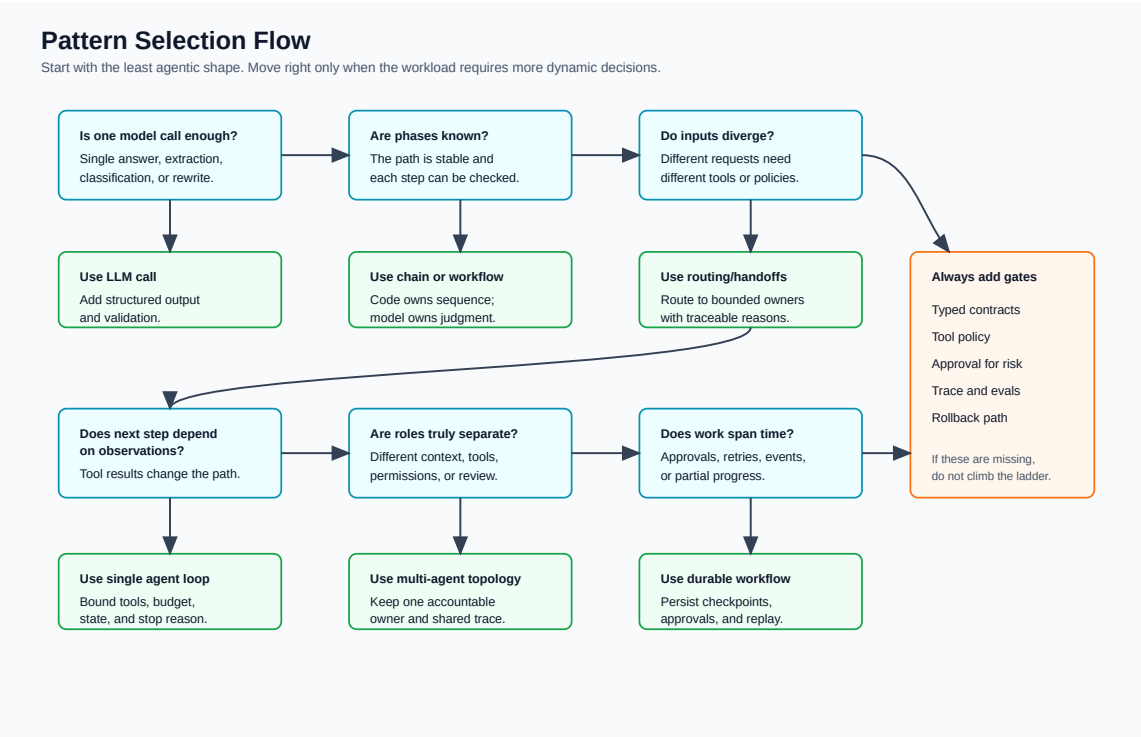
Choosing the Right Pattern

El diseño agentic es un espectro. La mejor arquitectura suele ser el agentic system menos complejo que pueda cumplir con el requerimiento con calidad, latencia, costo y riesgo aceptables.

Usa este capítulo antes de elegir un framework, agregar agents o construir una topología multi-agent. La decisión debe comenzar por la carga de trabajo, no por el pattern más poderoso disponible.

Selection Flow

Utiliza este diagrama como filtro práctico antes de subir en la escalera de autonomía. La pregunta no es “¿puede un agent hacerlo?” La pregunta es qué pattern más pequeño puede satisfacer la carga de trabajo con menos riesgo.



The Autonomy Ladder

Sube en la escalera solo cuando el peldaño inferior no pueda manejar el trabajo.

Level	Pattern Shape	Use When	Main Risk
1	LLM call	Una sola respuesta, reescritura, clasificación, extracción o resumen es suficiente.	Sin acceso a datos en vivo o acciones.
2	Prompt chain	El trabajo tiene fases conocidas y cada fase puede validarse antes de iniciar la siguiente.	Puertas frágiles o latencia innecesaria.

Level	Pattern Shape	Use When	Main Risk
3	Deterministic workflow with LLM steps	El código controla la secuencia, mientras los LLMs manejan juicios acotados dentro de los pasos.	Sobreajustar el workflow al proceso actual.
4	Routing and handoffs	Diferentes entradas requieren distintos models, prompts, tools, agents o policies.	Un mal routing envía el trabajo a la autoridad equivocada.
5	Single agent loop	El siguiente paso depende de observaciones descubiertas durante la ejecución.	Loops, mal uso de tools y errores acumulativos.
6	Orchestrator-workers	El sistema debe dividir un task desconocido en subtasks y sintetizar resultados.	El orchestrator se convierte en un plano de control oculto.
7	Multi-agent system	Diferentes especialistas necesitan context, tools, permisos o roles de revisión separados.	Sobrecarga de coordinación, fragmentación de traces y aumento de costos.
8	Autonomous long-running agent	El task abarca tiempo, fallas, eventos externos y progreso parcial.	Autonomía sin límites sin strong state, policy y observability.

Esta escalera no es un modelo de madurez. Un sistema en producción puede quedarse en el nivel 2 para siempre si el task es estable y el valor es claro.

First Questions

Haz estas preguntas antes de seleccionar un pattern:

- ¿El workflow es conocido antes de iniciar la ejecución?
- ¿El model necesita elegir el siguiente paso, o puede hacerlo el código?
- ¿El task requiere datos actuales, datos privados, tools o efectos secundarios?
- ¿El sistema puede tolerar latencia extra por múltiples llamadas al model?
- ¿Cuál es el costo máximo aceptable por ejecución?
- ¿Qué requiere aprobación humana?
- ¿Qué evidencia prueba que la respuesta o acción es correcta?
- ¿Qué state debe poder reproducirse tras una falla?
- ¿Qué podría salir mal si el model es persuasivo pero incorrecto?

Si las respuestas no son claras, comienza con un deterministic workflow y agrega comportamiento agentic solo donde el workflow necesite juicio del model.

Selection Outputs

No dejes la selección de pattern como una conversación. Documenta la decisión en una forma que otro ingeniero pueda revisar.

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:

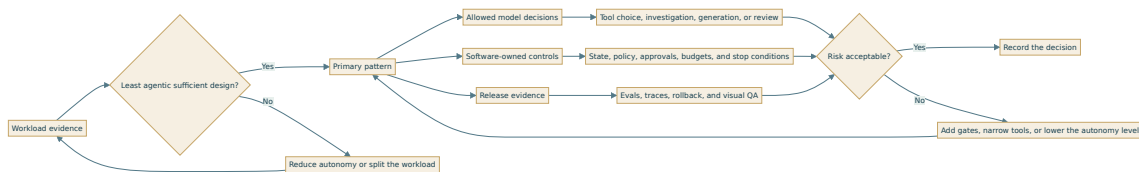
Eval cases required before release:

Fallback or rollback path:

La línea más importante es “por qué este es el diseño agentic suficiente más simple”. Si la respuesta es “porque los agents son flexibles”, el diseño no está listo. Nombra la incertidumbre real que el model debe manejar: intención ambigua, siguiente paso desconocido, evidencia incompleta, elección variable de tool o investigación abierta.

Decision Record Map

Utiliza este mapa para convertir la salida de selección en un artifact de revisión de ingeniería. Una elección de pattern está lista cuando la evidencia de la carga de trabajo, el límite de autonomía, los controles y la evidencia de liberación pueden revisarse por separado.



Selection Matrix

Workload Signal	Prefer	Avoid
Secuencia fija conocida	Prompt Chaining and Gates	Autonomous agents
Uno de varios tipos de tasks conocidos	Routing and Handoffs	Un solo giant prompt
Subtasks independientes	Parallel Agents	Cadenas secuenciales que generan latencia evitable
Subtasks desconocidos	Orchestrator-workers	Listas de tasks codificadas
La calidad mejora con revisión	Evaluator-Optimizer	Generación de una sola pasada
Efectos secundarios de alto riesgo	Human Approval Gates	Ejecución directa de tools desde la salida del model
Gran superficie de tools	MCP-first Tool Use with policy	Tools amplios sin tipado

Workload Signal	Prefer	Avoid
Trabajo de larga duración	Durable Workflows and Goals and State	Loops en memoria ocultos
Datos sensibles o cumplimiento	Policy Enforcement and audit logs	Controles solo con prompt
Respuestas con mucha recuperación	Agentic RAG Systems	Respuestas ciegas del model
La depuración es importante	Observability and Evals	Logs solo de respuesta final

El mismo producto puede combinar varias filas. Por ejemplo, un workflow de reembolso de soporte puede usar routing para clasificar la solicitud, pasos de deterministic workflow para consulta de cuenta, RAG para evidencia de policy, aprobación humana para excepciones y un agent loop solo para investigación abierta.

Example Choices

Workload	Start With	Add Autonomy Only If
Reescribir o resumir texto proporcionado por el usuario	Una llamada al model con structured output	El sistema debe obtener context faltante o revisar contra evidencia externa.
Generar respuestas de soporte a partir de policies conocidas	Workflow con retrieval y validación	La solicitud requiere investigación entre tools con siguientes pasos desconocidos.
Clasificar tickets por área de producto	Routing and handoffs	El router necesita hacer preguntas de seguimiento o inspeccionar sistemas en vivo.
Investigar una pregunta técnica	Agentic RAG o agent loop con retrieval tools	El sistema debe decidir si la evidencia es suficiente o continuar buscando.
Procesar reembolsos	Deterministic workflow con resumen de evidencia asistido por model	La policy tiene excepciones ambiguas que requieren investigación acotada.
Coordinar operaciones de entrega	Workflow más roles explícitos	Agents separados necesitan context, tools, permisos o auditorías distintas.
Mantener un task de fondo de larga duración	Durable workflow	Las decisiones en runtime dependen de eventos futuros, progreso parcial o recuperación.

Estos ejemplos son intencionalmente conservadores. Comienza con el pattern sencillo y promueve solo la parte del sistema que ha demostrado necesitar comportamiento dinámico.

Workflows vs Agents

Un workflow usa código para controlar el camino. Un model puede clasificar, extraer, resumir, criticar o generar dentro de un paso, pero el software decide qué ocurre después.

Un agent usa el model para decidir partes del camino. Observa el state, decide la siguiente acción, invoca tools, lee resultados y continúa hasta que se completa el goal o se alcanza un límite.

Prefiere un workflow cuando el proceso es estable, cuando la corrección depende de reglas deterministas, cuando la latencia o el costo deben mantenerse bajos, cuando el sistema debe ser fácil de auditar y cuando los operadores necesitan modos de falla predecibles. Prefiere un agent cuando el proceso es abierto, cuando el sistema debe descubrir información faltante, cuando la elección de tools depende de observaciones intermedias, cuando el número de pasos es desconocido o cuando un workflow fijo se ramificaría en un árbol inmantenible.

Complexity Budget

Cada agent adicional, llamada al model, tool, memory store y evaluator consume parte del complexity budget del sistema, así que gástalo deliberadamente. Vale la pena agregar complejidad cuando trae un resultado concreto: mayor tasa de tareas completadas, menor esfuerzo humano, mejor fundamentación de evidencia, efectos secundarios más seguros, menor costo mediante routing o models más pequeños, mejor capacidad de depuración o límites de propiedad más claros.

No agregues complejidad solo porque un pattern es popular. Un multi-agent system que reemplaza un workflow confiable de cuatro pasos usualmente hace el producto más lento, más difícil de probar y más difícil de explicar.

Ruta de Evolución de Patterns

Una ruta de evolución práctica se ve así:

1. Comienza con un solo prompt o un workflow determinista.
2. Agrega structured outputs y validación.
3. Agrega retrieval o tools donde el model necesita evidencia o acción.
4. Agrega routing cuando las entradas divergen en caminos distintos.
5. Agrega evaluator loops donde la calidad mejora con crítica.
6. Agrega durable state cuando el trabajo abarca varios pasos o sesiones.
7. Agrega agents solo donde se requieren decisiones dinámicas.
8. Agrega coordinación multi-agent cuando roles separados necesitan context, tools o permisos separados.

Cada paso debe mejorar un resultado medido. Si un pattern no mejora la precisión, confiabilidad, latencia, costo, seguridad o mantenibilidad, elimínalo.

Errores Comunes de Selección

- Elegir coordinación multi-agent cuando la necesidad real es routing.
- Usar un agent loop porque el workflow no fue documentado.

- Darle al model tools amplias cuando bastaría un paso de workflow específico.
- Agregar reflection cuando el sistema necesita un mejor evaluator o un test set.
- Agregar memory antes de definir qué state debe ser durable, corregible y eliminable.
- Tratar ejemplos de framework como arquitectura de producción.
- Optimizar para autonomía antes de medir completion de task, costo, latencia o riesgo.

Mínimos para Producción

Antes de que un pattern maneje usuarios, dinero, datos privados, infraestructura o comunicación con clientes, el sistema debe tener:

- entradas y salidas tipadas;
- condiciones de parada explícitas;
- permisos de tool limitados;
- llamadas de model y tool trazables;
- transiciones de state reproducibles;
- datasets de eval para el comportamiento esperado;
- comportamiento de fallback para fallas de model, retrieval y tool;
- aprobación humana para acciones de alto riesgo;
- rollback o remediación para acciones incorrectas.

Este mínimo aplica también para sistemas simples. Los sistemas pequeños fallan más rápido porque sus límites suelen ser implícitos.

Capítulos Relacionados

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

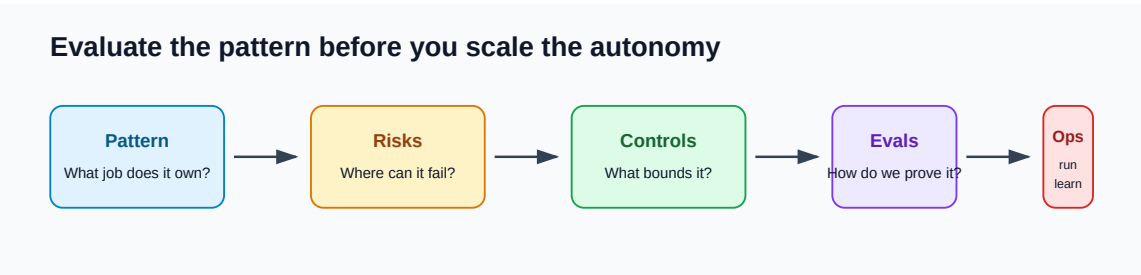
Lista de verificación para la evaluación de patrones

Un pattern no está listo solo porque suena bien. Está listo cuando el equipo puede explicar qué trabajo le pertenece, dónde puede fallar, qué limita la falla, cómo se evalúa y cómo se comporta en producción.

Esta lista de verificación es el lente de revisión compartido para el libro. Úsala antes de elegir un pattern, antes de componer varios patterns y antes de promover un agentic workflow a producción.

Descarga la hoja de trabajo reutilizable: [pattern evaluation scorecard](#).

Para conocer el ciclo de ingeniería detrás de esta lista, consulta [Evaluation-Driven Agent Development](#). Usa la lista de verificación para revisar un pattern. Usa el capítulo de desarrollo para convertir la revisión en datasets, fixtures, release gates y retroalimentación de producción.



Versión corta

- Todo pattern debe responder cinco preguntas:
1. ¿Qué responsabilidad posee este pattern?
 2. ¿Qué nuevo riesgo introduce?
 3. ¿Qué control mantiene ese riesgo limitado?
 4. ¿Qué eval demuestra que el control funciona?
 5. ¿Qué señal en producción nos indica que está desviándose?

Si el pattern no puede responder esas preguntas, probablemente aún no es un pattern. Es solo una idea de implementación.

Tabla de evaluación

Usa esta tabla como plantilla de revisión predeterminada.

Área	Pregunta	Evidencia a buscar
Goal	¿Qué goal de usuario o sistema posee el pattern?	Un contrato de task, criterios de éxito, criterios de rechazo y responsable.
Boundary	¿Qué queda fuera de la responsabilidad del pattern?	Un handoff claro, contrato del llamador o ruta de escalamiento.

Área	Pregunta	Evidencia a buscar
Autonomy	¿Qué decide el model y qué decide el software?	Una división entre propuesta, validación, ejecución y detención.
Loop	¿El pattern puede repetirse?	Máximo de pasos, máximo de llamadas a tools, timeout, presupuesto de reintentos y razón de detención.
Tools	¿Qué puede leer o cambiar el pattern?	Lista permitida de tools, validación de schema, revisiones de permisos y eventos de auditoría.
State	¿Qué state se lee, escribe o persiste?	Dueño del state, reglas de actualización, comportamiento de replay y memory write policy.
Context	¿Qué evidencia entra al conjunto de trabajo?	Elegibilidad de fuente, reglas de recuperación, revisiones de frescura y presupuesto de context.
Security	¿Qué puede influenciar una entrada no confiable?	Threat model, controles de prompt-injection, sandboxing y puertas de aprobación.
Evaluation	¿Qué falla debe detectarse antes del release?	Golden tasks, casos negativos, trajectory evals, tools simulados y regression fixtures.
Observability	¿Una ejecución fallida puede explicarse después?	Trace ID, model spans, tool spans, decisiones, policy denials, costos y razón de detención.
Operations	¿El pattern puede revertirse o deshabilitarse?	Prompts versionados, manifiestos de tools, rutas de model, feature flags y circuit breakers.

Puntuación de revisión

Usa una puntuación simple cuando el pattern vaya a una revisión de diseño o release gate.

Puntuación	Significado	Decisión
0	Falta o solo a nivel de prompt.	Bloquear. El control no es real.
1	Descrito pero no implementado ni probado.	No liberar. Convierte la descripción en código, configuración o pruebas.
2	Implementado pero con pruebas débiles o difícil de inspeccionar.	Liberar solo para uso interno de bajo riesgo.
3	Implementado, probado, trazable y con responsable.	Aceptar para el nivel de riesgo declarado.

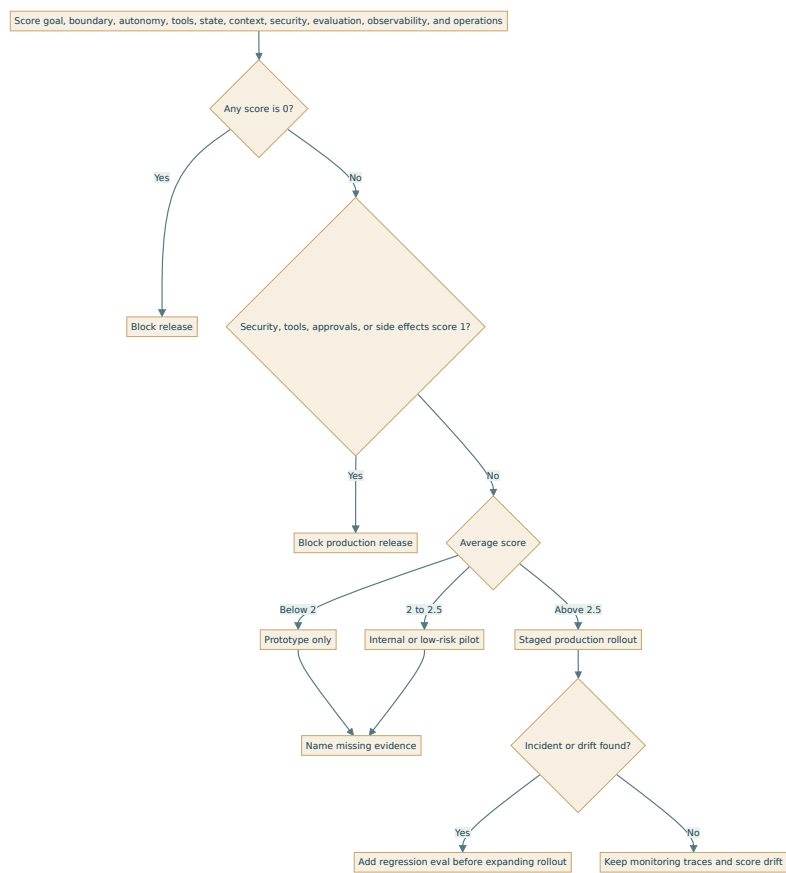
Evalúa estas áreas: goal, boundary, autonomy split, tools, state, context, security, evaluation, observability y operations. Un pattern en producción no debe tener ningún puntaje **0**. Un pattern que maneje dinero, datos privados, infraestructura, comunicación con clientes o memory durable no debe tener ningún puntaje **1**.

El objetivo no es crear burocracia. La puntuación previene una falla común en la revisión de diseño: todos están de acuerdo en que la idea es buena, pero nadie prueba que el boundary existe.

Usa la [pattern evaluation scorecard](#) descargable cuando la revisión deba dejar un registro auditable. Captura la puntuación, responsable, evidencia, modo de release, brechas bloqueantes, riesgos aceptados y siguiente evidencia para cada área.

Ruta de decisión de puntuación

Usa esta ruta durante la revisión. Convierte la puntuación en una decisión de release sin ocultar brechas de alto riesgo detrás de un buen promedio.



Mínimo requerido por tipo de pattern

Diferentes patterns requieren diferentes pruebas.

Tipo de pattern	Guía mínima de evaluación
Prompt chain	Valida la salida de cada paso, controla transiciones y prueba resultados intermedios malformados.
Router	Prueba solicitudes ambiguas, tasks no soportadas, rutas de alto riesgo y comportamiento de fallback.
Agent loop	Prueba condiciones de detención, selección de tools, recuperación de malas observaciones y agotamiento de presupuesto.
Tool-use pattern	Prueba tools prohibidos, argumentos inválidos, idempotencia, timeouts y policy denials.

Tipo de pattern	Guía mínima de evaluación
RAG o memory pattern	Prueba relevancia de fuentes, evidencia obsoleta, evidencia faltante, cobertura de citas y memory writes inseguros.
Evaluator o reflection pattern	Prueba aprobaciones falsas, sobrecorrección, ambigüedad en rúbricas y manejo de desacuerdos.
Multi-agent pattern	Prueba aislamiento de context, aislamiento de permisos, precisión de merge, fallas de worker y responsabilidad final.
Human approval pattern	Prueba criterios de escalamiento, visibilidad del aprobador, comportamiento ante timeout y registros de auditoría.
Production runtime pattern	Prueba replay, rollback, canary gates, conversión de incidentes a eval y diagnóstico de operadores.

Un contrato de revisión breve

Para revisiones de diseño ligeras, mantén el contrato corto:

```

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"
```

El contrato es intencionalmente simple. Debe ser fácil de revisar en un pull request, fácil de convertir en pruebas y fácil de comparar contra un trace de producción.

Reglas de decisión de release

Usa estas reglas después de puntuar:

- Cualquier **0**: bloquear release.
- Cualquier **1** en security, tools, approvals o side effects: bloquear release a producción.
- Promedio menor a 2: mantenerlo como prototipo.
- Promedio de 2 a 2.5: permitir solo piloto interno o de bajo riesgo.
- Promedio mayor a 2.5 sin brechas de alto riesgo: permitir rollout a producción escalonado.
- Cualquier incidente relacionado con el pattern: agrega un regression eval antes de expandir el rollout.

Si el equipo no está de acuerdo con una puntuación, registra el desacuerdo. Un desacuerdo usualmente significa que el boundary de ownership, la evidencia o la tolerancia al riesgo aún no están claros.

Señales comunes de falla

Atento a estas señales durante la selección de patterns:

- El pattern no tiene un único responsable.
- El model es responsable de los chequeos de permisos.
- El loop se detiene solo cuando el model dice que terminó.
- La lista de tools es más amplia que el task.
- Memory writes ocurren como efecto secundario de la conversación.
- El eval solo revisa la respuesta final, no la trayectoria.
- El trace no puede mostrar por qué se llamó a un tool.
- Multi-agent routing se usa para ocultar responsabilidades poco claras.
- El camino de fallback es “preguntar al model de nuevo”.
- El rollback requiere reconstrucción manual de prompts, tools o policies.

Estos no son problemas de estilo. Son problemas de arquitectura.

Regla de diseño

Elige el pattern más simple cuyos riesgos puedas limitar y cuyo comportamiento puedas evaluar. Si no puedes probar el boundary, el boundary aún no es real.

Capítulos relacionados

- [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](#)

De patrones a sistemas

Los patrones solo son útiles cuando te ayudan a construir un sistema. Un agent de producción rara vez es un solo patrón. Normalmente es un workflow con algunas decisiones mediadas por el model, un límite de retrieval, algunas tools, state, policy, aprobaciones, evals y observability. Los nombres de los patrones importan mucho menos que la forma en que esas piezas encajan.

Aquí es donde muchos proyectos de agents fallan. El equipo agrega un loop, luego memory, luego tools, luego un segundo agent, luego un judge, luego un motor de workflow. Cada adición parece razonable por sí sola, y el resultado es un sistema que nadie puede explicar. La composición es la disciplina que lo previene.

Empieza con la carga de trabajo

No compongas patrones desde un catálogo. Compónlos desde la carga de trabajo. Empieza preguntando qué intenta lograr realmente el usuario, qué pasos se conocen de antemano y cuáles requieren juicio del model, qué evidencia debe recuperarse, qué acciones tienen efectos secundarios, qué state debe sobrevivir a una falla, qué necesita aprobación y qué debe ser observable después de la ejecución.

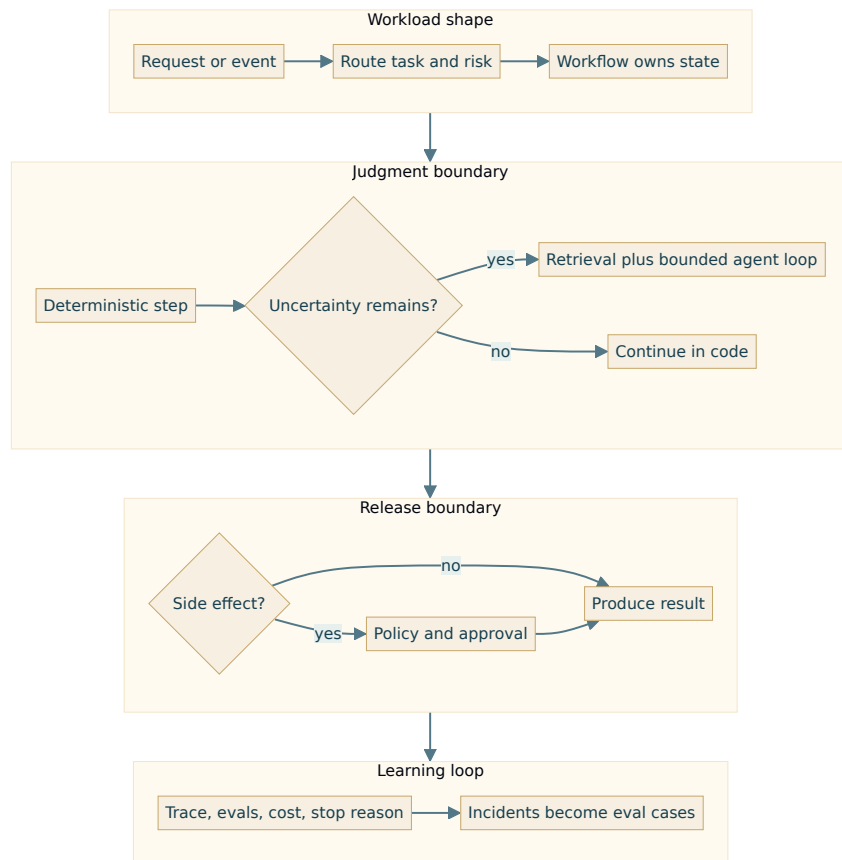
Las respuestas señalan las partes. Si el workflow es conocido, deja el control al código. Si el siguiente paso depende de observaciones, agrega un agent loop. Si el task necesita evidencia, agrega retrieval. Si el task puede cambiar el mundo exterior, agrega policy y aprobación. Si las fallas importan, agrega state durable y replay. Eso es composición: cada patrón se gana su lugar en vez de llegar por defecto.

Una forma común

Muchos agentic systems útiles siguen aproximadamente esta forma:

1. El punto de entrada recibe una solicitud, evento o task programada.
2. Un router clasifica el task, el riesgo y la capability requerida.
3. El workflow carga el state, el policy context y la memory relevante.
4. Retrieval reúne evidencia cuando la respuesta depende del conocimiento.
5. El agent loop maneja incertidumbre acotada dentro del workflow.
6. Las tools se ejecutan mediante schemas tipados y revisiones de permisos.
7. Una puerta de aprobación pausa efectos secundarios de alto riesgo.
8. Evaluators revisan la trayectoria, evidencia, output y policy.
9. El runtime almacena traces, costos, decisiones, llamadas a tools y razones de detención.
10. Incidentes y correcciones alimentan el eval suite.

No todos los sistemas necesitan cada paso. Lo que se mantiene constante es la propiedad. El código es dueño del flujo, state, policy y persistencia; el model es dueño del juicio acotado dentro de esas restricciones.



Usa este diagrama como prueba de composición. Si un patrón propuesto no puede ubicarse en el mapa con un dueño, entrada, salida y modo de falla, probablemente no está listo para entrar al sistema.

Reglas de composición

Revisa estas reglas antes de agregar otro patrón.

Regla	Por qué importa
Un componente es dueño del goal.	Sin propiedad del goal, los agents optimizan tasks diferentes.
Un componente es dueño del state.	Sin propiedad del state, el replay y la recuperación se vuelven conjeturas.
Las llamadas a tools cruzan un límite de policy.	Sin policy, las propuestas del model se convierten en acciones demasiado rápido.
Las escrituras a memory son eventos explícitos.	Sin disciplina en memory, persiste context obsoleto o inseguro.
Los loops tienen condiciones de parada.	Sin condiciones de parada, la autonomía se convierte en mayor costo y latencia.
Los evals inspeccionan trayectorias.	Sin trajectory evals, caminos inseguros pueden producir respuestas plausibles.
Los traces conectan decisiones con efectos.	Sin traces, las fallas no pueden convertirse en mejores pruebas.

Estas reglas importan más que el framework. Un framework puede ayudarte a implementarlas; no puede decidir las por ti.

Registro de composición del sistema

Para cualquier agentic system no trivial, escribe un breve registro de composición antes de la implementación. Debe caber en un pull request o en un registro de decisión de arquitectura.

```
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"
```

El registro obliga a que cada patrón justifique su existencia. Si un patrón no tiene trabajo, dueño o release blocker, elimínalo o déjalo fuera de la primera versión.

Mala composición

La mala composición suele tener el mismo olor: el model tiene demasiado control. El agent infiere el goal a partir de un historial de conversación vago, elige tools sin revisión de permisos y escribe en memory sin clasificación ni revisión. Los reintentos ocurren dentro de loops ocultos. Los subagents reciben toda la

conversación en vez de un task acotado. Los evaluators revisan el tono pero no la evidencia. La respuesta final se registra mientras se pierde la trayectoria, y un multi-agent system no tiene un solo dueño para la síntesis final.

Estos sistemas pueden lucir impresionantes en una demo. Son difíciles de operar porque nadie puede decir dónde vive la responsabilidad.

Buena composición

La buena composición suele ser aburrida. Un sistema de reembolsos de soporte, por ejemplo, podría ejecutar un workflow determinista para la recepción y búsqueda de cuentas, un router para el tipo de solicitud y riesgo, y retrieval para la policy de reembolso actual. Usaría structured output para los campos extraídos y la recomendación, un pequeño agent loop solo para la investigación de información faltante, enforcement de policy antes de cualquier acción de reembolso, aprobación humana para excepciones, y observability y evals en toda la ejecución.

El sistema es agentic donde existe incertidumbre y determinista donde importa el control. Ese es todo el truco.

Un boceto simple de composición podría verse así:

```
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);
}
```

Solo un paso usa un agent loop. El workflow sigue siendo dueño de la ruta, state, policy, aprobación y efectos secundarios.

Cuándo dividir agents

No dividas agents porque el task parezca grande. Divídelos cuando el límite aporte algo concreto: context windows separadas, tools separadas, permisos separados, equipos o propiedad separados, trabajo en paralelo, revisión independiente o diferentes responsabilidades de cara al usuario.

Las razones débiles son fáciles de reconocer una vez que las nombras. El diagrama de arquitectura se ve más avanzado. Cada prompt suena como un título de trabajo diferente. El equipo quiere un multi-agent system. El diseño de un solo agent tiene goals poco claros o tools débiles, así que dividirlo parece progreso. No lo es. Dividir agents no arregla una arquitectura débil; la multiplica.

Lista de verificación para revisión de diseño

Antes de aprobar un agentic system compuesto, pregunta:

1. ¿Qué partes son workflows deterministas?
2. ¿Qué partes son decisiones mediadas por el model?
3. ¿Quién es dueño del goal activo?
4. ¿Quién es dueño del state durable?
5. ¿Qué tools pueden causar efectos secundarios?
6. ¿Qué policy se ejecuta antes de esos efectos secundarios?
7. ¿Qué evidencia se requiere para la respuesta final?
8. ¿Qué memory se puede escribir, actualizar o eliminar?
9. ¿Cuáles son las condiciones de parada?
10. ¿Qué evals bloquean el release?
11. ¿Qué trace permite a un operador reproducir la ejecución?
12. ¿Qué pasa cuando el model está equivocado pero es persuasivo?

Si el diseño no puede responder a esto, no está listo para más autonomía.

Regla de diseño

Compón patrones solo cuando cada uno tenga un trabajo, un dueño y un modo de falla que puedas probar.

Capítulos Relacionados

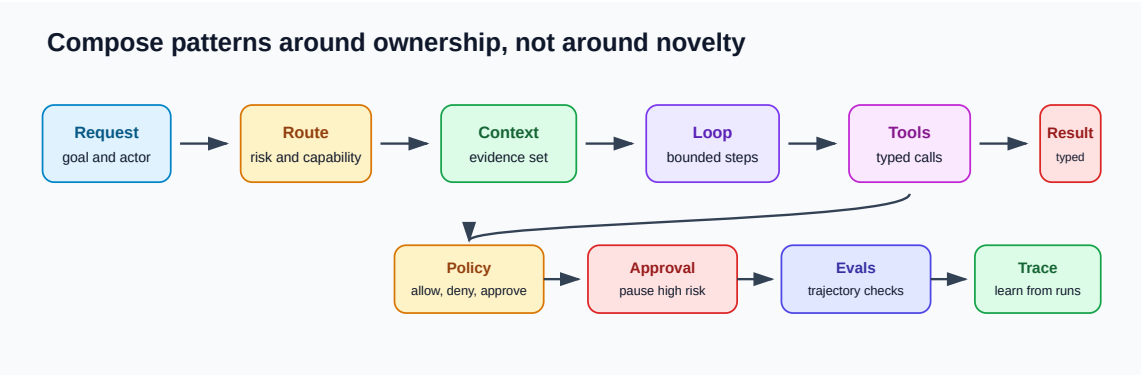
- [Architecture Before Autonomy](#)
- [Choosing the Right Pattern](#)
- [Pattern Evaluation Checklist](#)
- [Pattern Composition Playbook](#)
- [Agent Development Lifecycle](#)
- [Evaluation-Driven Agent Development](#)
- [Agentic System Architecture](#)
- [Reference Architecture](#)

Pattern Composition Playbook

Los patterns se vuelven útiles cuando se componen en un sistema que alguien puede operar. El objetivo no es usar la mayor cantidad posible de patterns. El objetivo es poner cada responsabilidad en el lugar correcto: el workflow controla el flujo, la policy controla los permisos, los tools controlan los efectos secundarios, retrieval controla la evidencia, el loop controla la incertidumbre acotada, los evals controlan la prueba y observability controla el aprendizaje a partir de fallas.

La forma más fácil de dañar un agentic system es componer patterns por intuición. Agregar memory porque el agent olvidó algo. Agregar un segundo agent porque el task parece grande. Agregar reflection porque la respuesta fue débil. Agregar tools porque el model necesita datos. Cada decisión puede sonar razonable, pero el resultado puede ser un sistema sin dueño, sin condición de parada, sin límite de evidencia y sin forma de replay de fallas.

La composición debe empezar por la propiedad.



La Pregunta de Composición

Antes de agregar un pattern, pregunta qué problema le pertenece.

Presión	Pattern a considerar	Qué debe controlar el límite
El task tiene pasos conocidos.	Prompt chain o deterministic workflow.	Workflow code.
El siguiente paso depende de observaciones.	Agent loop.	Loop controller y stop rules.
La respuesta necesita evidencia.	RAG, semantic recall o knowledge-bound agent.	Retrieval y source policy.
El model debe invocar tools.	Tool use o MCP-first tool use.	Tool manifest, schema, permission y audit.
La acción es riesgosa.	Policy enforcement y human approval gate.	Runtime policy y durable workflow.

Presión	Pattern a considerar	Qué debe controlar el límite
La calidad se puede juzgar mejor que lo generado.	Evaluator-optimizer o reflection.	Rubric, evidence checks y revision budget.
El trabajo necesita context o permisos separados.	Supervisor-worker o parallel agents.	Coordinator, worker contracts y merge policy.
La falla debe poder reproducirse.	Durable workflow, observability y eval feedback loop.	Runtime, trace store y eval suite.

Si ningún pattern controla un problema concreto, no lo agregues.

Para ejemplos concretos de extremo a extremo, lee [Vertical Slice Examples](#) después de este capítulo. Los slices muestran la misma regla de composición aplicada a workflows de soporte, código e investigación.

Una Composición Predeterminada

Para muchos sistemas en producción, la composición predeterminada es:

1. enrutar la solicitud por tipo de task, riesgo y capability;
2. cargar el durable state, policy context e identidad del solicitante;
3. armar un pequeño conjunto de trabajo con evidencia aprobada;
4. ejecutar un agent loop acotado solo donde exista incertidumbre;
5. ejecutar tools mediante schemas tipados y verificaciones de permisos;
6. aplicar policy antes de efectos secundarios o escrituras en memory;
7. pausar para aprobación cuando el riesgo lo requiera;
8. evaluar trayectoria, evidencia, output y comportamiento de policy;
9. registrar traces, costos, decisiones, llamadas a tools y razones de parada;
10. convertir fallas de producción en regression evals.

Esa secuencia no es un framework. Es un mapa de responsabilidades. Un sistema simple puede omitir varios pasos. Un sistema de alto riesgo puede necesitar todos.

Scorecard de Composición

Antes de aceptar una composición, califica cada pattern agregado según la responsabilidad que afirma controlar.

Verificación	Condición para aprobar
Job	El pattern resuelve una presión de carga de trabajo nombrada, no solo un deseo vago de flexibilidad.
Owner	Un componente o equipo es dueño del comportamiento del pattern y de la respuesta ante fallas.
Boundary	El pattern tiene entradas, salidas, autoridad y no-responsabilidades claras.
Risk	Se nombran los nuevos modos de falla del pattern.

Verificación	Condición para aprobar
Control	Hay un límite, policy, gate o fallback controlado por software para cada riesgo mayor.
Eval	Al menos un eval bloqueante prueba que el límite funciona.
Trace	Un trace de producción puede mostrar la decisión y efecto del pattern.
Removal	El equipo sabe qué se rompería si se elimina el pattern.

Si un pattern falla `job`, `owner` o `boundary`, elimínalo de la composición. Si falla `eval` o `trace`, no lo lles a producción hasta que exista la prueba.

Composición 1: Investigación de Reembolsos en Soporte

Usa esto cuando el agent investiga un reembolso pero no debe emitir dinero directamente.

Responsabilidad	Pattern
Intake y enrutamiento	Routing y handoffs.
Evidencia	Semantic recall y typed business tools.
Investigación	Bounded agent loop.
Ejecución de tools	Tool use con read tools restringidos y write tools solo en borrador.
Seguridad	Policy enforcement antes de acciones de reembolso.
Control humano	Approval gate para reembolsos de alto valor o excepciones.
Calidad	Evaluator revisa evidencia, policy y recomendación.
Operaciones	Trace, replay y feedback de incident-to-eval.

El límite importante es la autoridad financiera. El agent puede investigar, citar policy y redactar una solicitud de reembolso. No debe emitir el reembolso. La acción de reembolso pertenece a un workflow respaldado por policy, usualmente con aprobación.

```

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);
}

```

Esto es agentic, pero solo en la investigación. El workflow controla enrutamiento, policy, aprobación y efectos secundarios.

Composición 2: Asistente de Policy con Evidencia

Usa esto cuando el assistant responde a partir de policy aprobada, documentación o fuentes de compliance.

Responsabilidad	Pattern
Elegibilidad de fuente	Policy enforcement y knowledge-bound agent.
Recuperación de evidencia	Semantic recall y RAG.
Control de context	Context budgets y working sets.
Forma de la respuesta	Structured output.
Preguntas no soportadas	Refusal o human escalation.
Calidad	Citation coverage y missing-evidence evals.
Operaciones	Monitoreo de frescura de fuentes y revisión de trace.

El límite crítico es la evidencia. El assistant no debe responder porque el model “sabe” algo. Debe responder porque el sistema recuperó fuentes elegibles y la respuesta las cita.

```

{
  "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": []
}
```

Cuando falta evidencia, la salida correcta no es una respuesta más débil. Es `missing_evidence`, `conflicting_evidence`, `refused` o `needs_human`.

Composición 3: Investigación y Revisión Multi-Agent

Usa esto cuando un task se beneficia de flujos de trabajo separados y revisión independiente.

Responsabilidad	Pattern
Decomposición	Supervisor-worker.
Aislamiento de workers	Scoped context y permisos de tools.
Trabajo en paralelo	Parallel agents cuando el trabajo es independiente.
Revisión	Evaluator-optimizer o reviewer worker dedicado.
Merge	Supervisor merge policy.
Responsabilidad final	Un solo owner final y razón de parada.
Operaciones	Traces por worker y replay de merge-decision.

El límite crítico es la propiedad final. Varios agents pueden producir evidencia, borradores o críticas. Un componente debe controlar la síntesis final.

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

Si cada worker ve el mismo context y tools, probablemente no tienes un multi-agent system útil. Solo tienes llamadas duplicadas al model.

Qué No Componer

Algunas combinaciones son riesgosas a menos que exista un límite claro:

Composición riesgosa	Por qué falla	Control
Agent loop más tools amplios.	El loop puede amplificar una mala decisión de tool.	Tools limitados, policy, aprobación, reglas de detención.

Composición riesgosa	Por qué falla	Control
RAG más escrituras en memory.	Los errores recuperados se vuelven hechos durables.	Revisión de escrituras en memory y metadatos de fuente.
Evaluator sin evidencia.	El evaluator califica confianza sin pruebas.	Verificación de citas y trayectoria.
Multi-agent más tool compartido.	Cada worker puede causar el mismo daño.	Alcances de tool por worker.
Aprobación humana más solicitud vaga.	Humanos aprueban sin saber la acción.	Solicitud de aprobación tipada y vinculación a acción exacta.
Policy solo en prompts.	El model puede ignorar o reinterpretar la policy.	Ejecución de enforcement en runtime antes de ejecutar.

Componer no se trata de más partes. Se trata de mejores límites.

Revisión de Diseño

Antes de aprobar un sistema compuesto, pregunta:

1. ¿Quién es dueño del goal?
2. ¿Quién es dueño del state?
3. ¿Quién es dueño de la evidencia?
4. ¿Quién es dueño de los permisos de tool?
5. ¿Quién es dueño de la policy?
6. ¿Quién es dueño de la aprobación?
7. ¿Quién es dueño de la evaluación?
8. ¿Quién es dueño de la respuesta o acción final?
9. ¿Quién es dueño del replay y rollback?
10. ¿Qué falla en producción se convierte en un nuevo eval?

Si la respuesta es “el prompt” para cualquier responsabilidad de alto riesgo, la arquitectura no está lista.

Regla de Diseño

Compón patterns solo cuando cada pattern agregado tenga una función, un límite, un responsable y un eval que pueda fallar.

Capítulos Relacionados

- [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 divide una task en pasos conocidos de LLM y coloca validation gates entre ellos. Es una de las formas más seguras de agregar juicio del model sin darle control sobre todo el workflow.

Usa este pattern cuando el orden del trabajo es conocido pero los pasos individuales requieren comprensión de lenguaje, generación, clasificación o extracción.

Intent

Ejecuta varias llamadas al model limitadas en una secuencia fija. Cada llamada recibe un input acotado, produce un output tipado y pasa por un gate determinista antes de que corra el siguiente paso.

El model realiza juicios locales. El código controla el workflow.

Usa Cuando

- La task tiene fases estables.
- Cada fase puede probarse de forma independiente.
- Los outputs intermedios pueden validarse.
- Quieres menor riesgo que un agent loop autónomo.
- Necesitas puntos de falla claros y comportamiento de reintento.

Ejemplos comunes:

- clasificar una request, extraer fields, verificar fields y luego redactar una respuesta;
- resumir un documento, extraer claims, revisar claims y luego producir una respuesta final;
- generar código, correr tests, criticar fallas y luego revisar;
- recuperar evidence, normalizar citations y luego sintetizar una respuesta.

Evita Cuando

- Los pasos son desconocidos antes de iniciar la ejecución.
- La cadena se ramificaría en docenas de casos especiales.
- Ningún gate puede verificar los outputs intermedios.
- El mismo input suele requerir una secuencia diferente de trabajo.
- La latencia de múltiples llamadas es inaceptable.

Si la cadena sigue creciendo con ramas condicionales, considera [Routing and Handoffs](#) o un [Agent Loop](#).

Arquitectura

Input

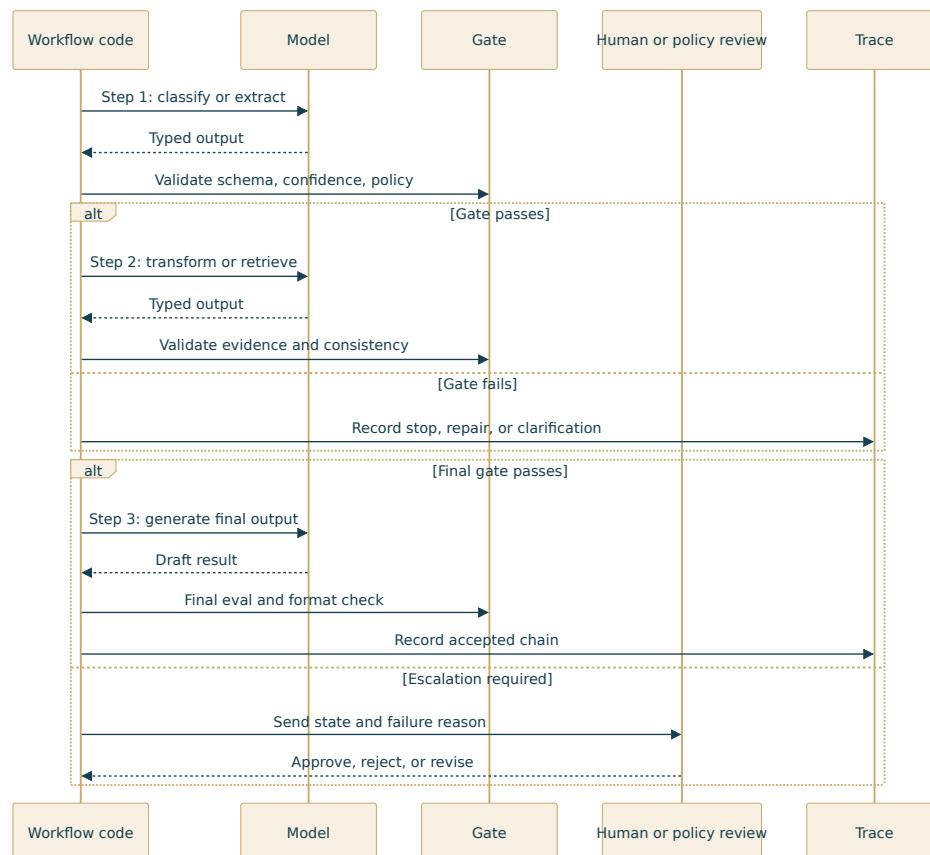
-> Step 1: `classify or extract`

-> 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

Los gates deben ser deterministas siempre que sea posible. Un gate puede usar un model como evaluator, pero el gate aún necesita criterios explícitos y un output claro de aprobación/rechazo.



Usa el diagrama como prueba de diseño. Cada flecha debe corresponder a una transición de state tipada, una decisión de gate nombrada y una regla acotada de reintento o escalamiento.

Tipos de Gate

Gate	Verifica	Comportamiento ante falla
Schema gate	Forma JSON, fields requeridos, enums, rangos numéricos	Pedir al model reparar o fallar rápido.
Evidence gate	Citations, frescura de source, cobertura de retrieval	Recuperar más evidence o escalar.
Policy gate	permisos, user role, acciones restringidas	Bloquear, redactar o pedir aprobación.
Confidence gate	confianza en clasificación, ambigüedad, inputs faltantes	Hacer una pregunta de aclaración.

Gate	Verifica	Comportamiento ante falla
Consistency gate	claims coinciden con el state y evidence previos	Repetir el paso con context más acotado.
Cost gate	conteo de llamadas al model, token budget, time budget	Devolver resultado parcial o diferir.
Human gate	decisión subjetiva o de alto impacto	Pausar hasta aprobación.

Las cadenas más robustas mezclan varios tipos de gate. Un workflow de soporte puede usar schema gates para fields extraídos, policy gates para reembolsos y human gates para excepciones.

Contrato de Gate

Cada gate debe producir una decisión legible por máquina. No ocultes el comportamiento del gate dentro de prosa.

```
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';
  };
```

La cadena debe registrar la decisión del gate, no solo el output final. Un trace de producción debe mostrar qué paso se ejecutó, qué gate lo aceptó o rechazó y qué ocurrió después.

Usa razones explícitas de falla:

Failure Reason	Significado	Próxima acción
<code>schema_invalid</code>	El output no puede ser parseado o faltan fields requeridos.	Reparar una o dos veces, luego detener.

Failure Reason	Significado	Próxima acción
<code>evidence_missing</code>	El paso hizo un claim sin suficiente soporte de source.	Recuperar más evidence o escalar.
<code>policy_denied</code>	El siguiente paso violaría permisos o policy de negocio.	Detener o pedir aprobación.
<code>ambiguous_input</code>	La cadena no puede elegir una interpretación segura.	Preguntar al usuario o enrutar a revisión.
<code>budget_exhausted</code>	Se agotó el retry, token, latency o tool budget.	Devolver resultado parcial o detener.

Ejemplo de Gate: Borrador de Reembolso

Una cadena de soporte para reembolsos debe detenerse antes de redactar una promesa al cliente si falta evidence requerida. El model puede clasificar, extraer y resumir, pero los gates deciden si el siguiente paso puede ejecutarse.

Chain Step	Model Output	Gate	Condición de pase	Comportamiento ante falla
Classify request	<code>refund_request</code> con confianza <code>0.91</code>	confidence gate	Confianza $\geq$ <code>0.72</code> y el tipo no es <code>unknown</code> .	Hacer una pregunta de aclaración.
Extract fields	<code>order_id</code> , <code>customer_id</code> , <code>issue</code> , <code>requested_outcome</code>	schema gate	Existen fields requeridos y los IDs tienen el formato esperado.	Reparar una vez, luego detener con <code>schema_invalid</code> .
Retrieve policy	source IDs y policy version	evidence gate	La policy de reembolso actual y evidence de la orden están presentes.	Detener con <code>evidence_missing</code> o recuperar de nuevo.
Draft recommendation	reembolso o rechazo propuesto	policy gate	La recomendación coincide con el threshold, estado de cuenta y policy version.	Detener con <code>policy_denied</code> o pedir aprobación.
Draft customer reply	texto para el cliente	final gate	El texto coincide con la recomendación aprobada y no promete pagos no autorizados.	Bloquear y enviar al líder de soporte.

El límite importante es la cuarta fila. Si el model propone “full refund approved” pero el policy gate devuelve `approval_required`, la cadena no debe continuar con una promesa al cliente. Debe pausar con la evidence exacta, monto propuesto, policy version y role del revisor.

```
{
  "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"
}
```

Esta es la diferencia entre una cadena y una secuencia suelta de prompts. La cadena lleva el state tipado y se detiene antes de que lenguaje inseguro o efectos secundarios se escapen.

Notas de Implementación

- Mantén cada prompt acotado. Un paso debe tener un solo objetivo.
- Usa structured output para cada handoff entre pasos.
- Persiste outputs intermedios, no solo la respuesta final.
- Trata cada output del model como no confiable hasta que un gate lo acepte.
- Haz que los repair loops sean acotados. Un schema fallido no debe disparar retries infinitos.
- Prefiere gates deterministas para seguridad y gates basados en model para calidad subjetiva.
- Registra cada paso con hash de input, model, output, resultado del gate, latency y cost.

Ejemplo de 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;
}

```

El ejemplo mantiene el control de flujo en el código. El model clasifica, extrae y redacta, pero no decide qué reglas de validación aplicar.

Modos de Falla

- Un chain aparenta ser determinista mientras prompts ocultos toman decisiones importantes.
- Gates validan la forma pero no el significado.
- El chain pasa el state resumido hacia adelante y pierde evidencia clave.
- Repair loops siguen pidiendo al model que corrija un output sin cambiar el input o el prompt.
- Un evaluador basado en model acepta afirmaciones fluidas pero no fundamentadas.
- El chain se convierte en un sustituto frágil de un verdadero workflow engine.

Lista de Verificación para Producción

- ¿Todos los inputs y outputs de cada paso tienen tipo?
- ¿Cada paso puede probarse con fixture inputs?

- ¿Cada gate tiene un motivo de falla nombrado?
- ¿Cada repair loop tiene un límite de reintentos?
- ¿Los outputs intermedios se persisten para replay?
- ¿Las acciones de alto riesgo están separadas de los pasos de generación?
- ¿Los operadores pueden ver qué gate detuvo una ejecución?

Capítulos Relacionados

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

Routing and Handoffs

El routing envía el trabajo al modelo, prompt, tool, workflow o agent correcto. Los handoffs transfieren la responsabilidad con suficiente state tipado para que el siguiente encargado pueda actuar de forma segura.

Este pattern suele ser el eslabón perdido entre un solo prompt y un multi-agent system.

Intent

Usa un clasificador, regla de policy, planner o condición determinista para elegir el siguiente camino de ejecución. Luego pasa un objeto de state reducido al camino seleccionado.

Un router debe decidir a dónde pertenece el trabajo. No debe completar el trabajo por sí mismo.

Usa cuando

- Los inputs pertenecen a dominios o intenciones distintas.
- Diferentes caminos necesitan diferentes tools, permisos, models o prompts.
- Las solicitudes simples deben usar caminos baratos y rápidos.
- Las solicitudes riesgosas necesitan una policy más estricta o aprobación humana.
- Un agent grande tiene problemas porque tiene demasiados tools o responsabilidades.

Ejemplos comunes:

- enrutar tickets de soporte a workflows de facturación, técnico, cuentas o fraude;
- enrutar solicitudes fáciles a un modelo pequeño y solicitudes inusuales a un modelo más fuerte;
- enrutar tasks de código a agents de búsqueda, edición, revisión o pruebas;
- enrutar solicitudes con efectos secundarios a través de workflows de aprobación;
- enrutar preguntas RAG al índice, tenant o corpus de dominio correcto.

Evita cuando

- Solo hay un camino significativo.
- El router no puede explicar o calificar su decisión.
- Los caminos posteriores tienen propiedad superpuesta.
- Una ruta incorrecta crearía efectos secundarios peligrosos.
- El router depende de context de conversación oculto que no se persiste.

Si la ruta no puede elegirse con confianza, pide aclaración o envía el task a un camino de fallback seguro.

Arquitectura

Request

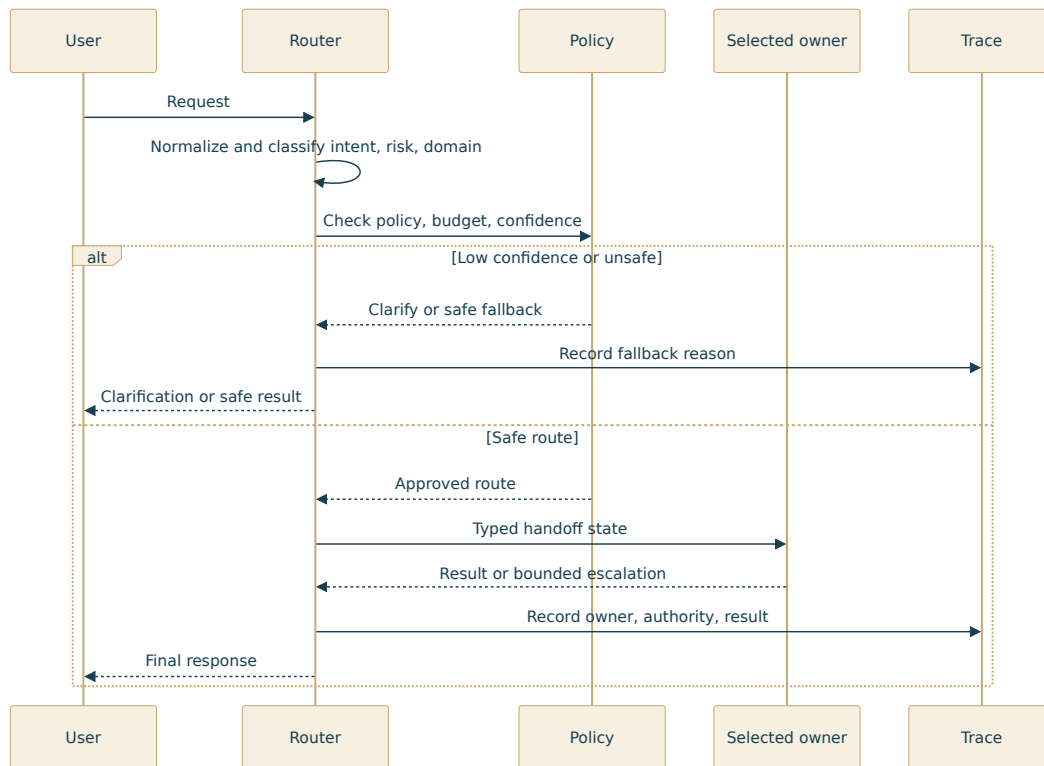
-> Normalize input

-> Classify intent, risk, domain, and required capability

-> Apply policy and budget rules

- > Select path
- > Handoff typed state
- > Downstream path completes or escalates

El handoff debe incluir solo el state que el camino siguiente necesita. Evita volcar toda la conversación en cada especialista.



Usa este diagrama para probar el diseño. Una ruta no está lista si no puede nombrar el owner, autoridad, state del handoff, umbral de confianza, fallback y campos de trace.

Salidas del Router

La salida de un router debe ser estructurada:

```

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';  
}
```

Las buenas decisiones del router son inspeccionables. El sistema debe registrar la ruta, confianza, razón, versión del model, verificaciones de policy y comportamiento de fallback.

Contrato de Handoff

Un handoff no es solo un mensaje. Es un cambio de responsabilidad.

Incluye:

- user intent normalizado;
- decisión de ruta y razón;
- campos extraídos relevantes;
- referencias de evidencia;
- alcance de permisos;
- presupuesto restante;
- deadline o timeout;
- rutas fallidas previas o fallos de tools;
- contrato de output esperado;
- camino de escalamiento.

Excluye:

- historial de conversación no relacionado;
- tools que el destinatario no puede usar;
- texto de policy oculto que debe seguir siendo propiedad del sistema;
- datos sensibles sin procesar cuando una referencia o campo redactado es suficiente.

Registro de Decisión de Ruta

Registra la decisión de routing como datos. Esto hace que el routing sea revisable en vez de anecdótico.

```
{  
  "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
}
```

El registro debe mostrar tres cosas: por qué se eligió la ruta, qué autoridad se transfirió con el handoff y a dónde va el trabajo si la ruta falla.

Tipos de Router

Router Type	Best For	Watch Out For
Rule router	Compliance, tenant, role, known metadata	Rule drift a medida que cambian los productos.
Model router	Natural language intent and ambiguous requests	Baja confianza e inyección de prompt.
Embedding router	Domain corpus or knowledge base selection	Corpora similares pero inseguros.
Cost router	Model tier selection	Uso excesivo de modelos baratos en tasks difíciles.
Risk router	Approval, sandbox, or policy paths	Casos extremos de alto riesgo no cubiertos.
Capability router	Tool or agent selection	Descripciones de tools superpuestas.

Los sistemas en producción suelen combinarlos. Por ejemplo, el código puede hacer cumplir reglas de tenant y policy antes de que un model elija entre workflows especialistas.

Notas de Implementación

- Mantén las etiquetas de ruta estables. Trata los nombres de ruta como contratos de API.
- Haz routing antes de cargar context grande.
- Separa el routing de intent del routing de policy.
- Usa umbrales de confianza con caminos explícitos de aclaración o fallback.
- Valida el state del handoff antes de que el destinatario comience.
- No permitas que los agents posteriores hagan reroute silencioso indefinidamente.
- Traza las decisiones de ruta en toda la ejecución.
- Prueba routers con ejemplos adversariales y ambiguos, no solo casos ideales.

Modos de Falla

- Un router se convierte en un agent general oculto.
- Las etiquetas de ruta se superponen, así que la misma solicitud puede encajar en varios caminos.
- Los handoffs pierden límites de autoridad y dan a los especialistas demasiados tools.
- Rutas de baja confianza avanzan en vez de pedir aclaración.
- Los handoffs multi-agent crean loops sin owner.
- El cost routing envía tasks difíciles a modelos débiles y oculta la pérdida de calidad.

Estrategia de Evaluación

Evalúa la decisión de ruta y el contrato de handoff por separado. Una ruta correcta con autoridad, evidencia o presupuesto faltante sigue siendo un handoff fallido.

- Construye un set de rutas etiquetado con solicitudes claras, ambiguas, fuera de dominio, adversariales y de alto riesgo.
- Prueba cada etiqueta de ruta más los caminos `clarify`, `reject` y fallback seguro.
- Prueba solicitudes que contienen lenguaje asociado a una ruta pero que requieren otra por policy o tenant.
- Asegura que cada handoff incluya su state requerido y excluya context no relacionado y autoridad excesiva.
- Prueba fallas posteriores, rerouting y prevención de loops.
- Compara el routing por model con una base determinista o de camino único antes de agregar más rutas.

Un fixture compacto debe hacer visibles las expectativas de ruta y handoff:

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

Mide la precisión de ruta por clase, tasa de misroute inseguro, precisión de aclaración, tasa de fallback, validez del contrato de handoff, expansión de autoridad, tasa de loops, tasa de finalización posterior, latencia y costo. Reporta una matriz de confusión en vez de depender solo de la precisión general. Un router que funciona bien en solicitudes comunes de soporte aún puede fallar en todos los casos de fraude o seguridad.

Para el contrato de eval compartido y el método de release-gate, consulta [Evaluation-Driven Agent Development](#).

Lista de Verificación para Producción

- ¿Las etiquetas de ruta son mutuamente comprensibles y estables?
- ¿Cada ruta tiene un owner y contrato de output?
- ¿El router tiene un fallback para baja confianza?
- ¿La policy se ejecuta antes de handoffs peligrosos?
- ¿Un operador puede explicar por qué una solicitud tomó un camino?
- ¿Los datasets de rutas son parte de pruebas de regresión?
- ¿Los loops de ruta son imposibles o están acotados?

Capítulos Relacionados

- [Eligiendo el Pattern Correcto](#)
- [Uso de Tool con MCP-first](#)
- [Interoperabilidad de Agent A2A](#)
- [Supervisor / Worker](#)
- [Aplicación de Policy](#)
- [Context Engineering](#)
- [Desarrollo de Agent Basado en Evaluation](#)

Resource-Aware Agent Design

Los agentic systems consumen recursos con cada llamada a un model, retrieval, tool, handoff, operación de memory y retry. El diseño resource-aware hace que el costo, la latencia, el context y el cómputo sean parte de la arquitectura.

Usa este pattern cuando un agent debe ejecutarse repetidamente, atender usuarios de forma interactiva u operar bajo restricciones de presupuesto.

Intent

Optimiza para tasks completadas, no para inteligencia bruta del model.

Un resource-aware agent elige el camino seguro más barato que pueda completar el trabajo con calidad aceptable.

Resource Types

Resource	Qué rastrear
Tokens	prompt tokens, output tokens, hidden context, costo de compresión.
Model calls	cantidad, nivel del model, retries, llamadas en paralelo.
Latency	tiempo de punta a punta, tiempo por paso, tiempo de espera de tool, tiempo de espera humano.
Money	costo por ejecución, costo por task completada, costo por ruta.
Context window	context relevante, context obsoleto, context duplicado.
Tool quota	límites de tasa de API, sesiones de navegador, carga de base de datos.
Human attention	aprobaciones, correcciones, escaladas, tiempo de revisión.
Operational risk	efectos secundarios, verificaciones de policy, costo de incidentes.

La human attention es un recurso. Un camino de model barato que genera trabajo constante de revisión no es barato.

Budget Objects

Dale a cada ejecución un budget object:

```
interface AgentBudget {
  maxCostUsd: number;
  maxModelCalls: number;
```

```

maxToolCalls: number;
maxIterations: number;
maxWallClockMs: number;
maxContextTokens: number;
approvalRequiredAboveUsd?: number;
}

```

Los budgets deben viajar con el state para que routers, tools, evaluators y lógica de fallback puedan tomar decisiones consistentes.

Budget Policy

Un budget solo es útil si el sistema sabe qué hacer a medida que se va gastando.

```

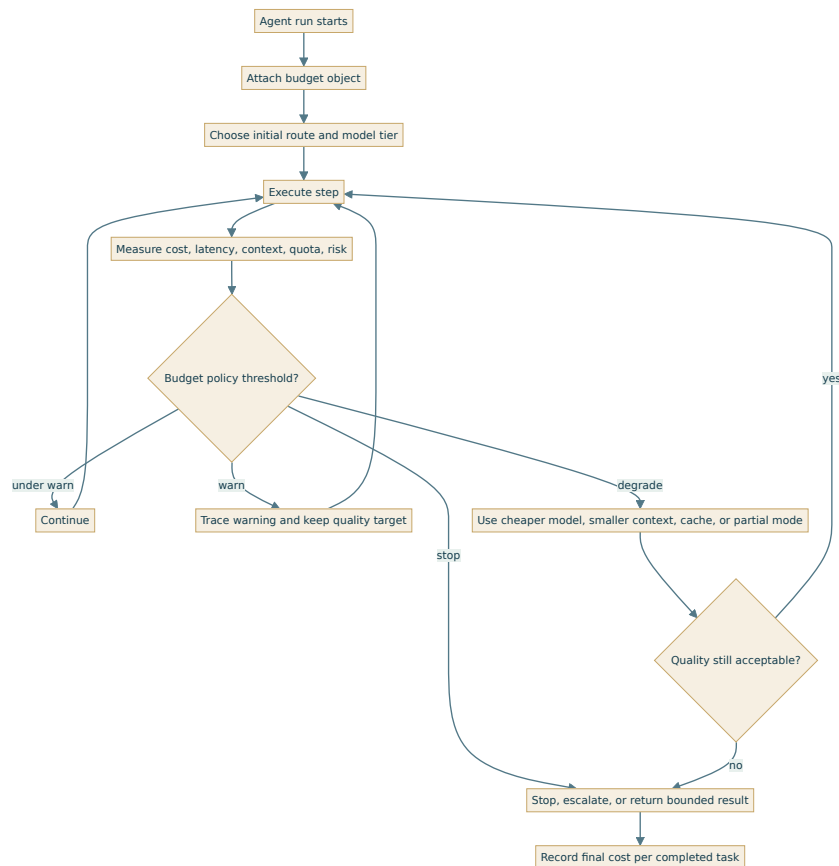
type BudgetAction =
  | 'continue'
  | 'switch_to_cheaper_model'
  | 'compress_context'
  | 'skip_optional_review'
  | 'return_partial_result'
  | 'escalate'
  | 'stop';

type BudgetPolicy = {
  warnAtPct: number;
  degradeAtPct: number;
  stopAtPct: number;
  preferredDegradedAction: BudgetAction;
};

```

Por ejemplo, un asistente de soporte interactivo podría advertir internamente al 60 por ciento del budget, cambiar a clasificación más barata y context más pequeño al 75 por ciento, y detenerse con una respuesta parcial o escalada humana al 95 por ciento. Un research agent en segundo plano podría tolerar mayor latency pero detenerse antes por tool quota o riesgo de frescura de fuentes.

La budget policy debe ser parte del contrato de la ruta o workflow, no un añadido en el logging.



Usa este loop para revisar implementaciones. El control de recursos no es solo un medidor; debe cambiar la ejecución antes de que el costo, la latency, el context, la quota o el risk rompan la ejecución.

Optimization Levers

Lever	Usar cuando	Trade-off
Model routing	Algunas tasks son fáciles y otras difíciles.	Los errores de ruta pueden reducir la calidad.
Prompt chaining	Los pasos son conocidos y pueden validarse.	Más llamadas, pero menos context por llamada.
Context minimization	El context es grande o ruidoso.	Puede omitir evidencia útil.
Retrieval filtering	Las fuentes son numerosas o de calidad mixta.	Los filtros pueden omitir casos límite.
Parallelization	Subtasks independientes dominan la latency.	Más llamadas totales y agregación más difícil.
Caching	Consultas o salidas de tools se repiten.	Riesgos de invalidación y frescura del cache.
Early exit	La confianza es suficientemente alta.	Requiere umbrales calibrados.

Lever	Usar cuando	Trade-off
Human escalation	La automatización sería riesgosa o costosa.	Usa la escasa human attention.

Optimiza solo después de poder rastrear costos por ruta y paso.

Context Compression

Comprime el context cuando sea demasiado grande, repetido u obsoleto.

Prefiere:

- state estructurado sobre historial largo de chat;
- referencias a archivos o registros sobre contenido pegado;
- fragmentos recuperados sobre documentos completos;
- memory específica de task sobre memory global;
- resúmenes con referencias a fuentes;
- eliminación explícita de context irrelevante.

No comprimas evidencia, restricciones, errores de tools, aprobaciones o preguntas sin resolver.

Cost-Aware Routing

El cost-aware routing envía el trabajo por el camino seguro más barato:

- código determinista para operaciones conocidas;
- model pequeño para clasificación y extracción;
- model más fuerte para planeación o síntesis;
- agent especializado para trabajo de dominio;
- revisión humana para ambigüedad de alto riesgo.

El router necesita evals. De lo contrario, los ahorros de costo pueden convertirse silenciosamente en pérdida de calidad.

Latency Design

Los agents interactivos necesitan progreso visible y espera limitada.

Usa:

- estado en streaming en los límites del workflow;
- retrieval en paralelo donde sea seguro;
- trabajos en segundo plano para tareas largas;
- resultados parciales con límites claros;
- cancelación;
- state reanudable.

Para tasks de larga duración, optimiza para reanudabilidad antes que para velocidad bruta.

Degraded Modes

Los sistemas resource-aware deben fallar en menor escala antes de fallar por completo.

Constraint alcanzada	Degraded Mode	No hacer
Token budget	Usa referencias, resúmenes con citas o un conjunto de trabajo más pequeño.	Eliminar restricciones, aprobaciones o historial de errores.
Model cost	Ruta pasos simples a models más baratos.	Rutar síntesis de alto riesgo a un model débil sin prueba de eval.
Latency	Devuelve resultado parcial, manda el resto a segundo plano o pide continuar.	Ocultar trabajo largo detrás de una UI congelada.
Tool quota	Usa datos en cache de solo lectura con etiquetas de frescura.	Llamar tools de escritura sin evidencia actual.
Human attention	Agrupar revisiones de bajo riesgo o eleva el umbral de confianza.	Crear aprobaciones vagas que los humanos firman sin revisar.
Risk budget	Requiere aprobación, sandbox o detenerse.	Permitir que el model decida que el riesgo es aceptable.

El comportamiento degradado debe ser visible en traces y en la salida al usuario cuando afecta la calidad. Un camino más barato o parcial es aceptable; pretender que es el mismo camino no lo es.

Failure Modes

- Optimizar el conteo de tokens mientras aumenta el trabajo de corrección humana.
- Usar un model barato para rutas que no puede clasificar de forma confiable.
- Paralelizar trabajo dependiente y crear resultados inconsistentes.
- Comprimir el context tan agresivamente que el agent olvida restricciones.
- Cachear evidencia obsoleta en dominios regulados o de rápido cambio.
- Medir el costo por llamada en vez del costo por task completada.

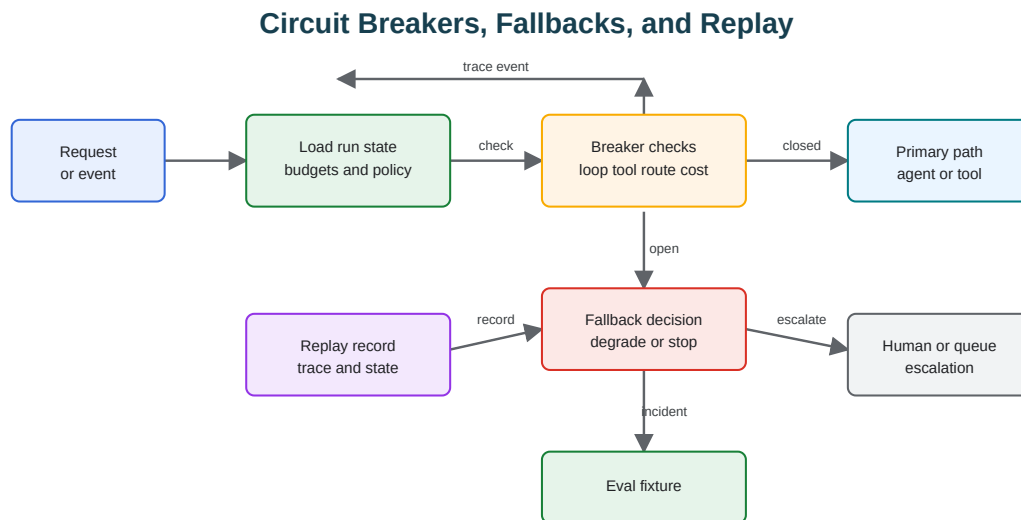
Related Chapters

- [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

Los agentic systems necesitan controles que detengan el desperdicio, contengan daños y hagan que las fallas sean explicables. Los circuit breakers detienen ejecuciones inseguras o improductivas. Los fallbacks le dan al sistema un siguiente paso más seguro. Replay permite a los ingenieros reconstruir lo que sucedió.

Este es un reliability pattern para agent loops, uso de tools, RAG, orquestación de workflow y multi-agent systems.



Intent

Proteger el sistema de fallas repetidas y hacer que cada ejecución sea lo suficientemente recuperable para depurar.

Un agent no debe seguir llamando al mismo tool que falla, buscando en el mismo corpus vacío, revisando el mismo draft defectuoso o transfiriendo el mismo task entre agents sin progreso.

La idea clave es simple: la autonomía necesita frenos en runtime. Un breaker no es un manejador de errores oculto dentro de un tool wrapper. Es un control arquitectónico que decide cuándo el sistema debe detenerse, degradarse, pedir ayuda o preservar suficiente state para replay.

Use When

- Los agents pueden llamar tools, APIs, browsers, shells o workflows.
- El trabajo puede hacer loop, reintentar, delegar o esperar.
- Los costos pueden crecer con cada llamada a un model o tool.
- El state parcial importa después de una falla.
- Los operadores necesitan explicar por qué se detuvo una ejecución.

Este pattern es obligatorio para sistemas en producción con efectos secundarios.

Avoid When

- El task es un prototipo desechable sin efecto externo.
- El sistema no tiene loop, retry ni uso de tools.
- Las fallas no necesitan investigación.

Aun así, mantén un simple run log. El prototipo que funciona a menudo se convierte en la semilla de producción.

Circuit Breakers

Un circuit breaker convierte una falla repetida o de alto riesgo en una detención, fallback o escalamiento.

Breaker	Trigger	Action
Tool failure breaker	El mismo tool falla N veces o retorna output malformado.	Deshabilitar tool para la ejecución y elegir fallback.
No-progress breaker	El state no cambia entre iteraciones.	Detener el loop y devolver estado bloqueado.
Cost breaker	Se alcanza el presupuesto de tokens, model-call o dinero.	Detener, resumir el progreso y pedir aprobación.
Latency breaker	Un paso o ejecución excede el presupuesto de tiempo.	Diferir, encolar o devolver resultado parcial.
Retrieval breaker	La búsqueda retorna evidencia de baja cobertura o conflictiva.	Pedir aclaración o escalar.
Policy breaker	La intención del tool viola permisos o reglas de riesgo.	Bloquear acción y registrar motivo de policy.
Handoff breaker	Los agents rebotan un task entre roles.	Asignar propietario o escalar a humano.
Output breaker	La respuesta final falla checks de schema, citation o eval.	Reparar una vez, luego fallar de forma segura.

Los breakers necesitan nombres y umbrales. “El agent se atascó” no es suficientemente operativo. “No-progress breaker se activó después de 4 iteraciones con el mismo evidence set” es depurable.

Coloca breakers en el límite donde el daño puede crecer:

Boundary	What The Breaker Protects
Loop boundary	Previene razonamiento sin límites, planeación repetida e iteración sin progreso.
Tool boundary	Previene fallas externas repetidas, argumentos malformados, efectos secundarios duplicados y escrituras inseguras.

Boundary	What The Breaker Protects
Model boundary	Previene costos descontrolados, picos de latencia, rutas de model defectuosas y comportamiento degradado de structured-output.
Retrieval boundary	Previene respuestas basadas en evidencia débil, obsoleta, contradictoria o no autorizada.
Memory boundary	Previene que memory privada, obsoleta o de baja confianza dirija futuras ejecuciones.
Policy boundary	Previene que el agent convierta intención en acción prohibida.
Multi-agent boundary	Previene loops de delegación, propiedad poco clara y desacuerdos sin resolución.

El breaker debe emitir un trace event con el nombre del breaker, trigger, umbral, valor observado, acción tomada y motivo de detención visible para el usuario. Si el breaker se activa pero el trace no explica por qué, los operadores seguirán depurando un misterio.

Fallbacks

Un fallback debe ser más seguro que el camino fallido.

Tipos útiles de fallback:

- pedir al usuario información faltante;
- devolver una respuesta parcial con límites explícitos;
- cambiar de uso autónomo de tools a workflow determinista;
- cambiar de un model débil a uno más fuerte;
- cambiar de acción de escritura a análisis solo lectura;
- usar datos en caché o previamente verificados;
- enrutar a revisión humana;
- programar procesamiento en segundo plano;
- fallar cerrado para acciones restringidas.

No uses fallback para ocultar la falla. El usuario u operador debe saber qué cambió.

Un buen fallback tiene un contrato:

- **Alcance:** qué puede hacer el fallback.
- **Límite de calidad:** qué no puede garantizar el fallback.
- **Regla de state:** si la ejecución puede continuar, debe pausar o debe detenerse.
- **Mensaje al usuario:** cómo se explica la limitación sin exponer ruido interno.
- **Operator trace:** qué breaker causó el fallback y qué evidencia se preservó.

Para agentic systems, el fallback más seguro suele reducir la autonomía. Cambia de escritura a draft, de acción autónoma a aprobación, de uso amplio de tools a workflow determinista, o de respuesta final a aclaración.

Replay

Replay es la capacidad de reconstruir una ejecución a partir de state durable.

Almacena:

- run ID y parent run ID;
- goal e intención normalizada;
- llamadas a model, versiones de model, parámetros y referencias de prompt;
- llamadas a tool, inputs, outputs, errores e identificadores de efectos secundarios;
- consultas de retrieval, source IDs y metadatos de citation;
- transiciones de state;
- decisiones de ruta y handoffs;
- policy checks;
- eventos de breaker y fallback;
- output final y resultados de eval.

Redacta contenido sensible cuando sea necesario, pero conserva suficientes referencias para investigar la ejecución.

Replay debe diseñarse antes del incidente. Después de una falla, es demasiado tarde para descubrir que prompts, argumentos de tools, decisiones de policy y transiciones de state estaban dispersos en logs no relacionados.

Replay también necesita reglas de seguridad. Un replay no debe enviar accidentalmente otro correo a un cliente, emitir otro reembolso o escribir otro registro en memory. Separa replay de trace solo lectura, replay determinista con outputs simulados y replay en sandbox con efectos secundarios falsos.

Replayable Actions

Cuando un agent realiza trabajo costoso o riesgoso, haz que la acción sea replayable:

- registra el input, tool seleccionado, decisión de policy y resultado;
- adjunta una idempotency key a efectos secundarios externos;
- almacena en caché resultados seguros de solo lectura cuando las reglas de frescura lo permitan;
- preserva suficiente context para reproducir una ejecución fallida;
- haz explícito el rollback para operaciones que cambian el state.

Replay convierte una falla de un misterio en un caso de prueba.

Replay Packet

Cuando un breaker se activa, almacena un replay packet con el conjunto mínimo útil de registros necesarios para reconstruir la falla.

```
{  
  "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"
}

```

El packet debe responder cuatro preguntas: qué se activó, qué creía el sistema, qué pasó antes del breaker y cómo hacer replay de forma segura sin repetir efectos secundarios.

Replay Levels

Level	Capability	Use
Trace replay	Reconstruir lo que sucedió a partir de logs.	Revisión de incidentes y depuración.
Deterministic replay	Re-ejecutar código determinista con outputs de model/tool grabados.	Pruebas de regresión y validación de workflow.
Full replay	Re-ejecutar llamadas a model y tool en un sandbox.	Reproducción cuando sistemas externos son estables.
Counterfactual replay	Re-ejecutar con prompt, policy, model o tool cambiado.	Evaluar correcciones antes de desplegar.

La mayoría de los equipos debe comenzar con trace replay. Full replay es útil pero más difícil porque models, tools y datos externos cambian.

Counterfactual replay es especialmente útil para agentic systems. Permite al equipo preguntar: ¿el nuevo prompt, ruta de model, regla de policy, retrieval index o tool schema habría evitado la falla sin romper casos conocidos buenos?

Notas de implementación

- Agrega breakers en los niveles de loop, tool, retrieval, route y workflow.
- Registra los eventos de breaker como eventos de trace de primer nivel.
- Vincula las decisiones de fallback a razones explícitas de falla.
- Separa los retries seguros de los intentos repetidos sin fundamento.

- Haz que cada efecto secundario sea idempotente o trazable.
- Almacena suficiente state para reanudar o fallar limpiamente.
- Convierte los incidentes en casos de eval.
- Prefiere eventos de breaker tipados sobre mensajes de log libres.
- Mantén los umbrales de breaker configurables por ambiente, clase de riesgo y capability de tool.
- Trata el disparo de un breaker como una señal de evaluación, no solo como un evento operativo.

Ejemplo de lógica de breaker

```
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;  
}
```

Lo importante no es el tamaño del código. Lo importante es que la razón de detención se convierte en parte del run state y del trace.

La acción debe guiar el comportamiento. `continue` sigue el camino principal. `fallback` pasa a un camino más seguro. `escalate` solicita que un humano o una cola tomen el control. `stop` termina el run con un state claro y un registro de replay.

Modos de falla

- Los breakers existen en logs pero no afectan la ejecución.
- Los fallbacks bajan la calidad en silencio sin informar al usuario.
- Los retries repiten los mismos inputs y esperan resultados diferentes.
- Los traces omiten los inputs de tool, haciendo imposible el replay.
- Los efectos secundarios no pueden asociarse al agent run que los creó.
- Los sistemas multi-agent no tienen un único owner cuando se dispara un breaker.
- Los umbrales de breaker están codificados y no pueden cambiar según la clase de riesgo.
- El replay solo puede reproducir la falla tocando sistemas en vivo nuevamente.
- El sistema recurre a un modelo más potente cuando el problema real es la falta de policy o state.

Guía de evaluación

Prueba breakers y fallbacks directamente. No esperes a incidentes en producción para demostrar que el runtime puede detenerse de forma segura.

Caso de Eval	Comportamiento esperado
Tool falla repetidamente	El breaker de tool se activa, se ejecuta el camino de fallback, el trace registra el umbral.
Loop no avanza en el state	El loop se detiene con una razón de no-progreso y sin llamadas extra a tools.
Se agota el presupuesto de costo	El run se pausa, resume el progreso y pide aprobación o presupuesto de continuación.
La evidencia de retrieval es débil	El agent pide aclaración o rechaza afirmaciones no soportadas.
Policy bloquea una acción de escritura	La acción es denegada o escalada antes de los efectos secundarios.
Delegación multi-agent se repite	El supervisor asigna un único owner o escala el caso.
Replay usa el trace grabado	El replay no dispara efectos secundarios en vivo.

Lista de verificación para producción

- ¿Cada loop tiene máximo de iteraciones, costo máximo y tiempo máximo de ejecución?
- ¿Cada tool tiene timeout, retry y umbrales de falla?
- ¿Cada fallback informa al operador qué cambió?
- ¿El sistema puede detenerse de forma segura después de un progreso parcial?
- ¿Un run fallido puede convertirse en un fixture de eval?
- ¿Los efectos secundarios pueden auditarse y revertirse cuando sea posible?
- ¿Los operadores pueden reproducir un run sin leer prompts crudos de logs dispersos?
- ¿Los eventos de breaker están vinculados a IDs de trace, casos de eval y revisiones de incidentes?
- ¿Los caminos de fallback se prueban antes del lanzamiento?
- ¿Se pueden deshabilitar tools riesgosos sin redeplegar todo el agent?

Capítulos relacionados

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

Source Map

Esta página mapea referencias externas a los capítulos de este libro. Úsala como una guía de lectura, no como un reemplazo del pattern language del libro.

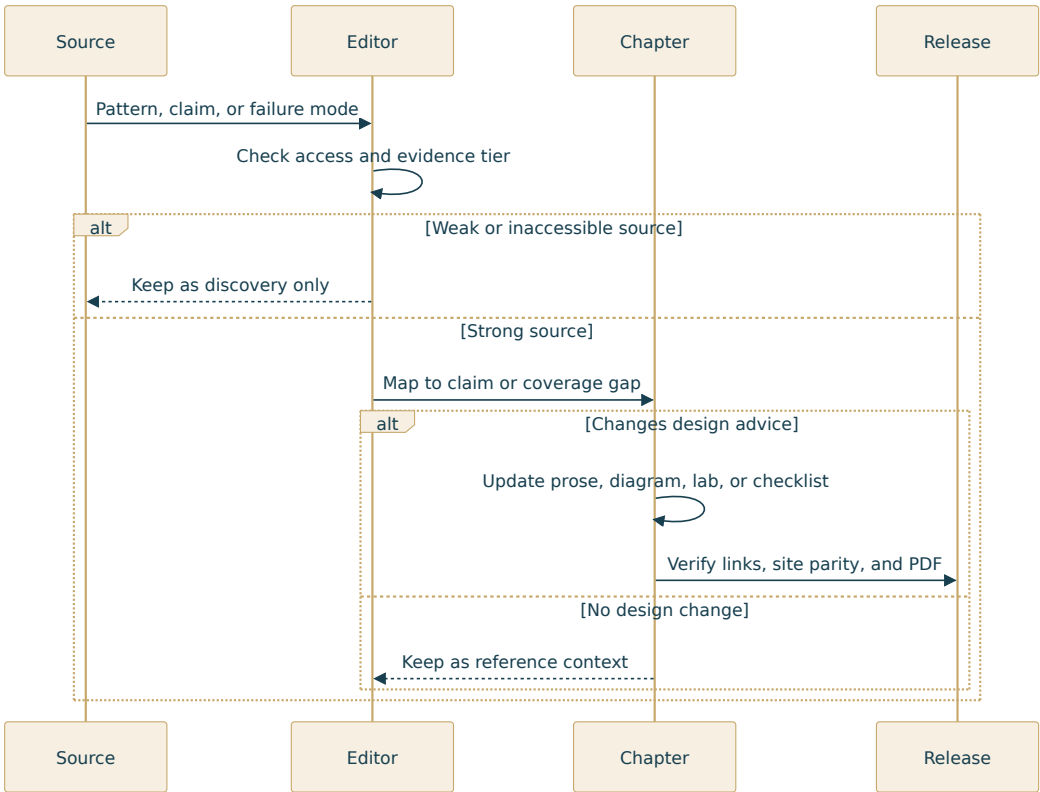
Las fuentes repiten varias ideas centrales: empieza simple, separa workflows de agents, usa tools mediante contratos tipados, agrega memory de forma deliberada, evalúa el comportamiento y reserva los sistemas multi-agent para casos donde la especialización justifique el costo de orquestación.

Cómo usar este mapa

Utiliza esta página cuando necesites responder una de tres preguntas:

Pregunta	Usa esta sección	Resultado
¿De dónde proviene el pattern language de este libro?	Primary References and Local PDF References Reviewed.	Un breve rastro de fuentes para un capítulo, decisión o término.
¿El libro cubre un pattern que vi en otro lugar?	Pattern Coverage Map and Local Pattern Repository Coverage.	Un enlace al capítulo, o una posible brecha para futura expansión.
¿A qué fuentes debo dar prioridad?	Evidence Tiers.	Un orden de prioridad para leer, citar y revisar.

Para la lectura normal, no empieces aquí. Comienza con [How To Read This Book](#), luego regresa a este mapa cuando quieras contexto de fuentes o verificar cobertura.



Lee el mapa como un filtro editorial. Una fuente gana espacio en el libro cuando cambia la decisión de diseño de un lector, expone un modo de falla no considerado o aporta evidencia más sólida para una afirmación del capítulo.

Atajos de decisión respaldados por fuentes

Usa estos atajos cuando una fuente nombra un pattern pero la decisión de ingeniería aún no es clara.

Decisión que necesitas tomar	Comienza con	Usa las fuentes para verificar
¿Es esto un workflow, un agent o un sistema multi-agent?	Choosing the Right Pattern	Si la fuente separa el control determinista de los pasos elegidos por el model.
¿Debemos agregar routing, handoffs o agents especialistas?	Routing and Handoffs y Choosing Multi-Agent Topology	Si contextos, tools, permisos o latencia separados justifican el costo de coordinación.
¿Esto necesita memory, retrieval o un límite de conocimiento?	Context Engineering, Working Memory y Agentic RAG Systems	Si la fuente distingue entre contextos transitorios, memory durable y evidencia citada.
¿Son los tools lo suficientemente seguros para uso mediado por el model?	Tool Capability Design, MCP-first Tool Use y Human Approval Gates	Si la autoridad del tool, schema, errores, policy y aprobación son explícitos.
¿El pattern puede sobrevivir a fallas en producción?	Circuit Breakers, Fallbacks, and Replay, Observability and Evals y Production Evaluation Feedback Loops	Si traces, replay, evals, rollback y retroalimentación de incidentes forman parte del diseño.
¿Una fuente está agregando un pattern útil o solo un nombre nuevo?	Pattern Composition Playbook	Si el pattern cambia ownership, state, tools, riesgo, evals u operaciones.

Evidence Tiers

No todas las fuentes tienen el mismo peso editorial. Usa este orden cuando las fuentes discrepen:

Tier	Tipo de fuente	Uso editorial
1	Guías de ingeniería de vendors, documentación de frameworks y documentación principal de implementación.	Usar para terminología actual, guía operativa y comportamiento específico de frameworks.
2	Libros técnicos y material extenso de practicantes.	Usar para conceptos duraderos, orden de enseñanza y ejemplos que requieren explicación más profunda.

Tier	Tipo de fuente	Uso editorial
3	Catálogos curados, listas de patterns y diagramas.	Usar para descubrir cobertura y enlaces cruzados, no como autoridad final.
4	Videos, artículos restringidos y publicaciones con poca revisión.	Usar solo como material de descubrimiento hasta que las afirmaciones se verifiquen con fuentes más sólidas.

Esto previene que el libro en línea se desvíe hacia la fuente con la lista de patterns más larga. El libro debe privilegiar la evidencia de ingeniería sobre la novedad.

Referencias Primarias

Fuente	Ideas Útiles	Mapeo en el Libro
Anthropic: Building Effective Agents	Workflows vs agents, prompt chaining, routing, paralelización, orquestador-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	Selección basada en requerimientos, patrones single-agent y multi-agent, secuencial, paralelo, refinamiento iterativo, human-in-the-loop, lógica personalizada.	Choosing the Right Pattern , Parallel Agents , Human Approval Gates , Agentic System Architecture .
Databricks: Agent system design patterns	Continuum de complejidad desde prompt hasta deterministic chain, single-agent y multi-agent systems, además de guías de producción para testing, tracing, manejo de fallos, actualizaciones de model y costo.	Choosing the Right Pattern , Circuit Breakers , Fallbacks , and Replay , Observability and Evals .
LangChain: Choosing the Right Multi-Agent Architecture	Selección multi-agent entre subagents, skills, handoffs y routers, con tradeoffs explícitos sobre aislamiento de context, state, paralelismo y overhead de llamadas a model.	Choosing the Right Pattern , Routing and Handoffs , Skills , Supervisor / Worker , Parallel Agents .
LangChain: Deep Agents overview y Frameworks, runtimes, and harnesses	Enfoque de agent harness: planeación, archivos, manejo de context, subagents, memory a largo plazo, aprobación humana, permisos, sandboxes, ejecución durable y distinciones entre framework/runtime/harness.	Agent Harnesses , Context Budgets and Working Sets , Agent Engineer Toolkit , Framework Selection , Skills , Durable Workflows .
MongoDB: 7 Practical Design Patterns for Agentic Systems	Flujos controlados, LLM routing, paralelización, reflection, human-in-the-loop, agents y arquitecturas multi-agent.	Prompt Chaining and Gates , Routing and Handoffs , Reflection , Supervisor / Worker .

Fuente	Ideas Útiles	Mapeo en el Libro
Agentic Patterns Graph	Gran catálogo de patterns con categorías para context, orquestación, confiabilidad, seguridad, uso de tool, aprendizaje y UX.	Circuit Breakers, Fallbacks, and Replay , Context Engineering , Working Memory , Policy Enforcement .
GitHub: awesome-agentic-patterns	Repositorio curado de patterns de producción y emergentes, útil como índice de descubrimiento.	Esta fuente informa el catálogo extendido y futuras adiciones, especialmente patterns de confiabilidad, context, coding-agent, seguridad y uso de tool.

Referencias Secundarias

Fuente	Ideas Útiles	Cómo Usar
ByteByteGo: Top AI Agentic Workflow Patterns	Explicaciones accesibles de reflection, uso de tool, ReAct, planeación y patterns multi-agent.	Buena lectura introductoria para quienes buscan una explicación más ligera antes de los capítulos profundos.
Phil Schmid: Zero to One - Learning Agentic Patterns	Ejemplos prácticos de routing, paralelización, uso de tool, orquestador-workers y sistemas multi-agent.	Útil para intuición de implementación y ejemplos de formas.
ProjectPro: Agentic AI Design Patterns	Planeación, uso de tool, reflection, ejemplos multi-agent y discusión práctica de MCP.	Útil para explicar ejemplos comunes en empresas.
Tungsten Automation: Tool-Use Pattern	Enfoque empresarial para uso de tool, ground truth y APIs de workflow.	Se relaciona con MCP-first Tool Use y Policy Enforcement .
Towards AI: 5 Design Patterns in Agentic AI Workflow	Enfoque introductorio para prompt chaining y descomposición de workflow.	Útil como visión general ligera; el artículo está parcialmente restringido.
Medium: Multi-Agent System Patterns	Dimensiones de arquitectura multi-agent: control, ejecución, coordinación e interacción. Útil distinción entre roles y patterns.	Buen soporte para un futuro capítulo puente sobre composición de sistemas multi-agent sin convertir el libro en un catálogo plano.
Medium: Agentic Design Patterns	Explicaciones comunes de patterns para reflection, uso de tool, planeación y diseño multi-agent.	Duplica conceptos centrales ya cubiertos aquí.
Medium: Agentic Patterns - Architectures	Coordinación jerárquica, peer-to-peer, basada en mercado y swarm.	Útil para futura expansión de patterns de coordinación multi-agent.

Fuente	Ideas Útiles	Cómo Usar
for Coordinated AI Systems		
YouTube: AI agent design patterns	Recorrido en video de agent design patterns.	Usar como material complementario, no como fuente textual principal a menos que se agregue revisión de transcripción.
YouTube: Master ALL 20 Agentic AI Design Patterns	Amplio catálogo en video de nombres de patterns y ejemplos.	Usar como material de descubrimiento; validar afirmaciones individuales con fuentes primarias.

Las fuentes que no eran de acceso público o solo sirvieron como rutas de descubrimiento fueron excluidas del mapa.

Cobertura del Repositorio Local de Patterns

Se utilizó un resumen plano local del proyecto `awesome-agentic-patterns` como lista de verificación de cobertura. Sus patterns no se copiaron en este libro. Ayudaron a identificar áreas faltantes o poco desarrolladas que ahora tienen tratamiento propio aquí.

El escaneo de ingreso del 2026-06-20 extrajo 167 registros de patterns en las áreas de orquestación y control, uso de tools y entorno, confiabilidad y evaluación, context y memory, UX y colaboración, seguridad y safety, feedback loops, y aprendizaje y adaptación. El libro en línea debe usar este catálogo como una cola de expansión para la calidad de los capítulos y enlaces cruzados, no como una taxonomía de reemplazo.

Área de Pattern Local	Patterns Representativos Revisados	Mapeo en el Libro
Agent threat model y 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 y 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</code>	Production Evaluation Feedback Loops , Evaluation-

Área de Pattern Local	Patterns Representativos Revisados	Mapeo en el Libro
	<code>rollback-for-agent-policy-changes</code> , <code>anti-reward-hacking-grader-design</code> , <code>background-agent-ci</code> .	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 y coordinación	<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.

Esta fuente local debe tratarse como una lista de verificación, no como la taxonomía del libro. Algunas entradas están probadas en producción, otras están emergiendo y algunas son específicas de investigación o producto. El libro debe seguir absorbiendo las ideas útiles de ingeniería mientras preserva su propia estructura: primero los límites, luego los patterns, después la evaluación y operaciones.

Referencias locales de PDF revisadas

Los siguientes PDFs fueron revisados o escaneados localmente como insumos de conocimiento fuente. No están incluidos en este repositorio, enlazados desde el libro, ni reproducidos. Solo informan verificaciones de cobertura, estructura de capítulos y priorización de temas.

Referencia local	Temas útiles	Mapeo en el libro
<i>AI Agents in Action</i> , Micheal Lanham	Definiciones de agent, interfaces LLM, asistentes GPT, sistemas multi-agent con AutoGen y CrewAI, acciones de agent, orquestación con behavior-tree, plataformas de agent, RAG, memory, flujo de prompt, razonamiento, evaluación, planeación y retroalimentación.	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	Fundamentos de ingeniería de agent, decisiones de toolkit, domain agents, retrieval agents, orquestación de tool, software agents, explicabilidad y agents específicos de dominio.	Agent Engineer Toolkit, Framework Selection, Domain Agent Architectures, Coding Agents, Agentic RAG Systems.
<i>Agentic Architectural Patterns for Building Multi-Agent Systems</i> , Ali	Madurez empresarial, selección de model agent-ready, espectro de RAG a fine-tuning, patrones de coordinación, cumplimiento,	Choosing the Right Pattern, Agent Development Lifecycle, Agent Security and Sandboxing,

Referencia local	Temas útiles	Mapeo en el libro
Arsanjani, Juan Pablo Bustos, Thomas Kurian	robustez, interacción humano-agent, preparación para producción.	Domain Agent Architectures, Agentic System Architecture.
<i>Agentic Design Patterns</i> , Antonio Gulli	Prompt chaining, routing, paralelización, reflexión, uso de tool, planeación, memory, MCP, A2A, monitoreo, guardrails, optimización de recursos, CLI y 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	Taxonomía multi-agent, principios de UX, execution loops, cancelación, memory, middleware, computer-use agents, workflow graphs, observability, evaluación, protocolos distribuidos, ética.	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	Diseño de capability de agent, context engineering, compresión de context, eval suites, evaluación con datos de producción, sandboxing, control de acceso granular, guardrails.	Context Budgets and Working Sets, Context Engineering, Resource-Aware Agent Design, Evaluation-Driven Agent Development, Agent Security and Sandboxing.
<i>Build a Multi-Agent System (from Scratch)</i> , Val Andrei Fajardo	Construcción fundamental de agent LLM, tools, interfaces LLM, tools MCP, skills, memory, human-in-the-loop, A2A y creación de un pequeño framework educativo desde primeros principios.	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	Diseño de interfaces LLM, uso de tool, ReAct, RAG, memory, planeación, reflexión, ejecución de código, orquestación multi-agent y evaluación de agent.	Single Agent, ReAct, Semantic Recall and RAG, Planning and Execution, Evaluation-Driven Agent Development.
<i>Designing AI Agents</i> , Jia Huang	Harnesses de agent, asignación de recursos limitada, funciones cognitivas, topología de ejecución, gobernanza y ejemplo en ejecución de code-review-agent.	Agentic System Architecture, Resource-Aware Agent Design, Coding Agents, Policy Enforcement, Architecture Decision Records.
<i>Multi-Agent Systems with AutoGen</i> , Victor Dibia	Fundamentos multi-agent con AutoGen, UX, interface agents, evaluación, optimización, despliegue, protocolos de mensajería, seguridad y 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	LangChain, LangGraph, MCP, RAG, agents basados en tool, sistemas multi-agent,	Framework Selection, Agentic RAG Systems, MCP-first Tool

Referencia local	Temas útiles	Mapeo en el libro
Infante	memory, guardrails y puesta en producción.	Use, Multi-Agent Systems, Observability and Evals.
<i>Agentic Transformation Playbook</i>	Enfoque de adopción empresarial, gestión del ciclo de vida de agent, gobernanza, diseño de roles humanos y casos de uso comunes en empresas.	Agent Development Lifecycle, Agent UX and Human Trust, Policy Enforcement, Domain Agent Architectures.

Mapa de Cobertura de Patterns

Nombre del Pattern Externo	Capítulo Actual del Libro
Augmented LLM	Single Agent, Tool Use, Structured Output
Prompt chaining	Prompt Chaining and Gates
Routing	Routing and Handoffs
Parallelization	Parallel Agents
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

Nombre del Pattern Externo	Capítulo Actual del Libro
Domain-specific agent	Domain Agent Architectures

Lista de Verificación de Mantenimiento

Actualiza esta página cuando una fuente cambie la estructura del libro, no cada vez que aparezca un nuevo enlace.

Antes de agregar una fuente, verifica:

- ¿Introduce un pattern, failure mode, runtime concern o técnica de implementación que aún no esté cubierto?
- ¿Es más sólida que una fuente existente para la misma afirmación?
- ¿Se asigna al menos a un capítulo donde el lector pueda aplicar la idea?
- ¿Es lo suficientemente pública para que un lector en línea de GitHub Pages la pueda inspeccionar?
- ¿Crea una tarea editorial concreta, como un ejemplo faltante, checklist, diagrama, laboratorio o cross-link?

Si la respuesta es mayormente no, deja la fuente fuera. Un mapa más corto con propósito editorial claro es más útil que una gran pila de referencias.

Regla Editorial

Las fuentes externas se usan para validar la cobertura y exponer patterns faltantes. El libro mantiene su propia taxonomía:

- fundamentos primero;
- control loops en segundo lugar;
- memory y knowledge como una preocupación separada;
- tools, skills y protocols como límites de integración;
- multi-agent systems solo cuando la especialización aporta valor;
- production runtime patterns para seguridad, evaluación y operaciones;
- selección de patterns informada por fuentes como punto de entrada para elegir el diseño correcto.

Esto mantiene el libro útil como referencia en lugar de convertirlo en un simple listado de enlaces.

Single Agent

Un single agent recibe un goal o mensaje, consulta su context y produce una respuesta o acción. Esta es la unidad útil más pequeña en el catálogo.

Fuente y descargas

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

Propósito

Un single agent recibe un goal o mensaje, consulta su context y produce una respuesta o acción. Esta es la unidad útil más pequeña en el catálogo.

Escenario

Un equipo de soporte quiere un pequeño asistente que reescriba notas internas en bruto en una respuesta orientada al cliente. La entrada contiene el resumen del ticket, un extracto de la policy y el tono deseado. La salida es un borrador de respuesta sin llamadas a tools, sin escrituras en memory y sin autoridad de pago.

Este es un buen baseline de single-agent porque la task tiene un solo worker, un paquete de context acotado y una salida tipada. El solicitante sigue siendo dueño del state del ticket, la versión de la policy, la escalación y el envío final. El agent solo es dueño del borrador.

```
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
```

Si el equipo después necesita consulta de órdenes en vivo, recuperación de policy, aprobación o envío de mensajes, este pattern ha llegado a su límite. Mantén el single agent como el worker de redacción y coloca tools, state, aprobación y entrega en el workflow que lo rodea.

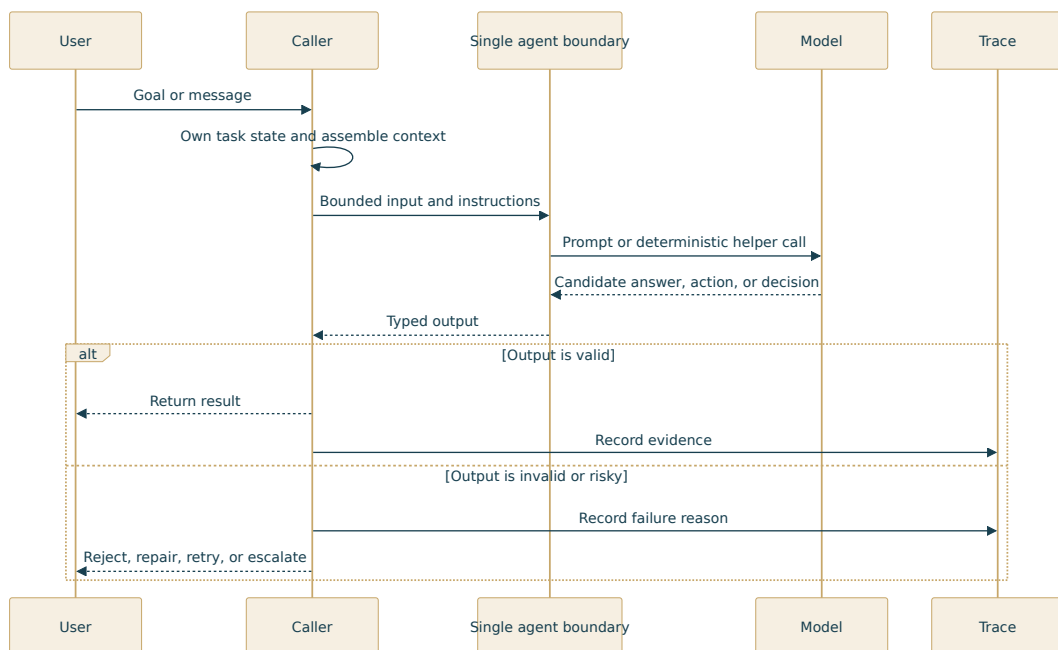
Usar cuando

- Un solo worker respaldado por un model puede completar la task sin delegar.
- La interacción tiene un objetivo estrecho y una condición de éxito clara.
- Quieres el baseline más simple y útil antes de agregar tools, memory u orquestación.

Evitar cuando

- La task necesita reintentos con state, aprobaciones externas, múltiples especialistas o evaluación independiente.
- El agent debe recuperarse de fallas prolongadas o reanudar después de una interrupción.

Arquitectura



Usa esto como la arquitectura baseline. Si el sistema necesita reintentos durables, evaluación independiente, orquestación de tools o handoffs de especialistas, ya superó el patrón de single-agent.

Forma del sistema

- **Límite del pattern:** una función, clase o servicio de agent con límite estrecho acepta input más context y retorna una respuesta, acción o decisión tipada.
- **Dueño del state:** el solicitante o un pequeño servicio de aplicación es dueño del task state hasta que se introduce un runtime pattern.
- **Artifact principal:** `single-agent-pattern/` contiene la implementación de referencia ejecutable y ejemplos.

- **Promesa operacional:** Un single agent recibe un goal o mensaje, consulta su context y produce una respuesta o acción. Esta es la unidad útil más pequeña en el catálogo.
- **Ruta ejecutable:** comienza con `npm run single-agent` antes de adaptar el pattern a un sistema mayor.

Protocolo central

1. Acepta un input, goal o solicitud de task acotada.
2. Ensambla las instrucciones mínimas útiles, context, state y descripciones de tools.
3. Ejecuta el model o helper determinista detrás de un límite tipado.
4. Valida el resultado antes de devolverlo a usuarios, tools o state durable.
5. Registra suficiente evidencia para explicar la salida después.

Notas de implementación

- Mantén explícito el límite del pattern: inputs, state, efectos secundarios y outputs deben ser visibles.
- Valida las decisiones producidas por el model antes de que afecten tools, usuarios o state durable.
- Emite suficiente trace data para depurar fallas después de la ejecución.

Modos de falla

- El pattern se aplica donde un workflow determinista más simple sería mejor.
- State, llamadas a tools o decisiones del model no son lo suficientemente observables para depurar.
- El sistema carece de comportamiento claro de detención, reintento o escalación.

Estrategia de evaluación

- Usa golden tasks que cubran solicitudes normales, solicitudes ambiguas, context faltante y input inválido.
- Verifica que los outputs coincidan con la forma esperada y que solicitudes inseguras o no soportadas sean rechazadas.
- Rastrea precisión, validez de schema, latencia, uso de tokens y calidad de rechazos.
- Incluye casos que prueben que cada condición de “Usar cuando” es verdadera para este pattern.
- Incluye casos negativos de “Evitar cuando” para que el sistema elija un pattern más simple o seguro cuando corresponda.

Lista de verificación para producción

- Define el input, context, output y contrato de errores.
- Mantén prompts, schemas y descripciones de tools versionados.
- Agrega pruebas deterministas para el comportamiento útil más pequeño.
- Registra decisiones del model sin filtrar secretos ni datos privados de usuarios.
- Define escalación humana para trabajo ambiguo, de alto riesgo o bloqueado por policy.
- Mantén el source bundle, capítulo generado, pruebas y artifact de despliegue en la misma release.

Ejecuta el ejemplo

```
npm run single-agent
```

Recorrido del código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el 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

Este ejemplo demuestra el Single Agent Pattern usando LangGraph y Python. El agent recibe un mensaje de usuario, lo envía a un Mistral LLM y retorna la respuesta.

Requisitos

- Python 3.8+
- Biblioteca `langgraph`
- `python-dotenv` (para soporte de .env)
- Acceso a la API de Mistral LLM

```

## Instalar dependencias

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

## Código de ejemplo

```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)
```

```

```
- Asegúrate de que tu archivo `.env` contenga tu clave de API de Mistral como  
`MISTRAL_API_KEY`.  
- Este es un ejemplo funcional y mínimo.
```

Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `single-agent-pattern/` actual de este repositorio.

Patrones relacionados

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

Agent Loop

El agent loop convierte una llamada de model en un agent: observa el state, decide la siguiente acción, actúa, evalúa el resultado y se detiene cuando el goal se completa o se alcanza un límite.

Fuente y descargas

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

Propósito

El agent loop es la estructura de runtime más pequeña que hace que un model actúe como un agent. Observa el state, decide el siguiente paso, valida la acción propuesta, actúa a través de una interfaz limitada, registra la observación y se detiene cuando el éxito, el fallo, el presupuesto o una escalada indican que la ejecución terminó.

El loop es poderoso porque el siguiente paso puede depender de lo que acaba de ocurrir. Es peligroso por la misma razón. Sin state explícito, presupuestos, validación y razones de detención, el loop se convierte en un lugar donde se acumulan costos, latencia, uso inseguro de tools y fallos ocultos.

Este pattern es dueño de la iteración: propuesta, validación, acción, observación, presupuesto y razón de detención. No es responsable del business goal, la implementación de tools, el durable workflow ni el control plane de producción. Esas responsabilidades pertenecen al caller, la capa de tools, el workflow engine y el runtime.

Úsalo cuando

- La task no puede resolverse con una secuencia fija conocida de antemano.
- El siguiente paso depende de observaciones de tools, retrieval, usuarios o salidas intermedias.
- El agent puede necesitar varios intentos, pero esos intentos pueden estar limitados.
- El sistema puede persistir suficiente loop state para reproducir o depurar la ejecución.
- Puedes definir condiciones de éxito, fallo, agotamiento de presupuesto y escalada.

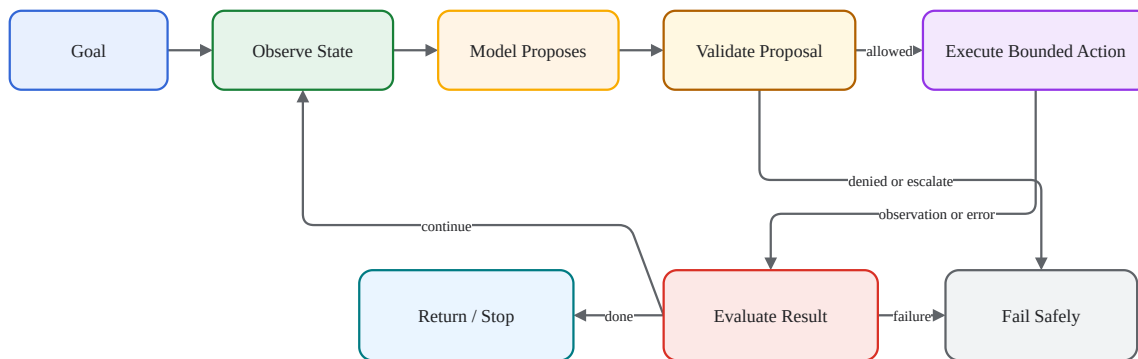
Evítalo cuando

- Un workflow determinista o prompt chain puede expresar claramente los pasos.
- La task no tiene observaciones intermedias útiles.
- El uso de tools, el costo, el runtime o los efectos secundarios no pueden ser limitados.
- El loop state viviría solo dentro del context del model.
- El equipo no puede explicar por qué el loop se detuvo.

Arquitectura

Usa este diagrama para leer Agent Loop como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el caller o un pequeño servicio de aplicación es dueño del task state hasta que se introduce un runtime pattern.

Agent Loop



Forma del sistema

- **Límite del pattern:** un loop controller es dueño del progreso, presupuesto, validación, actualizaciones de state y decisiones de detención.
- **Dueño del state:** el loop runtime es dueño del task-local state durante la ejecución y guarda checkpoints de state durable cuando la ejecución debe sobrevivir una interrupción.
- **Rol del model:** el model propone el siguiente paso, pero el software valida si el paso está permitido.
- **Límite de acción:** tools, handoffs, memory writes y respuestas finales pasan por validación tipada.
- **Promesa operativa:** el loop maneja la incertidumbre sin dar al model tiempo, tools o autoridad ilimitados.

Protocolo central

1. Recibe una task con un goal, caller, restricciones, presupuesto y trace ID.
2. Carga o inicializa el loop state: observaciones, evidencia, intentos, errores y presupuesto restante.
3. Ensambla el working set para el siguiente paso.
4. Pide al model que proponga una acción, un resultado final, una negativa o una escalada.
5. Valida la propuesta contra schema, tools, policy, state y presupuesto.
6. Ejecuta la acción permitida y registra la observación.
7. Actualiza el state, los contadores de presupuesto, eventos de trace y la razón de detención.
8. Se detiene en éxito, fallo, negativa, cancelación, agotamiento de presupuesto o escalada.

Notas de implementación

Mantén el controller simple. El loop debe ser fácil de inspeccionar.

```
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;
}

```

Un loop controller mínimo puede entonces hacer cumplir el límite:

```

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;
  }
}

```

El controller no necesita ser complicado. Debe hacer explícitas las partes ocultas: el goal activo, la propuesta, la decisión de validación, la acción, la observación, los contadores de presupuesto y la razón de detención.

Caso de uso: Investigación de reembolso limitada

En el sistema de soporte de reembolsos, todo el producto no debe ser un loop abierto. El workflow debe llamar a un loop solo para la parte incierta: investigar si la evidencia disponible respalda la recomendación de reembolso.

```

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

```

Esto mantiene el loop útil y pequeño. Puede decidir el siguiente paso para recolectar evidencia, pero no puede emitir el reembolso. La acción de pago queda detrás de la validación de policy, aprobación, idempotency y auditoría.

Modos de fallo

- El goal es vago, así que el loop sigue trabajando sin una condición de éxito estable.
- Se permite que el model decida cuándo debe detenerse el loop sin un presupuesto controlado por software.
- El loop repite la misma llamada a tool porque no compara nuevas observaciones con las anteriores.
- Los reintentos ocurren sin idempotency, duplicando efectos secundarios.
- El state existe solo en el prompt, así que la reproducción y recuperación son imposibles.
- El loop resume evidencia que era necesaria para la siguiente decisión.
- Los errores de tools se tratan como observaciones normales, causando acciones de seguimiento confusas.
- El agent declara éxito porque produjo una respuesta, no porque el goal fue satisfecho.
- Los traces capturan la salida final pero no el historial de iteraciones.

Estrategia de evaluación

Los evals de loop deben probar la trayectoria, no solo la respuesta final.

- Prueba una task que se complete dentro del presupuesto.
- Prueba una task que deba detenerse en `max_steps`.
- Prueba una task que deba detenerse en `max_tool_calls`.
- Prueba una task con un resultado de tool mal formado.
- Prueba una task con fallos repetidos de tool que debería escalar.
- Prueba una task donde el model proponga una acción prohibida.
- Prueba una task donde el model reclame finalización pero falte evidencia requerida.
- Prueba la reproducción desde un loop state guardado.

Un fixture de eval compacto puede hacer explícito el comportamiento de detención esperado:

```
{
  "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"]
}
}
```

Mide tasa de finalización, razón correcta de detención, tasa de acciones inválidas, tasa de acciones repetidas, precisión de escalada, costo de tokens y tools, latencia y éxito de reproducción.

Lista de verificación para producción

- Define el goal del loop y los criterios de éxito antes de la primera llamada al model.
- Almacena el state del loop fuera del context del model.
- Establece límites estrictos para pasos, llamadas a tools, tiempo de reloj, reintentos y costo.
- Valida cada acción antes de ejecutarla.
- Haz que las acciones con efectos secundarios sean idempotentes o requieran aprobación.
- Registra la propuesta, decisión de validación, acción, observación, error y motivo de detención en cada iteración.
- Trata la cancelación, rechazo, timeout y escalamiento como resultados normales.
- Agrega circuit breakers para fallas repetidas o propuestas de acciones inseguras.
- Mantén versionados los prompts del loop, manifiestos de tools, policies y rutas de model.
- Reproduce fallas de producción en regression evals.

La regla arquitectónica es simple: el model puede elegir la siguiente propuesta, pero el software decide si el loop continúa. Continúa con [Goals and State](#) para definir qué lleva el loop, luego con [Tool Use](#) para definir cómo actúa.

Ejecuta el ejemplo

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

Recorrido del código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el paquete de descarga y en el repositorio fuente.

```
agent-loop-pattern/typescript/src/agent_loop.ts
```

[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[] = [];

```


Extracto truncado para mayor legibilidad. Descarga el paquete o abre el archivo fuente para la implementación completa.

`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");

```

Descarga

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

El paquete de descarga contiene la carpeta actual `agent-loop-pattern/` de este repositorio.

Patrones relacionados

- [Goals and State](#)
- [Tool-Using Agent](#)
- [Structured Output](#)

- [Evaluator-Optimizer](#)
- [Durable Workflows](#)
- [Pattern Evaluation Checklist](#)
- [Evaluation-Driven Agent Development](#)

Goals and State

Los goals definen el éxito; el state registra el progreso. Juntos hacen que el trabajo del agent sea retomable, inspeccionable y más fácil de evaluar.

Fuente y descargas

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

Intent

El patrón Goals and State separa lo que el agent intenta lograr del state mutable que acumula mientras trabaja. Los goals definen el éxito. El state registra el progreso, restricciones, evidencia, trabajo pendiente, aprobaciones, presupuesto y razones de detención.

Este es el núcleo práctico de la working memory. La working memory no es memoria a largo plazo. Es state operacional con alcance de ejecución. Debe ser compacta, tipada, inspeccionable y desechable, a menos que otra policy promueva explícitamente parte de ella a durable memory.

Usa este pattern para state local de task. La durable memory se maneja con [Memory-Augmented Agent](#). El ensamblaje de context se maneja con [Context Engineering](#).

El model puede proponer un siguiente paso, una actualización de state o una razón de detención. El runtime es dueño de la transición de state. Ese límite evita que el agent reescriba silenciosamente lo que ha sucedido, lo que falta o por qué se le permite continuar.

Usa cuando

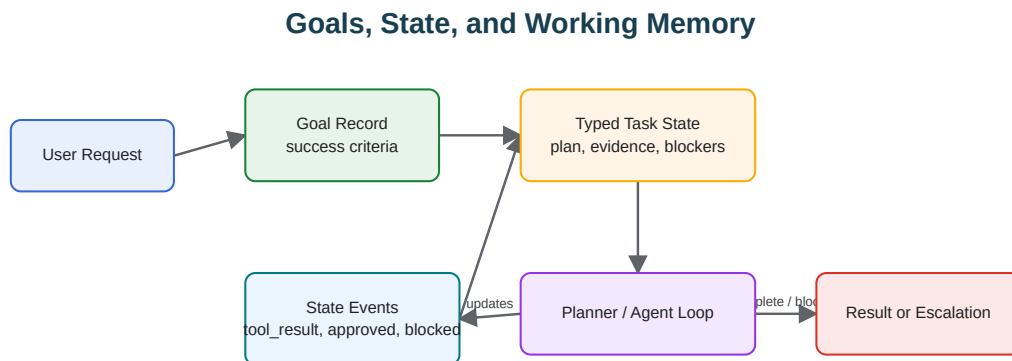
- Una task abarca múltiples turnos, tools, agents, reintentos, aprobaciones o pasos de workflow.
- Necesitas ejecución retomable después de una falla, interrupción, timeout o aprobación humana.
- El agent debe explicar el progreso respecto a un objetivo explícito.
- Necesitas un conjunto de trabajo compacto en vez de solo el historial de chat.
- La evaluación depende de la trayectoria, no solo de la respuesta final.

Evita cuando

- La task no tiene state y puede responderse en una sola llamada.
- El goal no puede expresarse como criterios de éxito observables.
- El state contendría datos sensibles que no puedes almacenar de forma segura.
- El sistema usaría el state como un scratchpad no estructurado y sin dueño.
- El runtime no puede reproducir ni inspeccionar las transiciones de state.

Arquitectura

Usa este diagrama para leer Goals and State como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: el llamador o un pequeño servicio de aplicación es dueño del state de la task hasta que se introduce un patrón de runtime.



Forma del sistema

- **Límite del pattern:** el límite de state es dueño del goal, working state, event log, reglas de transición, razón de detención y replay record.
- **Dueño del state:** el runtime, workflow engine o application service es dueño del state; el model solo es dueño de las actualizaciones propuestas.
- **Rol del model:** el model puede resumir el progreso, identificar brechas y proponer actualizaciones, pero no debe ser la fuente de verdad para el state.
- **Límite de policy:** los cambios de state que afectan permisos, aprobaciones, promoción de memory o efectos secundarios pasan por validaciones del runtime.
- **Promesa operacional:** el sistema puede explicar qué está haciendo, qué cambió, qué evidencia causó el cambio y por qué se detuvo o continuó.

Protocolo central

1. Crea un goal record con criterios de éxito, restricciones, dueño, clase de riesgo y condiciones de detención.
2. Inicializa la working memory con el state mínimo necesario para empezar.
3. Ejecuta un paso acotado mediante un planner, llamada al model, tool, worker, evaluator o puerta de aprobación.
4. Convierte observaciones en eventos de state tipados.
5. Aplica transiciones de state de forma idempotente.
6. Recalcula preguntas abiertas, trabajo pendiente, state de presupuesto, state de aprobación y razón de detención.
7. Continúa, reintenta, solicita aprobación, escala o detén según reglas explícitas.
8. Persiste suficiente state e historial de eventos para reproducir o auditar la ejecución.

Notas de implementación

El state debe ser más pequeño que la transcripción y más estructurado que un resumen. Es el modelo operativo para la ejecución actual.

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

El state debe cambiar a través de eventos, no por mutación oculta.

```
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 };
  }
}
```

Esto es intencionalmente simple. Lo importante es la regla: las transiciones de state son explícitas, tipadas, reproducibles y ligadas a observaciones.

Caso en ejecución: Refund Goal And State

Un asistente de reembolsos necesita más que una transcripción de chat porque los operadores deben saber qué evidencia usó el sistema antes de mover dinero. El goal record debe nombrar el resultado y el riesgo; el working state debe mostrar la evidencia, preguntas abiertas, state de aprobación y razón de detención.

```
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"
```

Este state le dice al runtime qué sucedió y qué falta. También le da a los evals algo concreto para inspeccionar: la falta de policy debe bloquear la recomendación, una aprobación denegada debe bloquear el pago y una ejecución reanudada debe preservar las mismas referencias de evidencia.

Reglas de promoción

La working memory no debe convertirse automáticamente en durable memory. Un paso completado, resultado de tool o corrección de usuario puede ser candidato para memoria a largo plazo, pero la promoción requiere una decisión de policy aparte.

Antes de promover la working memory, pregunta:

- ¿Esto es útil más allá de la ejecución actual?
- ¿La fuente es confiable y está referenciada?
- ¿El usuario o tenant permite que se almacene?
- ¿Hay datos sensibles que requieren redacción o expiración?
- ¿Es esto un evento, un hecho, una preferencia o una referencia de policy?
- ¿Existe un camino de corrección y eliminación?

Si la respuesta no es clara, mantenlo en el run trace y no lo promociones.

Modos de falla

- El state se convierte en un volcado de transcripción.
- El model reescribe silenciosamente el state sin un evento.
- El goal describe actividad, no éxito.
- Los subgoals se alejan del goal principal.
- Los resultados de tool se resumen sin referencias de evidencia.
- El state de aprobación, cancelación o presupuesto se pierde tras un reintento.
- El sistema no puede reanudar porque el state solo vivía en el context del model.
- El state obsoleto sobrescribe resultados frescos de tool.
- El state sensible se persiste sin reglas de retención o redacción.
- La working memory se promueve automáticamente a memoria a largo plazo.

Estrategia de Evaluación

Los evals de working memory deben inspeccionar la trayectoria, no solo la respuesta final.

- Prueba que una task de varios pasos crea un goal con criterios de éxito observables.
- Prueba que cada resultado de tool se convierte en un evento de state con una referencia de evidence.
- Prueba reintentos y verifica transiciones de state idempotentes.
- Prueba reanudar después de una interrupción.
- Prueba cancelación y verifica que la ejecución no continúe.
- Prueba esperas de aprobación y verifica que el approval state sobreviva al reanudar.
- Prueba agotamiento de presupuesto y verifica que la razón de detención quede registrada.
- Prueba resultados de tool obsoletos y verifica que la evidence más reciente prevalezca.
- Prueba ejecuciones bloqueadas y verifica que las preguntas abiertas se conserven.
- Prueba que la working memory no se promueva a durable memory sin policy.

Mide la completitud del state, validez de transiciones, éxito de replay, éxito de resume, tasa de state obsoleto, tasa de approval perdido, precisión de stop reason y precisión de promotion-policy.

Lista de Verificación para Producción

- Define goal schema, state schema, event schema y stop reasons.
- Mantén la working memory compacta y tipada.
- Almacena el state separado del historial de chat.
- Haz que las actualizaciones de state sean basadas en eventos, idempotentes y reproducibles.
- Adjunta referencias de evidence a resultados de tools, decisiones, aprobaciones y respuestas finales.
- Persiste approval, cancelación, presupuesto, reintentos y state bloqueado.
- Trata la promoción de state a durable memory como una decisión de policy separada.
- Redacta o evita datos sensibles en la working memory cuando sea posible.
- Traza lecturas, escrituras, transiciones, reintentos, reanudaciones y razones de detención del state.
- Convierte incidentes de lost-state y wrong-stop en regression evals.

Recorrido del Código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo circundante explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

Código Fuente

Este pattern actualmente no tiene un extracto de código dedicado. Usa los enlaces de source y descarga a continuación para acceder a la carpeta completa del pattern.

Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `goals-and-state-pattern/` actual de este repositorio.

Patrones Relacionados

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

Tool Use

El uso de tools le da a un agent acceso controlado a capacidades externas como calculadoras, búsquedas, bases de datos, archivos, ejecución de código, APIs o sistemas empresariales.

Fuente y descargas

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

Intento

El uso de tools permite que un agent cruce la frontera entre lenguaje y acción. El model puede proponer un cálculo, consulta, recuperación, operación de archivo, llamada a API, paso de workflow o acción de negocio, pero el software sigue siendo dueño del límite real de ejecución.

La idea importante es simple: el model no “usa el tool” directamente. El model propone una llamada a tool. El runtime valida la llamada, revisa la policy, ejecuta el tool, registra el resultado y decide si la observación puede influir en el siguiente paso.

Este pattern es dueño del límite entre propuesta y ejecución. No es responsable de la autorización de negocio, policy de aprobación humana, orquestación durable ni de la implementación interna de cada capability. Mantén esas responsabilidades en las capas de policy, aprobación, workflow y servicios.

Usar cuando

- El task necesita hechos, cálculos, recuperación o acceso a sistemas fuera del context del model.
- El tool puede expresarse como una capability limitada con entradas tipadas y salidas estructuradas.
- El software puede validar los argumentos antes de la ejecución.
- Los permisos y revisiones de policy pueden ubicarse entre la intención del model y la ejecución del tool.
- El resultado puede ser trazado, reproducido, simulado o auditado.

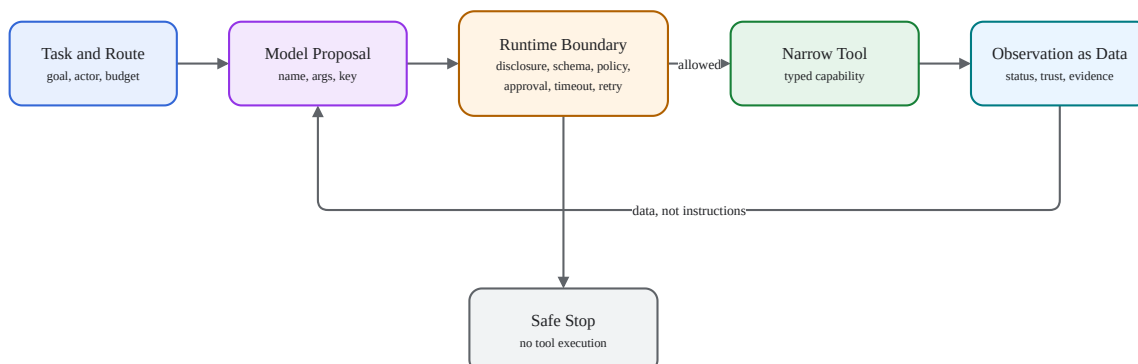
Evitar cuando

- Una función o workflow determinista puede hacer el trabajo sin selección del model.
- El tool propuesto es un primitivo amplio como `run_sql`, `send_http_request` o `execute_shell`.
- El tool puede crear efectos secundarios de alto riesgo sin aprobación.
- Los resultados del tool contienen contenido no confiable pero el sistema no puede separar datos de instrucciones.
- El equipo no puede explicar qué llamadas a tools están permitidas, prohibidas, reintentadas o escaladas.

Arquitectura

Usa este diagrama para leer Tool Use como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el llamador o un pequeño servicio de aplicación es dueño del task state hasta que se introduce un runtime pattern.

Tool Use Policy Boundary



Forma del sistema

- **Límite del pattern:** un agent runtime recibe un task, expone solo los tools necesarios para ese task, valida las llamadas propuestas por el model y retorna un resultado tipado.
- **Dueño del state:** el llamador, el motor de workflow o el agent runtime es dueño del task state. El model no debe ser el almacén durable de state.
- **Dueño del tool:** cada tool tiene un servicio o equipo responsable de schema, permisos, efectos secundarios, errores y campos de trace.
- **Límite de policy:** la ejecución del tool ocurre solo después de la validación de schema, autorización, revisión de presupuesto y reglas de aprobación.
- **Promesa operativa:** el uso de tools expande la capability del agent sin darle al model autoridad irrestricta.

Protocolo central

1. Recibe un task acotado con identidad del llamador, goal, referencia de state y presupuesto.
2. Selecciona el conjunto de tools más pequeño y útil para el task o fase actual.
3. Solicita al model la siguiente acción o respuesta final.
4. Valida cualquier llamada a tool propuesta contra schema, permisos, presupuesto y policy.
5. Ejecuta el tool con timeout, clave de idempotencia y correlación de trace.
6. Retorna el resultado del tool como datos de observación, no como nuevas instrucciones.
7. Detén, reintenta, escala o continúa según reglas explícitas del runtime.

Notas de implementación

Trata el registro de tools como una superficie de autoridad. Un registro pequeño es mejor que una lista grande de capabilities vagas.

```
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' };
}

```

El model puede elegir entre `read_order`, `search_refund_policy` y `draft_refund_request`, pero no puede inventar `issue_refund` a menos que el runtime exponga ese tool y la policy lo permita.

Usa resultados estructurados de tools:

```

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

```

No retornes cadenas simples para tools importantes. Las cadenas simples obligan al model a inferir si la llamada tuvo éxito, si es seguro reintentar y si el contenido es confiable.

Modos de falla

- Se permite al model llamar a un tool amplio que puede realizar muchas acciones ocultas.
- Las descripciones de tools se convierten en el único límite de permisos.
- Los argumentos del tool no se validan antes de la ejecución.
- Un reintento duplica un efecto secundario porque no hay clave de idempotencia.

- Los resultados del tool que contienen correos, páginas web, tickets o documentos se tratan como instrucciones.
- La respuesta final parece correcta, pero la trayectoria del tool usó un camino prohibido o inseguro.
- Los errores del tool son vagos, así que el agent reintent a ciegas o inventa evidencia faltante.
- Los trazes registran la respuesta final pero no la llamada propuesta, la decisión de policy, el resultado del tool y la razón de detención.

Estrategia de evaluación

Los evals de tool use deben probar tanto la capability como la restricción.

- Usa casos positivos donde el agent debe elegir el tool correcto con argumentos válidos.
- Usa casos negativos donde el comportamiento correcto es no llamar ningún tool, pedir input faltante, rechazar o escalar.
- Incluye casos de tools prohibidos como emisión directa de reembolsos, mensajería externa, ejecución de shell o exportación de datos privados.
- Simula tools para que los evals puedan inspeccionar la trayectoria sin tocar sistemas reales.
- Prueba resultados de tools malformados, timeouts, errores reintentables, errores fatales e instrucciones no confiables dentro de la salida del tool.
- Mide precisión en selección de tools, tasa de argumentos inválidos, tasa de llamadas no autorizadas, precisión en ruteo de aprobación, prevención de cadenas inseguras, costo, latencia y calidad de la razón de detención.

Un eval mínimo con tools simulados puede verse así:

```
{
  "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"
  }
}
```

El eval no solo revisa la respuesta final. Revisa el camino.

Lista de verificación para producción

- Mantén cada tool limitado y nombrado por su capability de negocio.
- Usa schemas tipados de entrada y salida.
- Declara clase de capability, efectos secundarios, permisos, reglas de aprobación y campos de trace.
- Valida los argumentos antes de la ejecución.
- Aplica permisos fuera del prompt.
- Agrega timeouts, límites de reintentos, claves de idempotencia y comportamiento de cancelación.
- Trata la salida no confiable del tool como datos, no instrucciones.

- Registra la llamada propuesta al tool, decisión de policy, resultado de ejecución, latencia, costo y razón de detención.
- Simula tools en los evals antes de conectar a sistemas de producción.
- Mantén un circuito de corte para tools riesgosos, rutas de model o capabilities del agent.

La regla arquitectónica es simple: expón la capability más pequeña que complete el task y valida cada uso propuesto antes de ejecutar. Continúa con [Tool Capability Design](#) para diseño de interfaces y [Human Approval Gates](#) para acciones de alto riesgo.

Ejecuta el ejemplo

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

Recorrido del código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo alrededor explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o flujo de control.

Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el repository source.

[tool-using-agent-pattern/typescript/src/tool_runtime.ts](#)

[Abrir fuente completa](#)

```
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",
    };
  }
}

```

Fragmento truncado para facilitar la lectura. Descarga el bundle o abre el archivo fuente para ver la implementación completa.

`tool-using-agent-pattern/typescript/test/tool_runtime.spec.ts`

[Abrir fuente completa](#)

```

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;

```

Fragmento truncado para facilitar la lectura. Descarga el bundle o abre el archivo fuente para ver la implementación completa.

Descargar

- [Descargar bundle de fuentes](#)
- [Abrir carpeta de fuentes](#)

El bundle de descarga contiene la carpeta `tool-using-agent-pattern/` actual de este repositorio.

Patrones relacionados

- [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 restringe las respuestas del model a datos tipados que el software puede validar y consumir.

Fuente y descargas

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

Propósito

El Structured Output Pattern restringe las respuestas del model a datos tipados que el software puede validar, enrutar, almacenar y probar. Es el límite entre el razonamiento en lenguaje natural y la lógica determinista de la aplicación.

Cuándo usar

- El output del model controla una llamada a tool, una rama de workflow, una decisión de policy o una escritura en base de datos.
- El código posterior necesita campos estables en lugar de prosa.
- Necesitas pruebas de regresión para el comportamiento asistido por model.

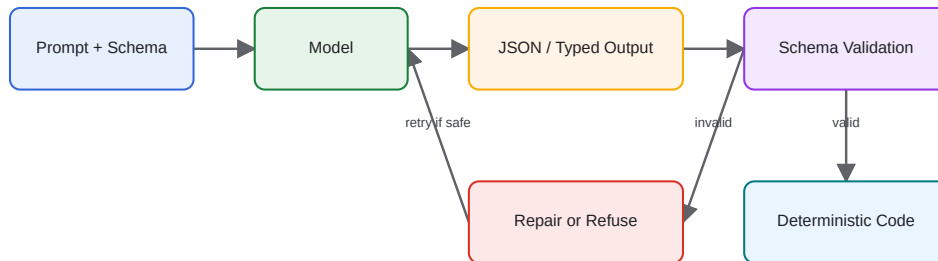
Evita cuando

- El output es prosa creativa solo para lectura humana.
- Un parser determinista ya maneja la entrada de forma segura.
- El schema es tan amplio que ya no restringe el comportamiento.

Arquitectura

Usa este diagrama para leer Structured Output como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: el caller o un pequeño servicio de aplicación posee el task state hasta que se introduce un runtime pattern.

Structured Output Validation



Forma del sistema

- **Límite del pattern:** una función de agent, clase o límite de servicio acepta input más context y devuelve una respuesta, acción o decisión tipada.
- **Propietario del state:** el caller o un pequeño servicio de aplicación posee el task state hasta que se introduce un runtime pattern.
- **Artifact principal:** `structured-output-pattern/` contiene la implementación de referencia ejecutable y ejemplos.
- **Promesa operativa:** Structured output restringe las respuestas del model a datos tipados que el software puede validar y consumir.

Protocolo central

1. Acepta un input, goal o solicitud de task acotado.
2. Ensambla las instrucciones mínimas útiles, context, state y descripciones de tools.
3. Ejecuta el model o helper determinista detrás de un límite tipado.
4. Valida el resultado antes de devolverlo a usuarios, tools o durable state.
5. Registra suficiente evidencia para explicar el output después.

Notas de implementación

- Define schemas cerca del código que los consume.
- Valida cada respuesta del model antes de usarla, incluso cuando el proveedor ofrece soporte de structured output.
- Prefiere enums para decisiones de enrutamiento y discriminated unions para outputs de múltiples acciones.
- Registra fallos de validación e intentos de reparación como datos de evaluación de primera clase.
- Mantén el output validado cerca de la siguiente acción de runtime. Un objeto válido aún debe pasar verificaciones de policy, aprobación y state antes de activar efectos secundarios.

Modos de falla

- Schemas que reflejan prosa y ofrecen poca seguridad.
- Corrección silenciosa de campos faltantes o inválidos.
- Reglas de formato solo en el prompt sin validador.
- Schemas demasiado estrictos que causan fallos frágiles ante variaciones inofensivas.
- Un objeto válido contiene valores no soportados que violan reglas de dominio o policy.
- Bucles de reparación ocultan fallos repetidos del model y aumentan el costo sin condición de parada.

Estrategia de evaluación

Evalúa sintaxis, semántica y seguridad posterior por separado. La validez del schema prueba que el output puede ser parseado. No prueba que los valores sean correctos o seguros para ejecutar.

- Prueba campos requeridos faltantes, campos extra, enums inválidos, tipos incorrectos y objetos anidados mal formados.
- Prueba valores que pasan la validación del schema pero violan restricciones de dominio, como un reembolso mayor al total del pedido.
- Prueba referencias de evidencia no soportadas y campos contradictorios.
- Prueba un intento de reparación, fallo repetido de reparación y la ruta final de rechazo o escalamiento.
- Prueba cambios de versión del schema contra fixtures almacenados y consumidores posteriores.
- Asegura que el output inválido nunca llegue a tools, decisiones de policy o durable state.

Usa validadores deterministas para estructura e invariantes de dominio. Usa revisión humana o de model solo para campos que requieren juicio.

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

Mide la validez de schema en el primer intento, tasa de validez de dominio, tasa de éxito de reparación, intentos de reparación por output aceptado, tasa de aceptación insegura, tasa de rechazo falso y compatibilidad de versión de schema.

Para el contrato de caso de eval compartido y el método de release-gate, consulta [Evaluation-Driven Agent Development](#).

Lista de producción

- Define el contrato de input, context, output y error.
- Mantén prompts, schemas y descripciones de tools versionados.

- Agrega pruebas deterministas para el comportamiento útil más pequeño.
- Registra decisiones del model sin filtrar secretos ni datos privados de usuarios.
- Define escalamiento humano para trabajo ambiguo, de alto riesgo o bloqueado por policy.
- Mantén el source bundle, capítulo generado, pruebas y artifact de despliegue en el mismo release.

Recorrido de código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o flujo de control.

Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y el 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 isStringArray(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" || !isStringArray(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" ||
      !isStringArray(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" || !isStringArray(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";  
  }  
}
```

Descarga

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

El bundle de descarga contiene la carpeta actual `structured-output-pattern/` de este repositorio.

Patrones relacionados

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

Agent Development Lifecycle

El desarrollo de agents no es solo escribir prompts y desplegarlos al final. Es ingeniería de producto con un componente probabilístico en su interior.

Esa distinción importa. Un prompt puede hacer que una demo funcione. Un lifecycle hace que el sistema sobreviva después de la demo: cuando los usuarios hacen preguntas poco claras, los tools fallan, las policies cambian, los costos aumentan y el model propone con confianza una acción incorrecta.

Usa este lifecycle cuando un agent atenderá usuarios reales, llamará tools, accederá a datos privados, modificará sistemas externos o funcionará por más de una sesión de demo.

El lifecycle tiene un solo objetivo: mantener la autonomía conectada con la evidencia, el state, la policy, la evaluación y las operaciones.

La primera disciplina es no empezar con el agent. Empieza con el task, el state que necesita, los tools que puede usar, el riesgo que genera y la evidencia que prueba el éxito. Agrega autonomía solo donde realmente aporta valor.

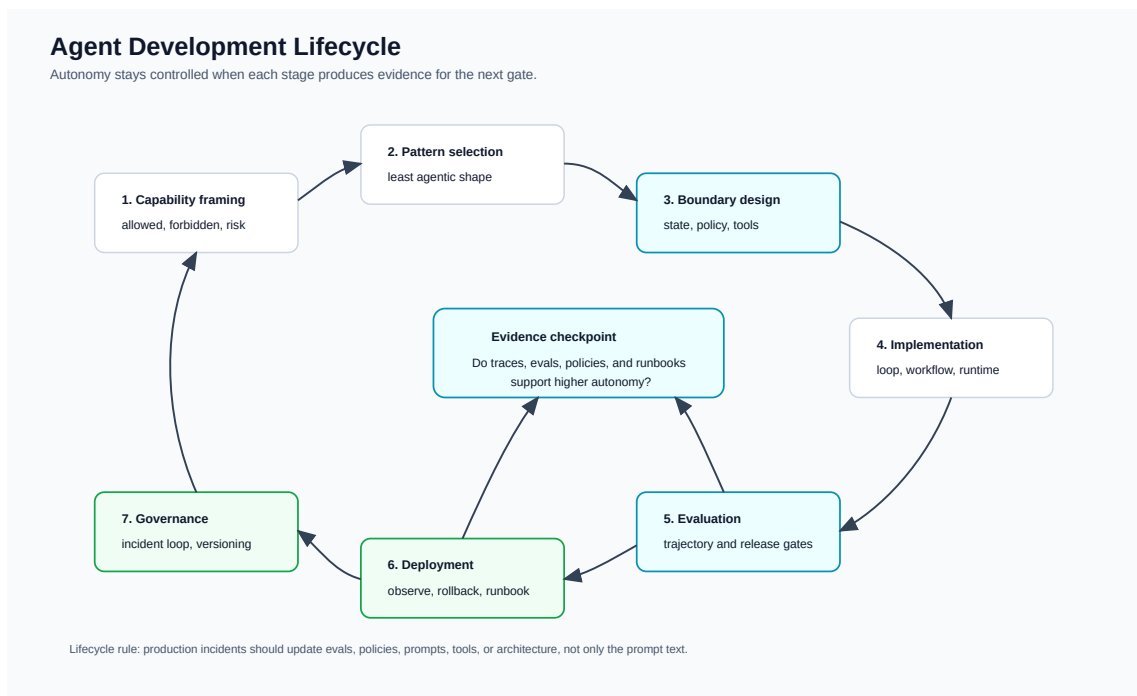
Qué Deberías Poder Hacer

Después de este capítulo, deberías poder:

- mover un agent de la idea a producción a través de gates explícitos;
- nombrar los artifacts requeridos antes de prototipo, piloto, producción y mayor autonomía;
- separar cambios de prompt de cambios de arquitectura, policy, tool, memory y eval;
- decidir cuándo la evidencia en producción justifica más autonomía;
- convertir incidentes en evals, actualizaciones de policy, cambios en runbooks o cambios de arquitectura.

Mapa del Lifecycle

Usa este diagrama para leer el lifecycle como un evidence loop, no como una secuencia de fases de proyecto. Cada etapa debe producir artifacts que hagan que el siguiente gate sea inspeccionable.



Etapas del Lifecycle

Stage	Main Question	Output
1. Capability framing	¿Qué debe hacer el agent y qué nunca debe hacer?	Capability map, exclusiones, clase de riesgo.
2. Pattern selection	¿Cuál es la arquitectura menos agentic que funciona?	Elección de pattern y presupuesto de complejidad.
3. Boundary design	¿Dónde terminan las decisiones del model y comienzan los controles de software?	Contratos de tool, policies, state model.
4. Implementation	¿Cómo percibe, decide, actúa y se detiene el agent?	Agent ejecutable, workflow o multi-agent system.
5. Evaluation	¿Cómo sabemos que funciona?	Eval suite, modos de falla, quality gates.
6. Deployment	¿Cómo corre de forma segura en producción?	Observability, rollback, límites de tasa, aprobaciones.
7. Governance	¿Cómo mejora sin perder el control?	Proceso de revisión, audit logs, versionado, incident loop.

El lifecycle es iterativo. Los datos de producción deben actualizar evals, policies, prompts, diseño de tools y, a veces, la arquitectura misma. Si los fallos en producción solo generan ajustes en el prompt, probablemente el equipo está tratando síntomas.

Development Gates

El lifecycle debe tener gates. Sin gates, una demo se convierte en piloto, un piloto en producción y la producción en autónoma sin que nadie tome la decisión explícitamente.

Gate	Must Be True
Prototype ready	El task está definido, existen criterios de éxito, las acciones inseguras están excluidas y existe una línea base.
Pilot ready	Contratos de tool, state model, fixtures de eval, traces y fallback humano están en su lugar.
Production ready	Revisión de seguridad, reglas de aprobación, observability, rollback, runbooks y release gates están en su lugar.
Higher autonomy ready	Traces de producción prueban bajo índice de override, evals cubren incidentes conocidos y rollback está probado.

Cada gate debe tener un responsable. Si nadie es dueño del gate, el gate no existe.

Conjunto de Artifacts del Lifecycle

El lifecycle se vuelve real cuando cada gate produce artifacts revisables.

Artifact	Created By	Used To Decide
Capability map	Producto e ingeniería.	Qué puede hacer el agent, qué no debe hacer y qué debe escalar.
Pattern decision	Dueño de arquitectura.	Si el sistema necesita un prompt, workflow, loop, tools, memory o varios agents.
Authority map	Seguridad, plataforma y equipo de features.	Qué acciones son solo lectura, solo borrador, requieren aprobación o están prohibidas.
State and context contract	Dueño de runtime.	Qué sobrevive a reintentos, reanudaciones, aprobaciones y handoffs.
Eval suite	Ingeniería y revisores de dominio.	Si el comportamiento es suficientemente bueno para avanzar un gate.
Trace schema	Dueño de operaciones.	Si los fallos pueden reconstruirse y reproducirse.
Rollback plan	Dueño de releases.	Cómo deshabilitar una ruta de model, prompt, tool, policy o capability.

Si falta un artifact, el gate debe indicarlo directamente. Una revisión vaga de “se ve bien” no es suficiente para sistemas que pueden actuar.

No Empieces Con El Agent

Antes de construir el agent loop, responde estas preguntas:

- ¿Qué intenta lograr el usuario?
- ¿Qué haría un workflow determinista?
- ¿Qué partes requieren juicio, recuperación, planeación o elección de tool?
- ¿Qué state debe sobrevivir reintentos o esperas de aprobación?
- ¿Qué tools son solo lectura, pueden escribir, son externas o de alto riesgo?
- ¿Qué evidencia prueba que el resultado es correcto?
- ¿Cuál es el comportamiento correcto cuando falta evidencia?
- ¿Qué debe negarse a hacer el sistema?
- ¿Qué se evaluará antes del lanzamiento?
- ¿Cómo se deshabilitará la capability si falla?

Si esas respuestas son vagas, construir un agent solo automatizará la ambigüedad.

Capability Framing

Empieza con un capability map, no con la elección de framework. Un framework no puede decirte qué está permitido que haga el agent.

Captura los trabajos del usuario y los resultados de negocio, las acciones permitidas y prohibidas, la evidencia requerida, los tools y datos necesarios, los puntos de aprobación, las restricciones de privacidad y cumplimiento, la latencia y costo aceptables y el comportamiento esperado ante fallos. Lo importante es el espacio negativo. Un agent con una lista clara de “no hará” es más fácil de lanzar que uno definido solo por lo que podría hacer.

Un buen capability framing suena concreto:

- “El agent puede redactar recomendaciones de reembolso, pero no puede emitir reembolsos.”
- “El agent puede consultar el historial de cuentas, pero no puede acceder a instrumentos de pago.”
- “El agent puede resumir un incidente, pero no puede llamar a un ingeniero de guardia sin aprobación.”

Si el equipo no puede escribir límites así de claros, el agent no está listo para implementación.

Pattern Selection

Selecciona el pattern más simple que resuelva el task. Aquí es donde muchos equipos sobreconstruyen. Un prompt resuelve un task acotado. Un prompt chain maneja fases conocidas. Routing maneja entradas que requieren diferentes caminos. Un agent loop resuelve el caso donde el siguiente paso depende de observaciones. Multi-agent systems manejan roles separados que requieren context, tools o permisos distintos.

No empieces con una arquitectura multi-agent solo porque el dominio tiene muchos tasks. Comienza agrupando capabilities en workflows estables. Separa agents solo donde los límites de rol son reales: context diferente, tools diferentes, permisos distintos, responsabilidades de revisión distintas o trabajo paralelo real.

El pattern incorrecto suele fallar de dos formas. Es demasiado simple y no maneja la incertidumbre. O es demasiado agentic y se vuelve costoso, lento y difícil de depurar. El lifecycle debe detectar ambos casos.

Usa una línea base determinista incluso si planeas lanzar un agent. La línea base te da algo contra qué comparar. Si un agent no puede superar un formulario, una página de búsqueda, un workflow o un prompt chain en calidad, cobertura o costo operativo, la autonomía extra aún no está justificada.

Boundary Design

Antes de implementar, define los límites del sistema. Este es el trabajo de arquitectura que evita que el model se convierta silenciosamente en toda la aplicación.

- ¿Quién es dueño del state?
- ¿Quién es dueño de la policy?
- ¿Quién controla los permisos de tool?
- ¿Quién controla los writes de memory?
- ¿Quién aprueba finalmente?
- ¿Qué se puede reproducir tras un fallo?
- ¿Qué se puede cambiar sin redeplegar código?

El model puede proponer acciones. El software valida, ejecuta, registra y aplica.

Este límite debe ser visible en el código. Los tool schemas no deben ser vagos. Las transiciones de state no deben estar ocultas en lenguaje natural. Las reglas de aprobación no deben vivir solo en el system prompt. Los writes de memory deben ser revisados o restringidos antes de convertirse en future context.

Implementation

La implementación debe hacer visible el lifecycle en el código. Un lector debe poder encontrar el goal object, el state model, el límite de tool, los chequeos de policy, la condición de stop y la emisión de trace sin tener que descifrar un prompt.

Unidades mínimas de implementación:

- un goal o request object;
- state tipado;
- tool schemas;
- chequeos de policy;
- condiciones de stop;
- eventos de trace;
- fixtures de eval;
- configuración de deployment.

Evita implementaciones donde el único artifact durable sea una transcripción de conversación. Las transcripciones son evidencia útil. No son un state model.

La implementación también debe hacer explícito el fallo. Una llamada a tool fallida, un chequeo de policy denegado, un stop por presupuesto y una cancelación de usuario son resultados distintos. No los colapsos en “el agent falló.”

Lista de Verificación de Preparación para Lanzamiento

Antes de que un agent llegue a producción, el equipo debe poder señalar artifacts concretos:

- owner y ruta de soporte;
- capability map y exclusiones;
- contrato de entrada, salida y errores;
- nivel de autonomía por acción;
- tool manifests y clases de capability;
- state schema y razones de detención;
- ensamblaje de context y memory policy;
- reglas de aprobación y ruta de escalamiento;
- revisión de seguridad y privacidad;
- eval suite con casos bloqueantes;
- plan de observability y trace schema;
- presupuestos de costo, latencia y tools;
- rollback, kill switch o desactivación de capability;
- ADRs para cambios de authority, memory, tools y autonomy.

Evaluación

Evalúa antes y después del lanzamiento. Los agents cambian su comportamiento cuando cambian los prompts, tools, models, memory, retrieval indexes, policies o la población de usuarios. La evaluación no es una lista de verificación de lanzamiento. Es el feedback loop del sistema.

La evaluación previa al lanzamiento debe cubrir tasks de happy-path, solicitudes ambiguas de usuarios, datos faltantes, fallas de tools, fallas de retrieval, intentos de prompt-injection, acciones que requieren aprobación y presupuestos de costo y latencia. La evaluación posterior al lanzamiento debe agregar fallas en producción, correcciones de usuarios, casos de intervención humana y edge cases encontrados en traces. Cada incidente serio debe dejar un caso de eval.

Los filtros de evaluación deben estar ligados a los lanzamientos. Una actualización de model, reescritura de prompt, cambio en el tool manifest, cambio en memory policy, cambio en retrieval index o cambio en approval policy debe ejecutar la regression suite relevante antes del despliegue.

Evals bloqueantes útiles incluyen:

- llamadas a tools prohibidas;
- falta de aprobación antes de efectos secundarios;
- escrituras inseguras en memory;
- evidencia obsoleta o faltante;
- violaciones de tenant-boundary;
- falla al detenerse por presupuesto o cancelación;
- citas alucinadas;
- mala recuperación ante fallas de tools.

Despliegue

El despliegue en producción requiere más que un endpoint alojado. Necesita control operacional: versionado de model y prompt, límites de tasa y presupuestos de costo, timeouts a nivel de tool, audit logs, trace IDs que abarquen tools y agents, alertas para eventos críticos, rutas de escalamiento humano y rollback para prompts, policies y tools. Si un despliegue no puede explicar qué hizo el agent, no está listo para trabajo de alto impacto. El operador debe poder responder cuál goal estaba activo, qué evidencia se usó, qué tools se invocaron, qué policies pasaron, qué cambió y por qué se detuvo la ejecución.

El rollout debe ser escalonado:

1. evals offline con tools simulados;
2. dogfood interno con tools de solo lectura;
3. piloto limitado con aprobación humana para efectos secundarios;
4. rollout canario en producción con presupuestos estrictos;
5. rollout más amplio solo después de que traces y overrides lo respalden;
6. mayor autonomía solo después de que la evidencia en producción lo justifique.

El objetivo no es ralentizar al equipo. El objetivo es que cada aumento en autonomía sea una decisión de ingeniería explícita.

Gobernanza

La gobernanza es el control loop alrededor del agent. Decide cómo cambia el sistema sin perder los límites que lo hicieron lo suficientemente seguro para lanzarse. Eso implica revisar nuevos tools, cambios en prompts y policies, memory schemas, cambios en eval, umbrales de aprobación, reportes de incidentes y upgrades de model.

Trata los cambios en agents como cambios de producto e infraestructura. Pequeñas ediciones en prompts pueden cambiar el comportamiento tanto como los cambios de código. Nuevos tools pueden cambiar la clase de riesgo de todo el sistema. Un upgrade de model puede invalidar resultados antiguos de eval. Un cambio en memory schema puede cambiar lo que el agent cree sobre usuarios recurrentes.

El ciclo de vida solo se completa cuando estos cambios son revisados, probados, desplegados, observados y retroalimentados en la siguiente iteración.

Modos de Falla

- El desarrollo orientado a demos pasa a producción sin nuevos controles.
- El equipo lanza un agent sin una línea base determinista.
- Cambios en prompts se lanzan sin regression evals.
- Se agregan tools antes de definir los límites de capability.
- Se habilita memory antes de que existan políticas de escritura, eliminación y corrección.
- El agent no tiene un owner claro después del lanzamiento.
- La aprobación humana existe en la UI pero no en la runtime policy.
- Los traces de producción no pueden reconstruir llamadas a tools, context, memory y decisiones.
- Upgrades de model ocurren sin medir el drift de comportamiento.
- Los incidentes solo llevan a ajustes en prompts, no a evals, cambios en policies o correcciones de arquitectura.

Preguntas para Revisión del Ciclo de Vida

En cada cambio importante, pregunta:

- ¿Cambió el nivel de autonomía?
- ¿Cambió la autoridad de tool?
- ¿Cambió la memory policy?
- ¿Cambió el context builder o retrieval index?
- ¿Cambió el model, prompt, routing o fallback?
- ¿Cambió la aprobación, presupuesto o comportamiento de detención?
- ¿Cambió la observability o redacción?
- ¿Qué evals prueban que esto sigue siendo aceptable?
- ¿Qué ADR o runbook debe actualizarse?
- ¿Cómo hacemos rollback?

Capítulos Relacionados

- [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

Un agent harness es la capa que rodea el loop. El loop llama al model, lee el state, decide, actúa y se detiene. El harness le da a ese loop un entorno de trabajo utilizable: tools, archivos, memory, skills, permisos, subagents, aprobaciones, traces y recuperación.

Esta es la parte que muchas personas confunden con el agent en sí. No es el agent. Es la shell operativa que hace que el agent sea útil para trabajo real.

El framework puede proveer un harness. El runtime puede alojar un harness. Pero la responsabilidad no desaparece: algo debe decidir qué puede ver el agent, qué puede hacer, qué state sobrevive, qué necesita aprobación, qué se traza y cómo se recupera la ejecución.

Este capítulo se encarga del entorno de trabajo del agent. No se encarga del proveedor del model, el workflow de negocio, la implementación de tools ni las operaciones de producción a nivel de flota. El framework, los servicios de aplicación y el runtime de producción son responsables de esos temas.

Este capítulo viene después del loop porque el loop solo es demasiado pequeño para trabajo real. Úsalo como el puente entre “¿qué está haciendo el agent?” y “¿qué entorno mantiene ese trabajo acotado?”. Los siguientes capítulos de producción llevan las mismas ideas de control a runtime, observability y seguridad.

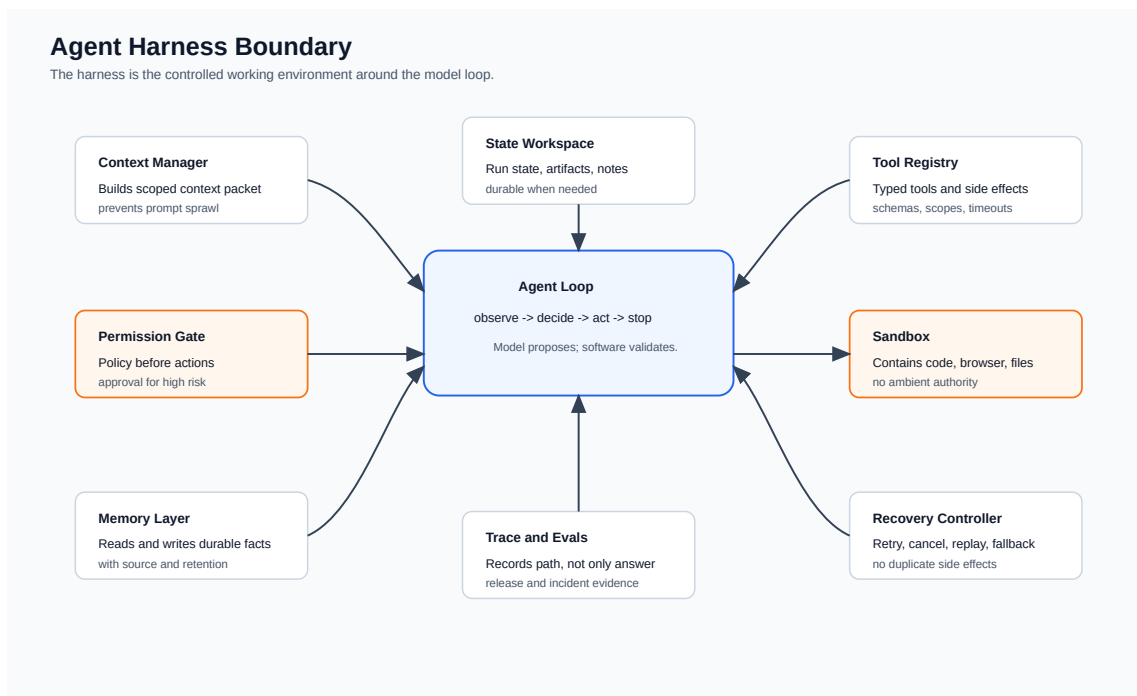
Qué Deberías Poder Hacer

Después de este capítulo, deberías poder:

- distinguir el agent loop del harness que lo rodea;
- nombrar los controles que un harness debe poseer antes de que el agent pueda hacer trabajo real;
- decidir qué características del harness se necesitan para una capability dada;
- evaluar el comportamiento del harness por separado de la calidad de respuesta del model;
- identificar los valores predeterminados del framework que requieren policy propia del producto.

Límite del Harness

Usa este diagrama para separar el model loop del entorno que hace que el loop sea seguro y útil. El harness posee los controles alrededor del loop: context, tools, policy, memory, traces, evals, sandboxing y recuperación.



Por Qué Existen los Harnesses

Un agent loop puro es lo suficientemente pequeño como para caber en unas pocas funciones. Las tareas reales necesitan más que eso: un lugar para guardar trabajo intermedio, una forma de cargar solo el context relevante, una manera de llamar tools de forma segura, una forma de dividir el trabajo entre subagents, una forma de pausar para aprobación, una forma de recordar hechos útiles, una forma de recuperarse tras fallas y una forma de inspeccionar lo que sucedió después. Todo eso junto es el harness.

Los coding agents lo hacen obvio. Un coding agent no solo llama a un model. Lee archivos, edita archivos, ejecuta pruebas, rastrea tasks, invoca tools, solicita aprobación, maneja fallas y mantiene un trace de la sesión. El model importa, pero el harness es lo que decide si el trabajo está controlado.

Harness vs Framework vs Runtime

Los términos se superponen, pero la distinción vale la pena.

Capa	Trabajo Principal	Ejemplos de Responsabilidad
Framework	Da abstracciones para construir agents.	Adaptadores de model, llamada de tools, chains, agents, prompts.
Runtime	Ejecuta y opera el trabajo del agent.	State durable, retries, streaming, deployment, persistence.
Harness	Da al agent un entorno de trabajo.	Archivos, skills, memory, subagents, permisos, aprobaciones, gestión de context.

Un producto puede usar los tres. Un framework puede incluir un runtime, y un runtime puede incluir características de harness. Los nombres importan menos que las responsabilidades, y la pregunta que corta la superposición es simple: ¿qué capa posee qué control?

Si la respuesta es “el framework lo maneja”, sigue preguntando. ¿Qué componente almacena el state? ¿Qué componente aplica el permiso? ¿Qué componente posee las escrituras de memory? ¿Qué componente puede detener un efecto secundario? ¿Qué componente emite el trace? Esas son responsabilidades del harness, ya sea que el código viva en tu aplicación o en un framework.

Matriz de Responsabilidad del Harness

Usa esta matriz al revisar un framework, SDK o harness personalizado.

Responsabilidad	Pregunta del Producto	Evidencia del Harness
Control de context	¿Qué vio el model en esta llamada?	Context packet, refs omitidas, trust labels.
Divulgación de tools	¿Qué tools eran visibles en este state?	Tool profile por ruta, rol y estado de aprobación.
Aplicación de permisos	¿Por qué se permitió o negó una acción?	Policy decision con actor, target, modo y razón.
Acceso a memory	¿Por qué se leyó o escribió memory?	Memory policy ref, source ref, retención, ruta de eliminación.
Vinculación de aprobación	¿Qué acción exacta aprobó el humano?	Approval ID ligado a tool, args, actor y expiración.
Recuperación	¿Puede esta ejecución reanudarse o reproducirse de forma segura?	Checkpoints, idempotency keys, modo de replay con tools simulados.
Observability	¿Puede un operador reconstruir la ejecución?	Trace events para propuesta, validación, acción, resultado, razón de detención.

Si el harness no puede producir evidencia para una fila, esa fila sigue siendo un riesgo de diseño, incluso si el demo funciona.

Capacidades Básicas del Harness

Un harness capaz normalmente expone estas superficies.

Capability	Qué Hace	Riesgo Principal
Workspace	Da al agent archivos, artifacts, notas y state temporal.	Fuga de datos sensibles o artifacts obsoletos.
Context manager	Selecciona lo que entra al context del model.	Exceso de tokens, evidencia faltante, drift en resúmenes.
Tool registry	Expone tools con schemas y descripciones.	Tools demasiado amplios, acciones inseguras, autorización débil.
Skill loader	Carga instrucciones procedurales solo cuando se necesitan.	Activación irrelevante o scripts inseguros ocultos.

Capability	Qué Hace	Riesgo Principal
Memory layer	Lee y escribe información durable.	Memory obsoleta, privada o contaminada.
Planner	Rastrea tasks, subtasks y progreso.	Los planes se vuelven teatro en vez de control.
Subagent manager	Delegar trabajo a contextos aislados.	Fragmentación de traces y propiedad poco clara.
Permission gate	Bloquea o pausa acciones riesgosas.	Aprobación solo por prompt o fatiga de aprobación.
Sandbox	Contiene acciones de código, navegador, shell o archivos.	Escape, exposición de secretos o acceso de red sin control.
Trace surface	Registra decisiones, llamadas a tools, costos y errores.	Logs solo de respuesta final.
Recovery controller	Maneja retry, cancelación, replay, fallback y escalamiento.	Efectos secundarios duplicados o progreso perdido.

No todo agent necesita todas las capabilities. Un clasificador de soporte limitado puede necesitar casi nada de esto; un coding o research agent probablemente necesita la mayoría.

Contrato del Harness

Un harness debe tener un run context explícito. Este es el objeto, o conjunto de objetos, que evita que el agent loop se convierta en un montón de variables globales.

```
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;
};
```

La forma exacta puede variar, pero el harness debe conocer el actor, tenant, goal, state, tools, memory policy, approval policy, budget y trace antes de que el model proponga la siguiente acción.

El mismo contrato debe hacer posible la cancelación:

```
type HarnessStopReason =
  | "success"
  | "blocked"
  | "approval_required"
  | "budget_exhausted"
  | "cancelled"
  | "policy_denied"
  | "tool_failure";
```

Si un harness no puede representar razones de detención, eventualmente convertirá diferentes fallas en el mismo error vago.

Workspace y Artifacts

Las tareas largas necesitan un workspace, y este se llena rápidamente con archivos proporcionados por el usuario, documentos recuperados, borradores generados, parches de código, planes, notas de tasks, salidas de tools y resultados de evaluación. Nada de eso debe tratarse como un basurero. Los archivos y artifacts necesitan reglas de propiedad, nombres, retención y visibilidad como cualquier otro state.

Un buen diseño de workspace responde algunas preguntas desde el principio: qué puede leer el agent, qué puede escribir, qué puede inspeccionar el usuario, qué es temporal, qué se convierte en state durable y qué nunca debe entrar al context del model. El workspace es parte del modelo de seguridad, no algo separado.

El workspace también necesita reglas de limpieza. Archivos temporales, borradores generados, salidas de tools, documentos descargados y logs no deben acumularse hasta convertirse en memory accidental. Mantén la retención, redacción y eliminación explícitas.

Gestión de Context

El harness decide qué ve el model, y esa decisión suele ser más importante que el prompt. Un model potente con mal context aún falla; un model más pequeño con un context limpio suele funcionar.

El context manager controla las instrucciones, el goal activo, el state actual, los archivos relevantes, la evidencia recuperada, los tools disponibles, la memory seleccionada, las observaciones recientes, y las reglas de presupuesto y detención. Todo esto debe cargarse de manera deliberada. Poner todo el workspace, la conversación completa, cada tool, toda la memory y todos los documentos recuperados en cada llamada

no es context engineering. Es context flooding. Para las reglas operativas detrás de esto, consulta [Context Budgets And Working Sets](#); un harness debe poder explicar el working set que ensambló para cada llamada al model.

Un buen harness puede responder, para cada llamada al model: qué context packet se ensambló, qué archivos y memories se incluyeron, qué tools se revelaron, qué fuentes se omitieron y por qué.

Skills y Divulgación Progresiva

Las skills permiten que un harness mantenga pequeño el base agent. Al iniciar, el agent puede ver solo los nombres y descripciones cortas de las skills disponibles. Cuando una se vuelve relevante, el harness carga las instrucciones, referencias, scripts, plantillas o ejemplos más profundos asociados.

Eso es divulgación progresiva, y rinde frutos en el presupuesto de tokens, especialización de dominio, procedimientos administrados por el equipo, trabajo repetible y uso más seguro de tools. Trata las skills como artifacts versionados y probados, no como fragmentos sueltos de prompt. Una skill es código procedimental y documentación que puede cambiar el comportamiento del agent, así que merece la misma disciplina que cualquier otro código.

Subagents y Aislamiento de Context

Los subagents valen la pena cuando una task se beneficia de un context aislado: cuando el trabajo puede hacerse en paralelo, cuando un especialista necesita instrucciones o tools diferentes, cuando una task no debe contaminar el context principal, o cuando el main agent debe recibir un resumen en vez de cada detalle. No los uses como decoración. Cada subagent agrega costo, latencia, coordinación y complejidad en el trace.

El harness debe hacer visible el trabajo de los subagents: quién delegó la task, qué context se pasó, qué tools estaban disponibles, qué resultado regresó, qué evidencia lo respalda y cómo lo usó el main agent. Si el trabajo de un subagent no puede ser trazado, no puede ser confiable.

Por defecto, los subagents deben heredar menos autoridad, no más. El harness debe pasar un goal delimitado, un context packet delimitado, un perfil de tool delimitado y una relación de trace hacia la ejecución principal.

Permisos y Aprobaciones

El harness aplica límites de permisos antes de la ejecución. El model puede proponer leer un archivo, llamar una API, ejecutar un comando, enviar un mensaje, actualizar un registro o escribir en la memory. El harness decide si cada una de esas acciones realmente está permitida.

Los buenos sistemas de permisos hacen distinciones que el model no puede hacer por sí mismo: lectura versus escritura, local versus remoto, seguro versus riesgoso, reversible versus irreversible, visible para el usuario versus oculto, barato versus caro, aprobado versus no aprobado. La aprobación debe ser específica. “Permitir tools” es demasiado amplio para significar algo. “Permitir que este agent actualice el ticket [INC-2048](#) con este resumen” es algo que una persona realmente puede revisar.

La policy del harness puede expresarse como un 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 };
}

```

Eso es el harness haciendo su trabajo: la intención del model se convierte en una solicitud verificada antes de que ocurra cualquier cosa.

El mismo pattern debe aplicarse a escrituras en memory, escrituras de archivos, comandos de shell, acciones en el navegador, mensajes externos y handoffs. El model propone. El harness valida. El tool o workflow ejecuta.

Memory en un Harness

La memory del harness es orquestación, no una nueva taxonomía de memory. El harness decide cuándo leer o escribir el state local de la task, la memory durable, la evidencia recuperada o las skills procedimentales, pero las policias viven en las capas canónicas: [Working Memory](#), [Memory-Augmented Agent](#), [Semantic Recall And RAG](#) y [Context Engineering](#).

La regla práctica es simple: el acceso a la memory es opt-in por task. No cargues la memory global del usuario, la memory del proyecto, la memory episódica o los documentos recuperados solo porque existen. Cárgalos porque son relevantes, permitidos, suficientemente recientes y corregibles. Escribe en la memory solo mediante eventos explícitos con reglas de fuente, alcance, retención y eliminación.

Sandboxes

Los harnesses se vuelven críticos para la seguridad en el momento en que exponen ejecución de código, comandos de shell, control del navegador o escrituras en el filesystem. Un sandbox debe controlar el acceso al filesystem, acceso a la red, variables de entorno, secretos, paquetes instalados, tiempo de vida de procesos, acceso al portapapeles y perfil del navegador, y rutas de subida y descarga.

El mejor valor por defecto es el menor privilegio: dale al agent el entorno más restringido que aún le permita completar la task, y amplíalo solo cuando una necesidad específica lo requiera.

Recuperación y Replay

El harness es responsable de lo que sucede tras una interrupción o falla. Un harness serio puede pausar, cancelar, reintentar, reanudar, hacer replay o escalar sin perder el state.

La recuperación requiere disciplina con los efectos secundarios:

- los reintentos necesitan claves de idempotencia;
- las aprobaciones requieren vinculación exacta a la acción;
- las ediciones de archivos necesitan diffs o snapshots;
- los mensajes externos necesitan registros de envío;
- las escrituras en memory necesitan IDs de escritura;
- las llamadas a tools necesitan IDs de correlación;
- el replay no debe repetir efectos secundarios inseguros.

Un modo de replay debe poder ejecutarse con tools simulados, context packets congelados y salidas del model grabadas al depurar. Así es como los equipos comparan cambios de prompt, model, policy o tool sin volver a emitir reembolsos, reenviar correos o reescribir memory.

Observabilidad del Harness para Background Agents

Los background agents necesitan un harness que pueda ser inspeccionado después de ejecutarse. Como mínimo, captura:

- la task inicial y los criterios de aceptación;
- archivos, tools y models seleccionados;
- llamadas a tools con entradas, salidas, decisiones de policy y errores;
- checkpoints o resúmenes de handoff tras fases importantes;
- comandos de verificación final y sus resultados.

Sin este registro, un background agent se convierte en un trabajador opaco. Con él, el agent se vuelve infraestructura revisable.

Evaluación del Harness

Evalúa el harness directamente. Congela o simula la respuesta del model cuando sea posible para que la prueba aísle el comportamiento del harness en lugar de la calidad del model.

- Prueba la selección y exclusión de context.
- Prueba la revelación de tools por ruta y rol.

- Prueba la denegación de permisos antes de la ejecución.
- Prueba las esperas de aprobación y la reanudación.
- Prueba la cancelación antes y durante la ejecución de tools.
- Prueba el reintento con idempotencia.
- Prueba la policy de escritura en memory.
- Prueba la trazabilidad del handoff de subagents.
- Prueba los límites del sandbox.
- Prueba la completitud del trace tras una falla.
- Prueba el replay sin efectos secundarios.

Los evals de harness son distintos de los evals de respuesta. Preguntan si el shell operativo mantuvo el control cuando el model propuso algo riesgoso, incompleto, costoso o incorrecto.

Un fixture de harness debe describir la trayectoria propuesta y los controles que deben mantenerse:

```
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;
  };
};
```

Mantén un conjunto pequeño de casos bloqueantes para el harness: revelación prohibida de tools, evasión de permisos, evasión de aprobaciones, context cruzado entre tenants, sobrepaso de presupuesto, cancelación seguida de un efecto secundario, ejecución duplicada tras reintento y replay que llega a un tool de escritura en vivo.

Mide la precisión en la revelación de tools, la corrección en la denegación de permisos, la tasa de evasión de aprobaciones, la tasa de enforcement de presupuesto, la latencia de cancelación, los efectos secundarios duplicados, las violaciones de policy de context, la completitud del trace, el aislamiento en replay y el éxito en la recuperación. Segmenta los resultados por perfil de tool, nivel de autonomía y versión del harness.

Para el contrato de casos de eval compartidos y el método de release-gate, consulta [Evaluation-Driven Agent Development](#).

Harness Failure Modes

Los harnesses fallan de maneras predecibles. Cada tool está disponible todo el tiempo. Cada archivo entra al context. Las escrituras en memory son invisibles. Los subagents crean trabajo que nadie posee. Las approvals son demasiado amplias. Los límites del sandbox no son claros. Los traces omiten los inputs y outputs de las tools. Las skills se vuelven obsoletas pero siguen cargándose. La compresión del context elimina justo la evidencia que necesitas para depurar el run. Cada uno de estos es una falla de arquitectura, no una falla del model, y por eso un mejor model no te salvará de ellas.

Agrega algunos más a la lista: la cancelación no detiene los side effects en cola, los retries duplican trabajo, los subagents heredan permisos amplios, los defaults del framework habilitan memory silenciosamente, los workspace artifacts se vuelven context oculto, y el replay es imposible porque el harness no guardó context packets ni resultados de tools.

Design Checklist

Antes de adoptar o construir un harness, pregunta:

1. ¿Qué posee el harness?
2. ¿Qué state persiste?
3. ¿Qué puede ver el model?
4. ¿Qué puede proponer el model?
5. ¿Qué puede ejecutar el harness sin aprobación?
6. ¿Qué tools están delimitadas por rol o task?
7. ¿Qué archivos se pueden leer o escribir?
8. ¿Qué memory se puede escribir?
9. ¿Qué trabajo de subagent es trazable?
10. ¿Qué sucede cuando el run se interrumpe?
11. ¿Qué ve el operador después de una falla?
12. ¿Cómo se puede evaluar un cambio en el harness antes de lanzarlo?
13. ¿Cómo se enlazan en los traces los context packets, tool calls, approvals, memory writes y subagent runs?
14. ¿Cómo hace el harness replay de un run sin repetir side effects?
15. ¿Qué defaults del framework han sido reemplazados por policy propiedad del producto?

Si el harness no puede responder estas preguntas, no está listo para sostener un agent serio.

Production Checklist

- Define el context de ejecución del harness.
- Mantén goal, state, context, tools, memory, approvals, budget y trace enlazados.
- Divulga las tools progresivamente por ruta y state.
- Haz cumplir los permisos fuera del prompt.
- Ata las approvals a acciones exactas.
- Haz que la cancelación y la pausa sean estados reales del runtime.
- Mantén las escrituras en memory explícitas, delimitadas, revisables y eliminables.
- Delimita los subagents por context, tools y permisos.

- Aísla acciones de alto riesgo con límites en filesystem, network, secrets y procesos.
- Soporta replay con tools simuladas y sin side effects repetidos.
- Evalúa el comportamiento del harness por separado de la calidad de la respuesta final.

Design Rule

El model aporta el juicio. El harness aporta el control.

Una vez que el contrato del harness es explícito, continúa con [Production Runtime Overview](#). Ese capítulo pasa del entorno de trabajo de un agent a colas, despliegues, presupuestos, recuperación y operadores.

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](#)

Construyendo un Agent Runtime Mínimo

Este capítulo explica el pequeño runtime detrás de los laboratorios de mini-framework desde cero. No lo construyes porque los equipos de producción deban evitar frameworks maduros. Lo construyes porque un runtime pequeño hace visible la arquitectura real.

Los frameworks cambian APIs y vocabulario. Las mismas responsabilidades siguen regresando: state, decisiones, tools, policy, context, traces, evals y condiciones de parada. Una vez que puedes construir esos primitivos en un runtime pequeño, sistemas al estilo de LangGraph, Mastra AI, AutoGen, CrewAI, MCP, A2A y harnesses personalizados son más fáciles de evaluar.

Lo Que Deberías Poder Hacer

Después de este capítulo, deberías poder:

- identificar los primitivos del runtime ocultos dentro de los agent frameworks;
- explicar por qué el model propone decisiones pero el runtime las ejecuta;
- construir un loop pequeño con state, policy, tools, context, traces y razones de parada;
- comparar características de frameworks por responsabilidad en vez de vocabulario;
- saber cuándo el runtime de aprendizaje debe dar paso a la infraestructura de producción.

Por Qué Construir Uno

La mayoría de las fallas de agent no son fallas misteriosas del model. Son fallas del runtime.

El sistema no sabía quién era dueño del state. La lista de tools era demasiado amplia. Policy vivía en un prompt. El loop no tenía razón de parada. El context se armaba volcando todo en el model. Una respuesta final parecía aceptable, pero el camino usó un tool prohibido. Ninguna de esas cosas se arregla cambiando nombres de frameworks.

Un runtime mínimo enseña el límite de control:

```
goal
-> build context
-> ask for a decision
-> validate the decision
-> check policy
-> execute allowed work
-> record observation
-> evaluate stop condition
```

Esa es la forma oculta bajo muchas abstracciones de frameworks.

Usa este diagrama para mantener separadas las responsabilidades del runtime. El model propone una decisión, pero el runtime es dueño de la validación, policy, ejecución, observación y condiciones de parada.

Lo Que Este Runtime No Es

Esto no es un framework de producción. No intenta resolver despliegue, streaming, ejecución distribuida, persistencia, autenticación, colas de workflow, adaptadores de model, integración de UI, backends de tracing o almacenes de memory.

Úsalo como andamio de aprendizaje. Usa frameworks maduros cuando necesites durabilidad de producción, integraciones operativas, concurrencia, checkpoints, reintentos, observabilidad alojada y soporte de ecosistema.

Registro de Preparación del Runtime

Antes de adaptar el mini-runtime a un sistema real, anota qué primitivos son solo para aprendizaje y cuáles son propiedad de producción.

```
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"
```

El registro previene el error común: tratar un runtime educativo útil como si ya tuviera durabilidad, seguridad y operaciones de producción.

Primitivo 1: State

El state es la fuente de verdad para la ejecución. El transcript no es suficiente. Un transcript dice lo que se dijo; el state dice lo que el sistema intenta hacer, lo que pasó, lo que se observó, lo que falta y por qué se detuvo la ejecución.

```

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;
};

```

Un buen state te permite reanudar, reproducir, depurar, evaluar y explicar una ejecución. Un mal state obliga a los operadores a inferir el comportamiento a partir del texto final.

Primitivo 2: Decision

La respuesta de un model es una propuesta. El runtime convierte esa propuesta en una decisión tipada antes de que pueda afectar tools, usuarios, state durable o sistemas externos.

```

type Decision =
  | { kind: "answer"; text: string }
  | { kind: "tool"; name: string; input: unknown }
  | { kind: "ask_human"; question: string }
  | { kind: "stop"; reason: StopReason };

```

Esta es la división más importante en el runtime: el model puede sugerir, pero el software valida y ejecuta.

Primitivo 3: Loop

El loop es dueño del progreso. Repetidamente construye el context, pide una decisión, valida esa decisión, ejecuta el trabajo permitido, registra observaciones y se detiene.


```

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;
}

```

El loop nunca debe ejecutarse solo porque el model sigue preguntando. Se ejecuta porque el runtime aún tiene presupuesto, la policy permite el siguiente paso y no se han cumplido las condiciones de parada.

Primitivo 4: Tool Registry

Los tools son capabilities. Un registry define las capabilities que el runtime puede exponer.

```

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>;
};

```

Mantén los tools acotados. Prefiere `lookup_order_summary` sobre `run_sql`, `draft_refund_request` sobre `post_http` y `search_policy_docs` sobre acceso irrestricto al navegador o shell.

Primitivo 5: Policy Gate

El registry dice qué existe. La policy decide qué está permitido ahora.

```
type PolicyDecision =  
  | { status: "allow" }  
  | { status: "deny"; reason: string }  
  | { status: "approval_required"; reason: string };  
  
type PolicyContext = {  
  actorId: string;  
  route: string;  
  approvedActionIds: string[];  
  remainingSteps: number;  
};
```

Un policy gate útil considera actor, ruta, tenant, tool, efecto secundario, estado de aprobación, sensibilidad de datos y presupuesto. Un prompt que dice “no hagas cosas peligrosas” no es un policy gate.

Primitivo 6: Context Packet

El context no es todo lo que el sistema sabe. Es el conjunto de trabajo para una decisión.

```
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 }>;  
};
```

El runtime debe poder explicar por qué cada elemento entró al context y por qué otro material disponible quedó fuera.

Primitivo 7: Trace

Los traces hacen que el comportamiento sea revisable. Sin ellos, el debugging se reduce a leer respuestas finales y adivinar.

```
type TraceEvent = {  
  runId: string;  
  step: number;  
  type:  
    | "context_built"
```

```
| "decision"
| "policy_decision"
| "tool_result"
| "stop";
data: unknown;
};
```

Los eventos de trace deben conectar la decisión del model, el resultado de policy, la llamada a tool, la observación, el costo, la latencia y la razón de parada.

Primitivo 8: Eval Harness

Los agent evals deben inspeccionar trayectorias, no solo respuestas.

```
type EvalCase = {
  caseId: string;
  input: string;
  expected: {
    toolsCalled?: string[];
    toolsNotCalled?: string[];
    stopReason: StopReason;
  };
};
```

Evals útiles detectan tools prohibidos, evidencia faltante, saltos de aprobación, decisiones inválidas, efectos secundarios repetidos y agotamiento de presupuesto. Una respuesta final plausible no es suficiente si la trayectoria fue insegura.

Cómo se Mapea Esto a los Frameworks

Runtime Primitive	LangGraph	Mastra AI	Sistemas estilo AutoGen	CrewAI
State	graph state y checkpoints	workflow y memory state	conversation/session state	flow state
Decision	node output o router result	agent response o workflow step	agent message	task output
Loop	graph traversal	workflow/agent runtime	conversation turn loop	flow execution
Tool registry	tools ligados a nodes o agents	tools	callable functions/tools	role tools
Policy gate	guard node o middleware	workflow/tool policy	manager o wrapper	flow guard o task constraint

Runtime Primitive	LangGraph	Mastra AI	Sistemas estilo AutoGen	CrewAI
Context packet	node input state	agent context y memory	message set	task context
Trace	callbacks y checkpoints	observability/evals	logs y messages	task y flow logs
Eval harness	graph-level tests	eval suites	transcript/trajectory tests	task/flow quality checks

Los frameworks pueden empaquetar estos primitivos, pero no eliminan la necesidad de diseñarlos.

La comparación importante es la responsabilidad, no la forma del API:

Pregunta	Si lo construyes tú mismo	Si usas un framework
¿Quién es dueño del state?	Tu runtime data model y plan de persistencia.	El framework puede proveer state containers o checkpoints, pero tu aplicación aún define el business state.
¿Quién autoriza los tools?	Tu policy function, approval records y audit trail.	El framework puede exponer hooks o middleware, pero la product policy sigue estando fuera del prompt.
¿Quién arma el context?	Tu context builder elige memory, evidence, tools y omisiones.	El framework puede proveer memory abstractions, pero aún necesitas reglas de fuente, frescura y privacidad.
¿Quién evalúa el comportamiento?	Tus tests inspeccionan decisions, tools, traces y stop reasons.	El framework puede correr evals, pero tú decides qué significa un comportamiento inseguro o de baja calidad.
¿Quién maneja fallas en producción?	Debes agregar retries, idempotency, durability, alerts y incident workflow.	Runtimes maduros pueden proveer partes de esto, pero deben configurarse según tu risk model.

Qué Hacen los Labs

Los labs de mini-framework implementan los primitivos en tres pasos:

1. [Lab 09 - Minimal Agent Loop](#) construye state, decisions, control del loop y stop reasons.
2. [Lab 10 - Tool Registry and Policy Gate](#) agrega tools, policy decisions, resultados que requieren aprobación y rutas de rechazo.
3. [Lab 11 - Context, Memory, Trace, and Evals](#) agrega context packets, memory con alcance, trace events y trajectory evals.

Haz los labs si quieres intuición sobre la implementación. Lee solo este capítulo si solo necesitas el modelo mental.

Regla de Diseño

Construye el runtime pequeño para aprender. Lanza con runtime capabilities maduras cuando el sistema deba sobrevivir usuarios reales, datos reales, efectos secundarios reales e incidentes reales.

Capítulos Relacionados

- [What Is An Agent?](#)
- [Agent Loop](#)
- [Agent Harnesses](#)
- [Tool Capability Design](#)
- [Context Engineering](#)
- [Observability and Evals](#)

Agent Engineer Toolkit

Un agent engineer elige models, tools, memory, orchestration, evals e infraestructura de deployment como un solo sistema. El toolkit debe apoyar la arquitectura, no definirla.

Usa este capítulo para decidir qué construir directamente y qué tomar de un framework.

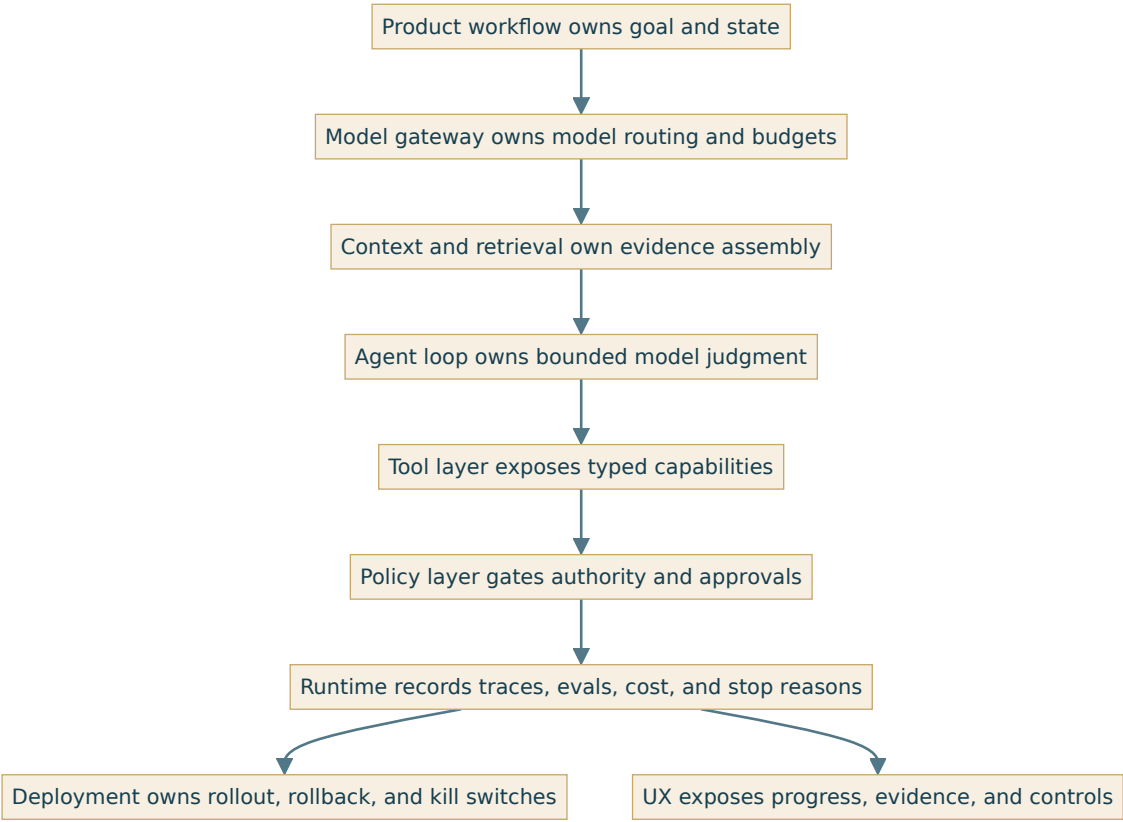
Piensa en el toolkit como un mapa de capabilities. Algunas capabilities pueden venir de un framework, otras de servicios de plataforma existentes, y otras deben permanecer bajo control del producto porque definen autoridad, policy y confianza.

Toolkit Layers

Layer	Responsabilidad	Ejemplos de Decisiones
Model layer	Razonamiento, generación, llamadas a tools, structured output.	Calidad del model, latencia, costo, context window, modo de deployment.
Model gateway	Routing, fallback, límites de tasa, abstracción de proveedores.	Qué models están permitidos, cuándo ocurre fallback, cómo se fijan versiones.
Context layer	Ensambla el conjunto de trabajo del model.	Forma del context packet, etiquetas de fuente, exclusiones, presupuesto de tokens.
Orchestration layer	Workflow, routing, loops, state, reintentos.	Workflow en código, grafo estilo LangGraph, crew estilo CrewAI, runtime estilo Mastra.
Tool layer	Acciones externas y acceso a datos.	Servidores MCP, APIs internas, browser tools, ejecución de código, funciones de base de datos.
Policy layer	Decisiones de allow, deny, aprobación, escalamiento y auditoría en runtime.	Autoridad de tool, acceso a datos, escrituras en memory, umbrales de aprobación.
Memory layer	Working, episodic, semantic y user memory.	Índice vectorial, almacén relacional, filesystem state, memory write policy.
Retrieval layer	Búsqueda de fuentes, filtrado, reranking, citación.	Tipo de índice, registro de fuentes, reglas de frescura, filtros de tenant.
Evaluation layer	Chequeos de calidad y regresión.	Golden datasets, judges, etiquetas humanas, production traces.
Observability layer	Depuración y operaciones.	Traces, métricas, costos, logs, run replay.
Deployment layer	Hosting en runtime, escalado, configuración, rollback.	Feature flags, canaries, kill switches, environment policy.

Layer	Responsabilidad	Ejemplos de Decisiones
Security layer	Permisos y contención.	Sandboxes, secrets, policy gates, auditoría, controles de egreso.
UX layer	Interacción humana y confianza.	Streaming, aprobaciones, explicaciones, handoffs, recuperación.
Governance layer	Cómo cambia el sistema.	ADRs, revisiones, release gates, incident-to-eval loop.

Las layers pueden venir de un solo framework, varias librerías o código directo. Lo importante es que cada layer tenga un responsable.



Usa el mapa como verificación de ownership. Si un toolkit oculta uno de estos límites, el equipo aún es responsable de la decisión faltante; solo se ha movido fuera de la vista.

Build, Buy, Or Compose

No preguntes primero “¿qué framework deberíamos usar?”. Pregunta qué capabilities del toolkit deben ser propiedad del producto.

Capability	Frecuentemente provisto por framework	Debe permanecer bajo control del producto
Model adapters	Cientes de proveedor, streaming, ayudantes de structured output.	Models permitidos, fallback policy, presupuestos de costo.

Capability	Frecuentemente provisto por framework	Debe permanecer bajo control del producto
Orchestration	Ejecución de grafos, pasos de workflow, reintentos.	State schema, razones de detención, esperas de aprobación, rollback.
Tooling	Registro de tools, ayudantes de invocación, clientes MCP.	Autoridad de tool, efectos secundarios, alcances, credenciales, aprobaciones.
Memory	Almacenes, embeddings, ayudantes de retrieval.	Write policy, consentimiento, retención, eliminación, corrección.
Retrieval	Índices, rerankers, conectores.	Registro de fuentes, filtros de acceso, frescura, reglas de citación.
Evals	Runners, judges, reportes.	Casos bloqueantes, fixtures de incidentes, release gates.
Observability	Spans, logs, dashboards.	Trace schema, redacción, requisitos de auditoría, replay policy.
Deployment	Hosting, configuraciones, escalado.	Plan de rollout, kill switches, ownership de incidentes.

Los frameworks son buenos para la mecánica. Los equipos de producto aún deben ser dueños de la autoridad.

Minimum Toolkit By Risk

El toolkit debe escalar según el riesgo. Un demo y un agent de producción con efectos secundarios no necesitan la misma maquinaria.

Nivel de riesgo	Toolkit mínimo
Demo	Llamada al model, prompt, entradas de ejemplo, simple run log.
Asistente interno solo lectura	Context builder, filtros de fuente, evals básicos, trace IDs, feedback path.
Agent de producción solo lectura	State model, retrieval policy, eval gates, observabilidad, presupuestos de costo, controles de rollout.
Agent de producción con efectos secundarios	Tools tipados, policy engine, approval gates, idempotencia, rollback, incident replay.
Agent regulado o de alto riesgo	Revisión de seguridad, audit logs, redacción, aislamiento de tenant, escalamiento humano, ADRs, release gates formales.

Agregar layers al toolkit no es ceremonia cuando el agent puede afectar usuarios, dinero, acceso, datos o sistemas de producción.

Build vs Framework

Usa un framework cuando elimina complejidad operativa real:

- Ejecución de grafos;
- State durable;
- Tracing;
- Registro de tools;
- Integración de eval;
- Flujos de aprobación humana;
- Gestión de deployment;
- Orquestación multi-agent.

Construye directamente cuando:

- El workflow es pequeño;
- El equipo necesita control total sobre state y policy;
- Las abstracciones del framework ocultan fallas importantes;
- El sistema debe integrarse con infraestructura existente;
- El agent es parte de un workflow de producto mayor.

Los frameworks deben hacer los límites más claros. Si un framework dificulta ver el state, llamadas a tools, permisos o modos de falla, usa una abstracción más pequeña.

Product-Owned Interfaces

Mantén estas interfaces explícitas incluso cuando un framework las ejecute:

- Contratos de request y response;
- Schemas de state y eventos;
- Tool manifests;
- Forma del context packet;
- Policy decision schema;
- Memory record schema;
- Approval request y decision schema;
- Formato de eval fixture;
- Trace schema;
- Controles de deployment y rollback.

Estas son las piezas que permiten a un equipo migrar frameworks, depurar incidentes y mantener la autoridad fuera de los prompts.

Framework Evaluation Checklist

Antes de adoptar un framework, pruébalo contra los requisitos de producción del agent.

- ¿Puede persistir y reanudar el state?
- ¿Puede representar approval gates?
- ¿Puede restringir tools por rol o ruta?
- ¿Puede trazar llamadas a models, tools y handoffs?

- ¿Puede ejecutar evals con fixtures realistas?
- ¿Puede soportar fallback y routing de models?
- ¿Puede exponer costo y latencia por paso?
- ¿Puede ejecutarse en el entorno de deployment del equipo?
- ¿Pueden los ingenieros depurar fallas sin leer el código interno del framework?
- ¿Puede ser removido después si el producto lo supera?

Haz un vertical slice delgado antes de comprometer toda la arquitectura.

Model Selection

Elige models por carga de trabajo, no solo por ranking de benchmarks.

Evalúa:

- Fiabilidad de structured output;
- Fiabilidad en el uso de tools;
- Seguimiento de instrucciones;
- Latencia;
- Costo por task completado;
- Comportamiento del context window;
- Comportamiento de rechazo y seguridad;
- Necesidades multimodales;
- Soporte para restricciones de deployment;
- Estabilidad del proveedor y versionado.

Usa múltiples models cuando reduzca costo o mejore calidad. Un model pequeño puede hacer routing, clasificar o extraer. Un model más fuerte puede planear, sintetizar o resolver edge cases.

Tooling Decisions

Los tools requieren disciplina de ingeniería.

Al seleccionar el toolkit, busca las superficies mínimas que hagan los tools gobernables: nombres, descripciones, entradas y salidas tipadas, scopes de permisos, timeouts, idempotencia, etiquetas de efectos secundarios y test fixtures. La checklist de diseño completa está en [Tool Capability Design](#); este capítulo solo pregunta si tu toolkit puede soportarlo.

Los tools deficientes exponen superficies amplias como shell sin restricciones, browser control genérico o “database query” sin policy checks. Los tools amplios pueden ser útiles para coding agents confiables, pero necesitan sandboxing y auditoría.

La tool layer debe ser aburrida a propósito. Los tools deben parecerse a capabilities de producto, no a infraestructura cruda. Prefiere `create_refund_draft` sobre `run_sql`, `send_approved_message` sobre `post_http` y `lookup_order_summary` sobre acceso amplio a bases de datos.

Memory Decisions

Memory no es una sola función. Son varios almacenes con reglas diferentes.

- Working memory guarda el state actual del task.
- Episodic memory almacena ejecuciones previas e interacciones de usuario.
- Semantic memory almacena hechos y documentos.
- Procedural memory almacena habilidades, playbooks y preferencias.

Decide quién puede escribir en memory, cómo se validan las escrituras, cómo expira la memory antigua y cómo se cita la retrieval.

Decisiones de Evaluación y Observabilidad

Evals y traces son capas del toolkit, no detalles finales.

Como mínimo, el toolkit debe soportar:

- evals offline con tools simulados;
- trajectory evals que inspeccionan llamadas a tools y decisiones de policy;
- regression evals a partir de incidentes;
- seguimiento de versiones de model, prompt, tool, policy y memory;
- traces a través de llamadas a model, llamadas a tool, retrieval, aprobaciones y escrituras en memory;
- reglas de redacción para logs y fixtures de eval;
- replay sin repetir efectos secundarios inseguros.

Si el toolkit no puede probar lo que sucedió, el equipo depurará la respuesta final en vez del sistema.

Lista de Verificación para el Ensamblaje del Toolkit

Antes de elegir librerías o frameworks, completa esto:

Pregunta	Respuesta Necesaria
¿Quién posee el goal activo y el state?	workflow, runtime, app service o framework.
¿Quién posee la autoridad de tool?	tool registry, policy engine, access-control service.
¿Quién posee las escrituras en memory?	memory service, policy gate, revisión humana.
¿Quién posee el ensamblaje de context?	context builder con paquetes trazables.
¿Quién posee los eval gates?	CI, proceso de release, eval service.
¿Quién posee la observabilidad?	tracing platform, app logs, audit store.
¿Quién posee el rollback?	deployment system, feature flags, tool disablement.
¿Quién posee los controles de usuario?	product UI, approval service, support workflow.

Las capas sin dueño se convierten en incidentes en producción.

Anti-Patrones del Toolkit

- Elegir un framework antes de elegir un pattern.
- Darle a cada agent todas las tools.
- Tratar la búsqueda vectorial como memory sin policy de escritura.
- Registrar solo respuestas finales y no decisiones de tools.
- Usar model routing sin evals para la calidad de las rutas.
- Lanzar un demo framework path sin state, policy y observability de producción.
- Tratar la memory proporcionada por el framework como segura por defecto.
- Dejar que la policy viva solo en el texto del prompt.
- Agregar observability después del primer incidente en producción.
- Construir una capa de tool sin manifest, dueño o ruta de desactivación.
- Elegir un model gateway sin version pinning ni reglas de fallback.

Capítulos Relacionados

- [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

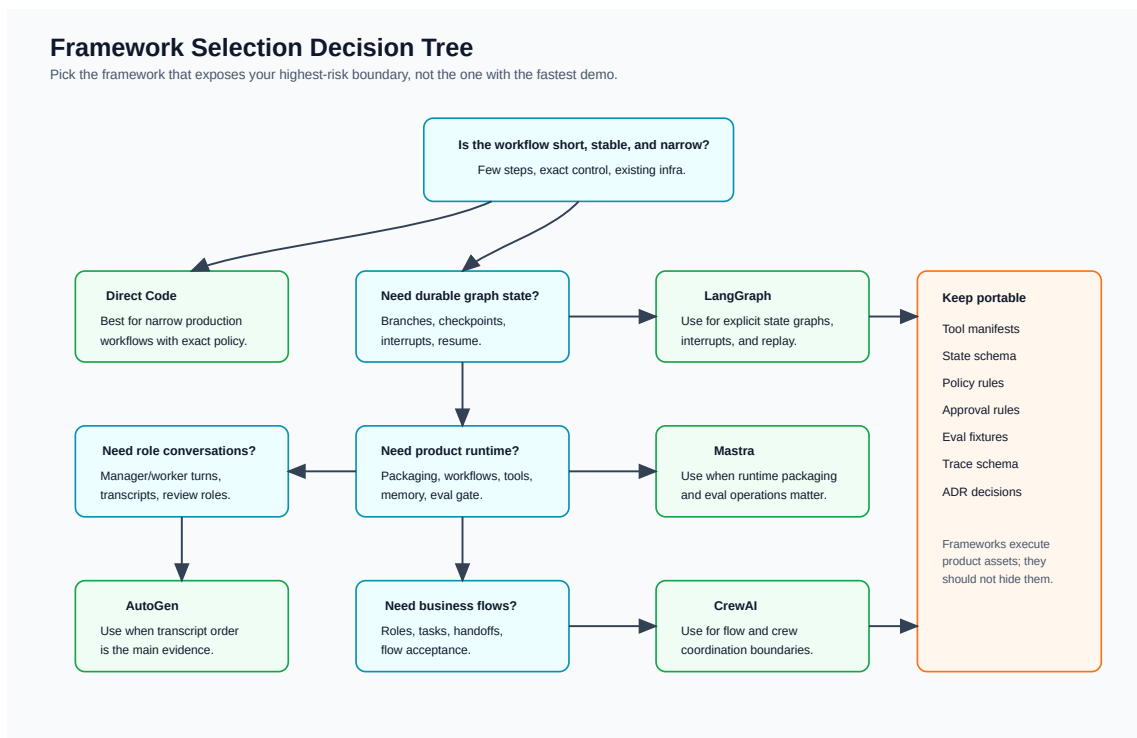
Los agent frameworks pueden acelerar el desarrollo, pero también definen los límites del sistema. Selecciona un framework después de conocer el pattern, el state model, las tools, las necesidades de eval y las restricciones de deployment.

Usa este capítulo para comparar frameworks sin convertir la arquitectura en una demostración del framework.

Para una comparación lado a lado de LangGraph, sistemas estilo AutoGen, Mastra AI, CrewAI y un mini-runtime personalizado, usa [Cross-Framework Decision Matrix](#). Para notas concretas de instalación y portabilidad, usa [Real Framework Setup Notes](#). Para el registro de decisiones, usa [Templates and Worksheets](#).

Decision Tree

Usa este diagrama para mantener la conversación sobre frameworks enfocada en la forma de la carga de trabajo. La pregunta útil es qué hace más fácil inspeccionar el framework: el state directo, la ejecución de graphs, el runtime packaging, los transcripts de roles o la coordinación del business-flow.



Intent

Elige si construir directamente, usar una librería ligera o adoptar un agent framework completo.

El framework correcto debe hacer más visibles las partes importantes: state, tools, policy, traces, handoffs y modos de falla.

El framework incorrecto hace la demo más rápida y el sistema más difícil de mantener. Oculta el state, convierte las llamadas a tools en magia, hace que la memory parezca automática y le da al equipo un vocabulario antes de tener una arquitectura.

Selection Criteria

Criterio	Preguntas
Control flow	¿Soporta chains, graphs, loops, routing y human gates?
State	¿Puede persistir, reanudar, inspeccionar y migrar el state?
Tooling	¿Las tools pueden ser tipadas, acotadas, testeadas y auditadas?
Multi-agent	¿Puede representar roles, handoffs y ownership claramente?
Memory	¿Separa working, episodic, semantic y procedural memory?
Evaluation	¿Pueden los evals ejecutarse sobre routes, tools, traces y outputs finales?
Observability	¿Expone llamadas a model, tool, costos, latencia y errores?
Security	¿Pueden las tools y agents ejecutarse con el menor privilegio posible?
Deployment	¿Puede ejecutarse donde tu sistema lo requiere?
Escape hatch	¿Puedes bajar a código cuando la abstracción es incorrecta?
Portability	¿Pueden prompts, tools, evals, policies y state sobrevivir un cambio de framework?
Ownership	¿Tu equipo puede depurar fallas en producción sin esperar por los internals del framework?

No adoptes un framework que oculte una preocupación que tu producto debe controlar.

Fit By Problem Shape

Empieza desde la carga de trabajo, no desde la categoría de framework.

Forma del problema	Mejor opción	Precauciones
Single bounded task	código directo o prompt library.	sobreconstruir state y orquestación.
Known phases	prompt chain o workflow.	ocultar validación entre pasos.
Branching workflow	graph o durable workflow framework.	proliferación de graphs y ownership de state poco claro.
Tool-heavy agent	tool registry más policy boundary.	tools demasiado amplias y controles de aprobación débiles.

Forma del problema	Mejor opción	Precauciones
Evidence-heavy answers	RAG framework más context builder.	retrieval desconectado de policy y evals.
Long-running work	durable workflow runtime.	loops en memoria y pérdida de approval state.
Multi-role work	multi-agent framework o topología de servicios.	especialización falsa con context duplicado.
Cross-agent communication	A2A, contratos de servicio, colas o workflows.	handoffs en lenguaje natural sin schema.
Product runtime	application/runtime framework.	lock-in de plataforma y operaciones opacas.
Coding agents	harness aislado con controles de archivos, comandos, tests y aprobación.	acceso a shell sin policy o replay.

Si dos enfoques funcionan, elige el que haga más fácil inspeccionar state, tools, policy, evals y rollback.

Common Framework Shapes

Forma	Fortaleza	Riesgo
Prompt and tool library	Rápido para agents simples.	Debes construir state, evals y operaciones tú mismo.
Graph/workflow framework	Rutas de ejecución claras, state durable, edges condicionales.	Los graphs pueden volverse difíciles de leer si cada rama es un nodo.
Multi-agent framework	Primitivas de roles y delegación.	Fácil sobreutilizar agents donde un workflow es suficiente.
RAG framework	Retrieval, índices y conectores de datos.	Retrieval puede desconectarse de policy y evals.
Runtime framework	Deployment, tracing, memory, workflows y operaciones.	Las decisiones de runtime pueden llevar a lock-in de plataforma.

La mejor opción puede ser combinar pocas capas en vez de un solo framework grande.

Capability Matrix

Usa esta matriz durante la evaluación. El objetivo no es asignar puntajes por características de marketing. El objetivo es identificar qué responsabilidades asume el framework y cuáles debe seguir asumiendo tu aplicación.

Capability	Preguntas a realizar
State model	¿Dónde se almacena el state? ¿Puede inspeccionarse, migrarse, reproducirse y versionarse?
Tool boundary	¿Las tools son tipadas? ¿Scopes, efectos secundarios, aprobaciones y trace fields son explícitos?
Policy	¿Puede la policy ejecutarse antes de llamadas a tools, escrituras de memory, retrieval y respuestas finales?
Memory	¿Se separan working, episodic, semantic y user memory? ¿Se pueden revisar las escrituras?
Context	¿Puede el framework mostrar qué context packet se armó para cada llamada a model?
Human approval	¿Las aprobaciones son durables, ligadas a acciones exactas, expiran y son auditables?
Evals	¿Pueden los evals probar trayectorias, llamadas a tools, rechazos, escrituras de memory y citas?
Observability	¿Se trazan llamadas a model, tool, costos, latencia, decisiones de policy y reintentos?
Security	¿Puede hacer cumplir least privilege, sandboxing, alcance de credenciales y límites de salida?
Deployment	¿Soporta tu runtime, residencia de datos, escalabilidad, rollback y proceso de incidentes?
Interop	¿Puede exponer o consumir MCP, A2A, REST, colas o tus contratos de servicio existentes?
Exit path	¿Puedes mover prompts, tools, state, policies y evals después?

Build Directly When

- El workflow es corto y estable.
- Necesitas control exacto sobre state y policy.
- El equipo ya tiene infraestructura de workflow.
- El agent está embebido en un producto existente.
- La capacidad de depuración importa más que el prototipado rápido.
- El framework introduciría más conceptos de los que el dominio requiere.

El código directo suele ser la mejor primera versión productiva para un workflow acotado.

Build directly no significa construir sin cuidado. Significa usar límites normales de software: inputs tipados, tablas de state, clientes de tools, chequeos de policy, tests, traces y controles de deployment. Una implementación directa pequeña con límites fuertes es mejor que una implementación en framework que los oculte.

Use a Framework When

- El sistema necesita ejecución de graphs durables.
- Los agents requieren registro y descubrimiento de tools.
- Los handoffs multi-agent son centrales para el producto.
- Necesitas tracing, evals o soporte de deployment integrados.
- Vas a construir muchos agents relacionados.
- El equipo se beneficia de convenciones compartidas.

Usa un framework para hacer consistentes los patterns repetidos, no para evitar entender la arquitectura.

La adopción de frameworks es más fuerte cuando el equipo ya conoce las responsabilidades que necesita que el framework asuma. Es más débil cuando el equipo dice “necesitamos agents” y deja que el framework defina la arquitectura después.

Evaluation Process

Prueba un framework con un vertical slice:

1. una solicitud real de usuario;
2. una ruta o workflow;
3. una tool de lectura;
4. una tool de escritura con aprobación;
5. una lectura o escritura de memory;
6. un camino de falla;
7. un caso de eval;
8. una inspección de trace;
9. un camino de deployment.

Si el slice es difícil de inspeccionar, probar o asegurar, detente antes de que el framework se expanda.

Agrega un segundo vertical slice para fallas:

1. solicitud de usuario mal formada;
2. evidencia faltante;
3. rechazo de policy check;
4. timeout de tool;
5. espera de aprobación;
6. cancelación;
7. replay desde trace;
8. rollback de prompt, model o configuración de tool.

La mayoría de los frameworks se ven bien en el camino feliz. El slice de fallas te dice si puedes operarlo.

Portability And Exit

Asume que el framework puede cambiar. La forma más segura de adoptar uno es mantener los activos principales portables:

- tool manifests;
- archivos de prompt e instrucciones;
- state schemas;
- memory schemas;
- reglas de aprobación;
- reglas de policy;
- fixtures de eval;
- trace schema;
- context packet shape;
- architecture decision records.

Evita enterrar estos activos dentro de callbacks o decoradores específicos del framework si representan policy del producto. El framework puede ejecutarlos. No debe ser el único lugar donde existen.

Anti-Patrones de Framework

- Seleccionar un framework porque soporta todos los patterns.
- Reemplazar decisiones de arquitectura con valores predeterminados del framework.
- Dar a todos los agents del framework el mismo context y tools.
- Aceptar trazes opacos.
- Saltarse evals porque el demo del framework funcionó.
- Tratar la memory del framework como confiable por defecto.
- Usar un framework multi-agent para simple routing.
- Tratar la observability incorporada como suficiente sin incident replay.
- Aceptar abstracciones de tools del framework que no pueden expresar side effects o approvals.
- Permitir que los valores predeterminados del framework decidan la retención de memory, el ensamblaje de context o el alcance de seguridad.

Lista de Verificación para Decisiones

Antes de adoptar un framework, responde:

- ¿Qué pattern estamos implementando?
- ¿Qué responsabilidades asume el framework?
- ¿Qué responsabilidades permanecen en nuestra aplicación?
- ¿Podemos inspeccionar state, context, llamadas a tools, memory, decisiones de policy y trazes?
- ¿Podemos ejecutar evals sin llamar tools reales con side effects?
- ¿Podemos hacer cumplir approval y least privilege fuera del prompt?
- ¿Podemos desplegar, hacer rollback y deshabilitar una capability rápidamente?
- ¿Podemos migrar sin reescribir prompts, tools, state, evals y policy?
- ¿Qué ADR registra esta elección?
- ¿Qué falla nos haría reconsiderar el framework?

Capítulos Relacionados

- [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

Este capítulo compara LangGraph, sistemas estilo AutoGen, Mastra AI, CrewAI y un runtime personalizado pequeño según la responsabilidad de ingeniería. El objetivo no es declarar un framework como el mejor. El objetivo es decidir qué debe ser responsable de state, tools, policy, memory, evals, deployment y recuperación ante fallas para un sistema específico.

Usa este capítulo después de los laboratorios. Los laboratorios muestran la forma de cada framework de manera aislada. Este capítulo ayuda a elegir entre ellos para un producto real. Úsalo junto con [Real Framework Setup Notes](#) cuando necesites orientación concreta de instalación y portabilidad.

Utiliza este diagrama para ubicar al propietario del state antes de comparar APIs. Un framework es mejor opción cuando su límite de state coincide con la falla que más necesitas inspeccionar y recuperar.



Decision Rule

Elige el framework que haga más fácil inspeccionar, probar y operar el límite de mayor riesgo.

Si el mayor riesgo es un state reanudable, prefiere el state de graph o workflow. Si el mayor riesgo es la responsabilidad multi-agent, prefiere un transcript o flow que registre el comportamiento de los roles. Si el mayor riesgo es el empaquetado del runtime de producción, prefiere un runtime con convenciones para tools, memory, evals y observability. Si el mayor riesgo es el control exacto de policy, un runtime pequeño y directo puede ser más seguro que una abstracción grande.

Framework Fit Matrix

Opción	Mejor cuando	Evitar cuando	Mantener portable
LangGraph-style state graph	El trabajo tiene ramificaciones, checkpoints, interrupciones, reanudación u observabilidad a nivel de nodo.	Una secuencia simple o una sola llamada a tool es suficiente. La forma de graph ocultaría el state en vez de aclararlo.	State schema, contratos de nodos, registros de checkpoints, payloads de interrupciones, fixtures de eval.
AutoGen-style team	La colaboración, el comportamiento de roles y la revisión de transcripts son centrales.	El equipo es solo una cadena con nombres de agent. El historial de conversación se convierte en el único almacén de state.	Message schema, contratos de roles, reglas de terminación, evals de transcript, tool policy.
Mastra-style runtime	El producto en TypeScript necesita agents, tools, workflows, memory, evals y	Las convenciones del runtime ocultarían la policy del producto o los requisitos de deployment.	Tool manifests, contratos de workflow, reglas de memory, evals, trace schema.

Opción	Mejor cuando	Evitar cuando	Mantener portable
	observability empaquetados juntos.		
CrewAI-style flows and crews	Los workflows en Python necesitan state propio del flow más crews de especialistas acotados.	Los roles se superponen, la aceptación del flow es vaga o los crews reemplazan el diseño determinista del workflow.	Flow state, contratos de task, permisos de roles, outputs de crews, evals de aceptación.
Mini-runtime/custom code	El alcance es reducido, la policy requiere control exacto o el equipo ya tiene infraestructura de workflow.	Necesitas durabilidad en producción, escalabilidad, observability alojada e integraciones con el ecosistema de inmediato.	Todo: state, policy, tools, memory, evals, traces, contratos de deployment.

Responsibility Matrix

Responsabilidad	LangGraph-style	AutoGen-style	Mastra-style	CrewAI-style	Mini-runtime
State owner	State de graph y checkpoints.	State de equipo/task fuera del transcript.	State de workflow y memory en el runtime.	State del flow.	El model de tu aplicación.
Control flow	Nodos, edges, transiciones condicionales, interrupciones.	Policy de turnos del equipo y terminación.	Workflows y agent runtime.	Flows coordinan crews y tasks.	Loop, router o código de workflow que escribas.
Tool policy	Nodos de guardia, middleware o wrappers de tools.	Límite de ejecución del manager/runtime.	Hooks de policy de tools y workflows.	Tools de roles más restricciones del flow.	Policy gate explícito.
Memory	State más almacenes de memory.	Transcript más memory externa.	Abstracciones de memory en el runtime.	Context de flow/crew y almacenes externos.	Context packet y policy de memory.
Evals	Paths de nodos, diferencias de state, checkpoints, output final.	Turns del transcript, roles, llamadas a tools, terminación.	Traces del runtime, llamadas a tools, resultados de workflows.	Aceptación del flow y outputs de roles.	Pruebas de trayectoria que definas.

Responsabilidad	LangGraph-style	AutoGen-style	Mastra-style	CrewAI-style	Mini-runtime
Observability	Traces por nodo e inspección de checkpoints.	Transcript de mensajes estructurados.	Observability del runtime y hooks de eval.	Registros de flow/task/role.	Trace schema y almacenamiento que construyas.
Deployment	App runtime más checkpointer y almacenes.	App runtime más límites de servicio de agent.	Empaquetado de runtime en TypeScript.	App de flow en Python y workers.	Infraestructura de producto existente.
Escape hatch	Los nodos son código plano si el state se mantiene explícito.	El transcript puede exportarse y reproducirse.	Mantén tools/evals fuera del código exclusivo del framework.	Mantén el state del flow separado del chat de roles.	Máximo control, máxima carga operativa.

Choose By Risk

Riesgo dominante	Mejor opción por defecto	Por qué
Progreso perdido o workflows interrumpidos	LangGraph-style o durable workflow	Checkpoints, reanudabilidad y transiciones de state son de primera clase.
Comportamiento multi-agent sin responsabilidad	AutoGen-style transcript o CrewAI-style flow	Puedes inspeccionar turns de roles, handoffs y aceptación final.
Consistencia de runtime de producto en TypeScript	Mastra-style runtime	Agents, tools, workflows, memory y evals comparten una sola estructura de aplicación.
Automatización de workflow basada en roles en Python	CrewAI-style flow	El state del flow y la ejecución de crews se adaptan bien a equipos de automatización en Python.
Policy estricta y alcance mínimo	Mini-runtime/custom code	La aplicación controla el límite exacto y evita defaults del framework.
Incertidumbre sobre el framework	Mini-runtime primero, luego migrar	Construir primitivas una vez aclara lo que el framework debe proveer.

Migration And Escape Hatches

No permitas que el framework sea el único lugar donde exista la lógica del producto. Mantén estos activos portables:

- state schemas;
- tool manifests y clases de side-effect;
- reglas de policy;
- registros de aprobación;
- reglas de retención y eliminación de memory;
- archivos de prompt e instrucciones;
- forma del context packet;
- trace schema;
- fixtures de eval;
- notas de deployment y rollback;
- ADRs explicando por qué se eligió el framework.

La portabilidad no es teórica. Las APIs de los frameworks cambian, los equipos cambian de stack y los sistemas en producción superan las suposiciones iniciales. La adopción más segura de un framework mantiene los contratos del producto fuera de decoradores y callbacks específicos del framework siempre que sea posible.

Decision Checklist

Antes de elegir un framework, responde:

- ¿Quién es responsable del durable state?
- ¿Dónde corre la policy antes de los side effects?
- ¿Cómo se tipifican, delimitan, aprueban y trazan los tools?
- ¿Cuál es el trace reproducible más pequeño que explica una falla?
- ¿Qué evals fallan la build?
- ¿Qué pasa si una ejecución se interrumpe?
- ¿Cómo hacemos rollback de prompts, tools, policies y elecciones de model?
- ¿Qué datos nunca deben entrar en memory o traces?
- ¿Qué partes pueden migrar si el framework cambia?
- ¿Qué incidente de producción probaría que este framework fue la elección equivocada?

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](#)

Notas sobre la configuración de frameworks reales

Los laboratorios enseñan arquitectura a través de pequeñas implementaciones locales. Este capítulo muestra cómo traducir esos patterns a proyectos reales con frameworks, sin hacer que el libro dependa de un solo lenguaje o proveedor.

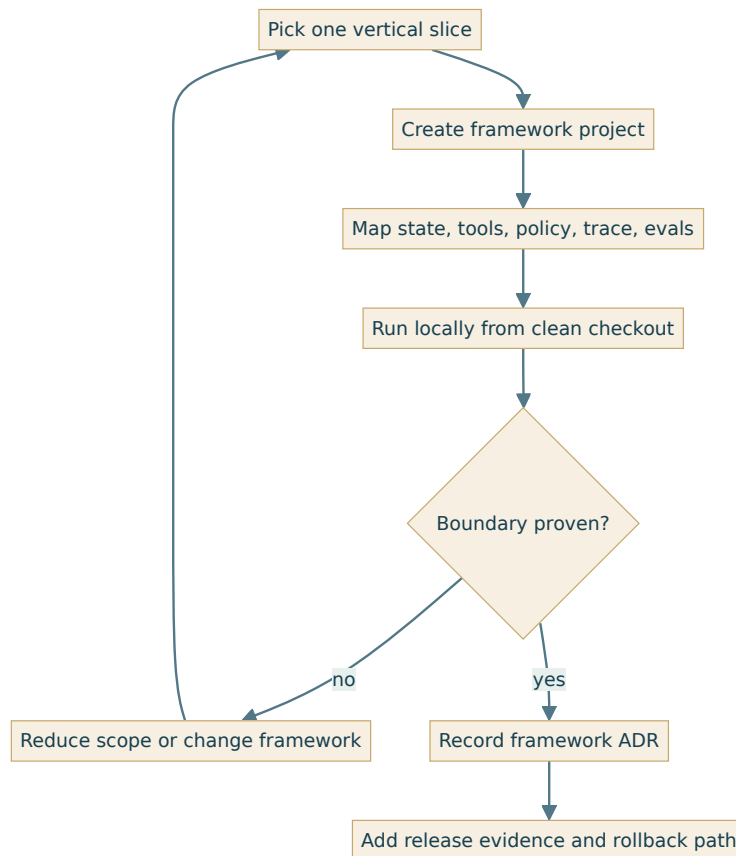
Usa estas notas después de los laboratorios neutrales al framework. Mantén los mismos contratos arquitectónicos: state schema, tool manifest, policy gate, trace schema, eval fixtures y rollback plan. El framework debe alojar esos contratos, no reemplazarlos.

Cómo usar este capítulo

Comienza con una vertical slice:

1. una solicitud de usuario;
2. un objeto de state;
3. una tool de solo lectura;
4. una tool de efectos secundarios detrás de policy o aprobación;
5. un trace;
6. un caso de eval;
7. un comando local de ejecución;
8. una ruta de rollback o deshabilitación.

No comiences portando todos los laboratorios. Porta el comportamiento más pequeño que demuestre que el framework puede asumir la responsabilidad que necesitas que asuma.



Usa este flujo antes de expandir el port. Un framework es candidato solo después de que demuestre el boundary que más importa: state, approval, tool authority, transcript ownership, workflow packaging o eval integration.

Reglas compartidas de configuración

Cada variante de framework debe definir estos archivos antes del primer demo:

Artifact	Propósito
<code>.env.example</code>	Nombra las keys requeridas sin almacenar secretos.
<code>state.schema.*</code>	Define state reanudable y campos de migración.
<code>tools.*</code>	Declara tool names, inputs, outputs, side effects y permisos.
<code>policy.*</code>	Hace cumplir reglas de acción antes de retrieval, memory writes, tool calls o respuestas finales.
<code>trace.*</code>	Emite eventos de run, model, tool, policy, approval y evaluator.
<code>evals/*</code>	Almacena happy path, negative path y trajectory checks.
<code>README.md</code>	Muestra comandos de install, local run, test, eval y cleanup.

Mantén los secretos fuera de los ejemplos. Usa `.env.example` para nombres y variables de entorno locales para valores.

Playbook de fallas de configuración

Los ports de frameworks fallan de maneras repetibles. Trata las fallas de configuración como evidencia de diseño, no como una molestia local. Un framework no está listo para el camino de producción del libro hasta que un nuevo ingeniero pueda reproducir install, run, test, eval y cleanup desde un checkout limpio.

Failure	Señal común	Qué capturar	Solución o decisión
Wrong runtime version	Python, Node.js o el package manager rechaza la instalación.	Runtime version, lockfile version, comando de instalación, paquete fallido.	Fija el runtime en docs y CI, o elige un slice de framework con versiones soportadas.
Hidden provider dependency	El demo importa un paquete de provider que no fue instalado.	Módulo faltante, nombre del paquete provider, model route, campo <code>.env.example</code> .	Haz explícitos los paquetes provider y mantén pruebas determinísticas de fallback.
Secret required for local test	La prueba base falla antes de que se pueda inspeccionar la arquitectura.	Comando, variable de entorno faltante, si existe fallback determinístico.	Separa pruebas unit/eval de pruebas smoke con provider en vivo.
Generated scaffold hides policy	Las reglas de tool, approval o memory viven solo dentro del prompt de ejemplo.	Ruta del archivo, regla oculta, módulo de policy faltante, prueba de boundary fallida.	Mueve policy al código y agrega un trajectory eval.
State cannot resume	Interrupción, retry o reinicio del servidor local pierde el run state.	Thread ID, checkpoint store, comando de resume, diferencia de state esperada.	Agrega checkpointer persistente o rechaza el framework para durable workflows.
Trace is final-answer only	Los logs muestran output pero no route, tool, policy, approval o stop reason.	Trace fields presentes y faltantes, sample run ID, requisito de eval.	Agrega instrumentación antes de expandir el port.
Upgrade breaks behavior	La actualización de un paquete cambia routing, tool calls, memory o termination.	Versiones vieja/nueva, trace cambiado, fixture fallido, comando de rollback.	Requiere evals de regresión antes de actualizar el framework.

Captura la falla en el framework ADR. La pregunta útil no es “¿Podemos instalarlo?” La pregunta útil es “¿Qué boundary de producción demostró o no demostró la configuración?”

Usa este registro de evidencia de configuración para cada framework slice:

Field	Valor de ejemplo
Framework slice	<code>native-framework-examples/langgraph-refund/</code>

Field	Valor de ejemplo
Runtime versions	<code>python --version</code> , <code>node --version</code> o versión del package manager.
Install command	Comando exacto desde un checkout limpio.
Baseline run command	El comando más pequeño que demuestre el boundary principal.
Eval command	Comando acotado que demuestre el comportamiento de falla.
Required secrets	Solo nombres; sin valores.
Deterministic fallback	Sí/no y comando.
Known setup failure	Una falla capturada y su solución.
Rollback o ruta de deshabilitación	Comando, feature flag o eliminación de ruta.

Variante LangGraph

Usa LangGraph cuando el principal riesgo es el control de flujo con estado: branching, checkpoints, interrupts, replay o esperas de aprobación humana.

Ejemplos nativos del repositorio:

- `native-framework-examples/langgraph-refund/` para approval, interrupt y comportamiento de resume.
- `native-framework-examples/langgraph-research-rag/` para source filtering, citation faithfulness y comportamiento de escalation.

Referencias oficiales de configuración:

- [LangGraph install](#)
- [LangGraph graph API](#)
- [LangGraph local server](#)
- [LangGraph persistence](#)
- [LangGraph interrupts](#)

Configuración local típica:

```
python3 -m venv .venv
source .venv/bin/activate
pip install -U langgraph
```

Para un servidor local de LangGraph, la documentación oficial usa el LangGraph CLI:

```
pip install -U "langgraph-cli[inmem]"
langgraph dev
```

Usa el servidor in-memory solo para desarrollo local. Producción necesita almacenamiento persistente para checkpoints y cualquier almacén de largo plazo.

Ruta de port desde los laboratorios:

Lab Asset	LangGraph Mapping
state object	state schema <code>StateGraph</code>
loop step	node function
route decision	conditional edge
approval wait	interrupt más checkpoint
trace event	spans de node, model, tool, policy e interrupt
eval case	graph input, expected state diff, expected route, expected stop reason

Preguntas de producción:

- ¿Qué checkpointer almacena el graph state con alcance de thread?
- ¿Cómo se asignan los thread IDs y se protegen contra acceso cruzado entre tenants?
- ¿Qué nodes pueden causar side effects?
- ¿Los nodes de side-effect son idempotentes bajo retry o resume?
- ¿Qué campos de state necesitan migraciones?
- ¿Qué interrupts requieren registros de aprobación humana?

Criterio de aceptación de ejemplo nativo: el ejemplo debe demostrar el boundary de mayor riesgo. Para workflows de reembolso, debe pausar en un approval interrupt, reanudar por thread ID, preservar el state previo y pasar evals sin emitir dinero. Para Research RAG, debe omitir fuentes obsoletas y prohibidas antes de la síntesis de la respuesta, citar solo evidencia actual aprobada y escalar cuando falte evidencia aprobada.

Variante AutoGen

Usa AutoGen cuando el principal riesgo es el comportamiento colaborativo: contratos de roles, historial de mensajes, terminación del equipo y revisión de transcripciones.

Ejemplo nativo del repositorio: `native-framework-examples/autogen-delivery/`.

Referencias oficiales de configuración:

- [AutoGen documentation](#)
- [AgentChat user guide](#)
- [AgentChat installation](#)
- [AgentChat agents](#)
- [AgentChat teams](#)
- [AgentChat termination](#)
- [AutoGen to Microsoft Agent Framework migration guide](#)

Configuración local típica:

```
python3 -m venv .venv
source .venv/bin/activate
```

```
pip install -U "autogen-agentchat" "autogen-ext[openai]"
```

AutoGen AgentChat actualmente requiere Python 3.10 o posterior. AutoGen es mantenido por la comunidad, así que los nuevos proyectos de larga duración en el stack de Microsoft también deberían evaluar Microsoft Agent Framework antes de estandarizar con AutoGen. Mantén los paquetes de proveedores explícitos para que las dependencias de model no queden ocultas dentro del framework.

Ruta de portabilidad desde los laboratorios:

Activo del laboratorio	Mapeo en AutoGen
supervisor	team manager o coordinator
worker contract	agent role, tools y expected output
transcript	durable message record
termination rule	team stop condition
tool policy	execution wrapper fuera del texto del mensaje
eval case	transcript replay con aserciones de role, tool y termination

Preguntas para producción:

- ¿Quién es dueño del transcript como durable state?
- ¿Qué mensajes se persisten, redactan y pueden reproducirse?
- ¿Qué detiene al equipo?
- ¿Qué agent puede llamar a qué tool?
- ¿Un retry puede duplicar un side effect de tool?
- ¿Qué fallas de transcript se convierten en fixtures de eval?

Criterio de aceptación del ejemplo nativo: el ejemplo debe definir AgentChat agents específicos por rol, una regla de terminación del equipo, una exportación normalizada de transcript y evals que fallen si faltan roles, el orden de turnos es inválido o falta el owner final.

Variante Mastra

Usa Mastra cuando el principal riesgo es empaquetar un product runtime en TypeScript alrededor de agents, workflows, tools, memory, observability y evals.

Ejemplo nativo del repositorio: [native-framework-examples/mastra-refund/](https://github.com/microsoft/autogen/tree/main/native-framework-examples/mastra-refund/).

Referencias oficiales de configuración:

- [Mastra docs](#)
- [Mastra quickstart](#)
- [Mastra manual install](#)
- [Mastra agents](#)
- [Mastra tools](#)
- [Mastra workflows](#)

- [Mastra evals](#)

La configuración local típica inicia desde el CLI del framework:

```
npm create mastra@latest
```

Usa el scaffold para inspeccionar la estructura del proyecto y luego mapea los contratos del libro en módulos administrados por el framework. No dejes la policy del producto solo dentro del código de ejemplo generado.

Ruta de portabilidad desde los laboratorios:

Activo del laboratorio	Mapeo en Mastra
agent decision	Mastra agent
deterministic control flow	workflow
tool registry	typed tool declarations
memory contract	framework memory más retention policy
trace schema	runtime observability export
eval case	framework eval más repository fixture

Preguntas para producción:

- ¿Qué partes son propiedad de Mastra y cuáles siguen siendo propiedad de la aplicación?
- ¿Dónde declaran los tools sus side effects y permisos?
- ¿Cómo se despliegan, reintentan y hacen rollback los workflows?
- ¿Cómo se exportan los traces y resultados de eval al sistema operativo del equipo?
- ¿Qué actualizaciones del framework requieren regression evals?

Criterio de aceptación del ejemplo nativo: el ejemplo debe definir un agent de borrador de reembolso, tools tipados de policy y borrador, un workflow que haga cumplir el orden policy-before-draft y un eval que falle en movimientos de dinero o mensajes al cliente.

Variante CrewAI

Usa CrewAI cuando el principal riesgo es la automatización de workflows en Python con state administrado por el flujo y crews de especialistas acotados.

Ejemplo nativo del repositorio: `native-framework-examples/crewai-delivery/`.

Referencias oficiales de configuración:

- [CrewAI docs](#)
- [CrewAI installation](#)
- [CrewAI quickstart](#)
- [CrewAI introduction](#)

La documentación actual de CrewAI enfatiza Python 3.10 a 3.13, instalación basada en `uv` y un quickstart que crea un Flow más un agent crew. Mantén el flow generado lo suficientemente pequeño para que las transiciones de state sigan siendo visibles.

Ruta de portabilidad desde los laboratorios:

Activo del laboratorio	Mapeo en CrewAI
flow state	CrewAI Flow state
task delegation	crew tasks
worker contract	agent role, goal, tools y output shape
merge policy	flow acceptance step
trace event	flow, task y crew output records
eval case	flow output más aserciones de role-output

Preguntas para producción:

- ¿Qué posee el Flow que el Crew no debe mutar implícitamente?
- ¿Qué aporta cada rol que una función determinista no podría?
- ¿Cómo se validan los outputs del crew antes de que el flow los acepte?
- ¿Qué sucede cuando un rol falla o no está de acuerdo?
- ¿Qué checkpoints de flow se necesitan antes de efectos secundarios externos?

Criterio de aceptación del ejemplo nativo: el ejemplo debe mantener separados los outputs de planner, reviewer y tester, y luego dejar que el Flow decida la aceptación final.

Variante Mini-Runtime

Usa el mini-runtime personalizado cuando el principal riesgo es comprender y poseer la arquitectura. No es un reemplazo de framework para todas las necesidades de producción. Es una herramienta de enseñanza y diseño.

Mapea los mismos contratos directamente:

Contrato	Ubicación en Mini-Runtime
state	objeto o tabla de aplicación explícita
tools	registro con schemas y etiquetas de side-effect
policy	función llamada antes de la autoridad
memory	context packet más almacenamiento gobernado
trace	lista de eventos append-only
evals	pruebas deterministas sobre state, trace y output

El mini-runtime es valioso porque hace visibles las responsabilidades ocultas del framework. Después de construirlo, los lectores pueden juzgar si LangGraph, AutoGen, Mastra o CrewAI agregan suficiente valor operativo para justificar la abstracción.

Lista de verificación de aceptación agnóstica al framework

Antes de considerar completo un port a framework, verifica:

- el comando de instalación está documentado y es reproducible;
- el comando de ejecución local funciona desde un checkout limpio;
- los secretos viven en variables de entorno, no en el código fuente;
- el owner del state está nombrado;
- los side effects de tools están declarados;
- la policy se ejecuta antes de la autoridad;
- los traces incluyen eventos de model, tool, policy y evaluator;
- los evals cubren happy path, negative path y trayectoria;
- rollback o kill switch está documentado;
- el código específico del framework no oculta los contratos del producto.

Si el port solo pasa el happy path, sigue siendo una demo.

Plantillas y Hojas de Trabajo

Estas plantillas convierten las recomendaciones del libro en artifacts de ingeniería revisables. Úsalas cuando un laboratorio se convierte en un slice de producto, cuando un equipo elige un framework o cuando un agent obtiene más autoridad.

Copia solo las secciones que importan. Un registro de decisión corto y completo es mejor que una plantilla larga con respuestas vacías.

Para ejemplos llenados, compara estas plantillas con los [Proyectos Capstone](#).

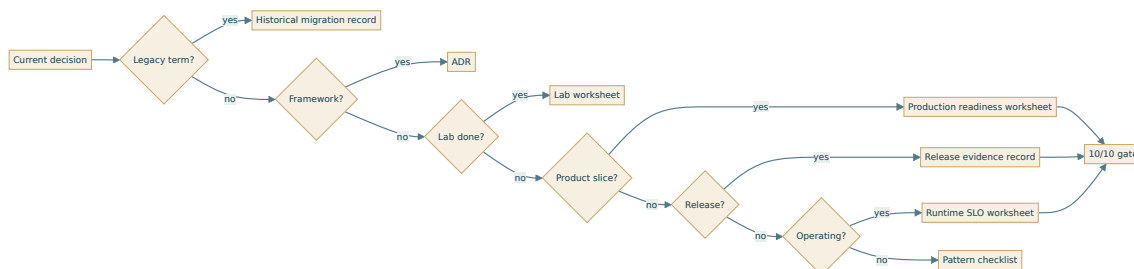
Elige el Artifact Correcto

Comienza con el artifact que corresponda a la decisión que tienes enfrente. No llenes todas las plantillas por defecto.

Situación	Usa este Artifact	Listo Cuando
Elegir LangGraph, AutoGen, Mastra, CrewAI o un runtime personalizado	Framework selection ADR template	El equipo puede nombrar qué controla el framework y qué sigue controlando la aplicación.
Estudiar ADRs de agent completados	Completed agent ADR examples	El equipo puede comparar su registro de decisión con ejemplos realistas de authority, RAG y multi-agent.
Estudiar evidencia de laboratorio completada	Completed lab evidence examples	El equipo puede comparar su hoja de trabajo con comandos concretos, traces, rutas de falla, brechas de producción y decisiones de lanzamiento.
Estudiar evidencia de preparación para producción	Completed production readiness examples	El equipo puede comparar su hoja de producción con responsables concretos, gates, bloqueadores, calificaciones de readiness y siguientes acciones.
Revisar una respuesta Agentic RAG	Agentic RAG query trace worksheet	El equipo puede reconstruir la selección de fuentes, evidencia omitida, verificación de citas y la decisión final de respuesta o rechazo.
Revisar un workflow de debate o consenso	Debate and consensus review checklist	Independencia, merge policy, manejo de disenso, presupuesto, comparación base y errores del juez son revisados.
Modelar amenazas en una ruta de agent	Agent threat model worksheet	Datos privados, contenido no confiable, autoridad de tool, riesgos STRIDE, evals y evidencia de trace se revisan en conjunto.

Situación	Usa este Artifact	Listo Cuando
Revisar UX y confianza de agent	Agent UX review worksheet	States en runtime, evidencia visible, controles de usuario, UX de aprobación, rutas de corrección y UX evals son explícitos.
Terminar un laboratorio	Lab completion worksheet	Se registran el comando de ejecución, comando de prueba, boundary del pattern y la lección.
Completar Lab 02 Agent Loop y Planeación	Lab 02 planning loop guided exercise worksheet	Se capturan el trace del plan base, input cambiado, paso no soportado, input faltante y contrato de condición de parada.
Completar Lab 03 Agentic RAG	Lab 03 Agentic RAG guided exercise worksheet	Se capturan retrieval base, cambio de grounding, comportamiento ante evidencia faltante, contrato de fuente, comparación con grafo nativo y fixture de eval.
Completar Lab 06 Observability y Evals	Lab 06 observability and evals guided exercise worksheet	Se capturan contrato de trace, falla por missing-policy, caso negativo, CI gate y nota de incidente a eval.
Completar Lab 07 Runtime Packaging	Lab 07 runtime packaging guided exercise worksheet	Se capturan límites de runtime, orden de tools, efectos secundarios prohibidos, rollback y comparación con Mastra nativo.
Completar Lab 12 State Graphs	Lab 12 state graph guided exercise worksheet	Se capturan interrupción, checkpoint, reanudación, seguridad de replay, comparación con LangGraph nativo y seguimiento de producción.
Convertir un laboratorio en un slice de producto	Lab production readiness worksheet	State faltante, policy, eval, trace, rollback y controles de ownership son explícitos.
Revisar un candidato a producción	Production readiness worksheet	Cada ruta de alta authority tiene evidencia o está fuera del alcance del release.
Operar una ruta en producción	Runtime SLO and incident review worksheet	SLOs, paneles de dashboard, triaje de incidentes, revisión de traces, conversión de eval y acciones de rollback son explícitas.
Preparar un release	Release evidence record	El release tiene output de eval, riesgo conocido, aprobador, responsable de rollback y comando de rollback.
Revisar la composición de un agentic system	Agentic system architecture review checklist	Experiencia, control, ejecución, conocimiento, seguridad, observability, evaluación, contratos y límites de authority son explícitos.
Revisar la arquitectura completa de un sistema	Reference architecture review checklist	Entrada, ruteo, state, conocimiento, tools, memory, aprobación, evals, observability y

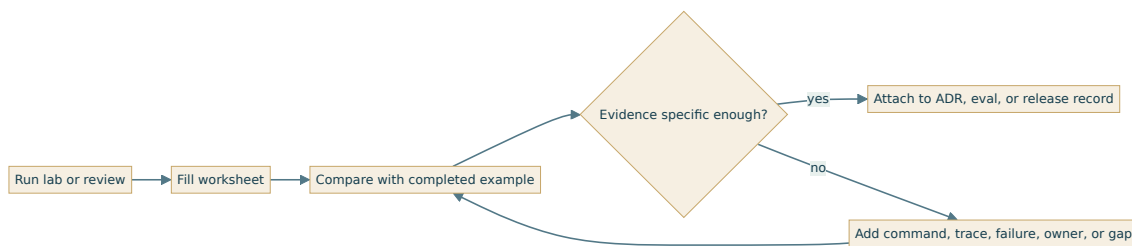
Situación	Usa este Artifact	Listo Cuando
		ownership del release se revisan en conjunto.
Revisar trabajo desatendido disparado por eventos	Event-triggered agent review checklist	Contrato de evento, admisión, dedupe, orden, reintentos, dead letters, tormentas, traces y evals son explícitos.
Revisar un workflow auto-recuperable	Self-healing workflow review checklist	Clases de falla, recovery policy, idempotencia, compensación, breakers, paquetes de replay y regression evals son explícitos.
Revisar una arquitectura de agent personal	Personal agent architecture review checklist	Consentimiento, división local/nube, credenciales, memory, controles de usuario, conectores y evidencia de auditoría son explícitos.
Revisar el bar final	10/10 production gate scorecard	El sistema es revisable, testeable, observable, reversible y tiene ownership.
Comparar un capstone con tu propio sistema	Capstone review scorecard	Tienes un plan concreto de adaptación y una lista de brechas bloqueadoras.
Calificar un pattern antes de adoptarlo	Pattern evaluation scorecard	El pattern tiene evidencia de goal, boundary, autonomía, tools, state, context, seguridad, evals, observability y operaciones.
Traducir terminología antigua de agent	Historical pattern migration record	Una etiqueta heredada vaga se mapea a una decisión de arquitectura actual con evidencia.
Revisar un paquete de skill reutilizable	Skill review checklist	Activación, formato de instrucciones, assets, seguridad, versionado, pruebas, observability y decisión final son explícitos.
Revisar un pattern	La checklist de revisión de pattern correspondiente abajo	Casos de uso, casos a evitar, modos de falla, evals y controles de producción tienen evidencia.



Usa el artifact más pequeño que pruebe la decisión. Agrega el production gate solo cuando el sistema esté lo suficientemente cerca del release como para que la evidencia faltante deba bloquear el progreso.

Ejemplos Completados

Las plantillas en blanco solo ayudan después de que el lector entiende el estándar de evidencia. Usa ejemplos completados para comparar la densidad y especificidad de tu propio artifact.



Usa [Completed agent ADR examples](#) para decisiones de arquitectura. Usa [Completed lab evidence examples](#) para paquetes de evidencia de laboratorio y revisiones de eval. Usa [Completed production readiness examples](#) cuando un equipo necesita calibrar evidencia de owner, gate, blocker, rollback y decisiones de release.

Versiones descargables:

- [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

Usa este ADR al adoptar LangGraph, AutoGen, Mastra, CrewAI, un mini-runtime u otro framework.

```
# ADR-000: Choose [Framework] for [Agent or Workflow]
```

```
## Status
```

```
Proposed | Accepted | Superseded
```

```
## Context
```

```
¿Qué problema de producto estamos resolviendo?
```

```
¿Qué workflow visible para el usuario alojará este framework?
```

```
¿Qué restricciones importan: lenguaje, despliegue, cumplimiento, habilidades del equipo, latencia, costo o infraestructura existente?
```

```
## Decision
```

```
Usaremos [framework] para [scope].
```

```
El framework será responsable de [state/control
```

flow/tools/memory/evals/observability/deployment].

La aplicación seguirá siendo responsable de [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

- los comandos de instalación y ejecución local están documentados;
- el state puede ser inspeccionado y reproducido;
- la policy se ejecuta antes de los efectos secundarios;
- los evals fallan la build ante regresiones críticas;
- los traces explican una ejecución fallida;
- el rollback puede deshabilitar la capability riesgosa;
- el código específico del framework no oculta la policy del producto.

Consecuencias

¿Qué se vuelve más fácil?

¿Qué se vuelve más difícil?

¿Qué riesgos de lock-in o migración permanecen?

¿Qué incidente en producción nos haría reconsiderar esta decisión?

Disparador de Revisión

Revisa este ADR después de una actualización de model, actualización de framework, incorporación de una nueva tool con capacidad de escritura, nuevo tipo de memory, incidente en producción o repetidas intervenciones humanas.

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
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: cada ruta de alta autoridad tiene evidencia
yellow: solo para lanzamiento interno limitado o de solo lectura
red: solo demo; sin usuarios reales, datos ni efectos secundarios

Lab-To-Production Checklist

Use this checklist after completing any hands-on lab.

lab:

target product slice:

framework/language:

owner:

Arquitectura

- ☐ Pattern seleccionado y enlazado
- ☐ Decisión de framework registrada
- ☐ State owner nombrado
- ☐ Límite de tool definido
- ☐ Límite de policy definido
- ☐ Límite de aprobación humana definido cuando sea necesario

Implementación

- ☐ Comando de instalación documentado
- ☐ Comando de ejecución local documentado
- ☐ Comando de prueba documentado
- ☐ Comando de eval documentado
- ☐ .env.example comprometido
- ☐ Secrets excluidos del código fuente
- ☐ Tool schemas validados
- ☐ Side effects usan claves de idempotencia
- ☐ Los errores tienen resultados tipados

Producción

- ☐ Checkpoint o decisión explícita de stateless registrada
- ☐ Trace schema implementado
- ☐ Redacción de trace implementada
- ☐ Umbral de eval definido
- ☐ CI gate configurado
- ☐ Ruta de rollback documentada
- ☐ Kill switch probado
- ☐ Runbook creado

Evidencia

- ☐ Salida de prueba adjunta
- ☐ Salida de eval adjunta
- ☐ Ejemplo de trace adjunto
- ☐ ADR enlazado
- ☐ Owner aceptó el riesgo residual

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:

¿Qué sucedió?

¿Qué límite falló?

☐ state

☐ tool

☐ policy

☐ approval

☐ memory

☐ retrieval

☐ model/prompt

☐ workflow

☐ observability

☐ eval gate

¿Qué debió haber pasado?

Nuevo caso de eval:

input:

expected trajectory:

expected tool behavior:

expected policy behavior:

expected output:

blocking threshold:

Regla de lanzamiento:

☐ bloquea futuros lanzamientos

☐ solo advertencia

☐ solo monitoreo

Seguimiento:

cambio de código:

cambio de policy:

cambio de runbook:

actualización de ADR:

owner:

fecha límite:

Un incidente que no produce un eval, un cambio de policy o una actualización de runbook probablemente se repetirá.

Capítulos Relacionados

- [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

La evaluación convierte el desarrollo de agents de suposiciones basadas en demos a ingeniería real.

Sin evals, el equipo discute a partir de demos, anécdotas y percepciones. Con evals, el equipo puede decir qué falló, por qué importa, si el siguiente cambio ayudó y qué rompió.

Usa este capítulo cuando un agent pasa de prototipo a producción, o cuando un prototipo ya parece impresionante pero nadie puede probar que es confiable.

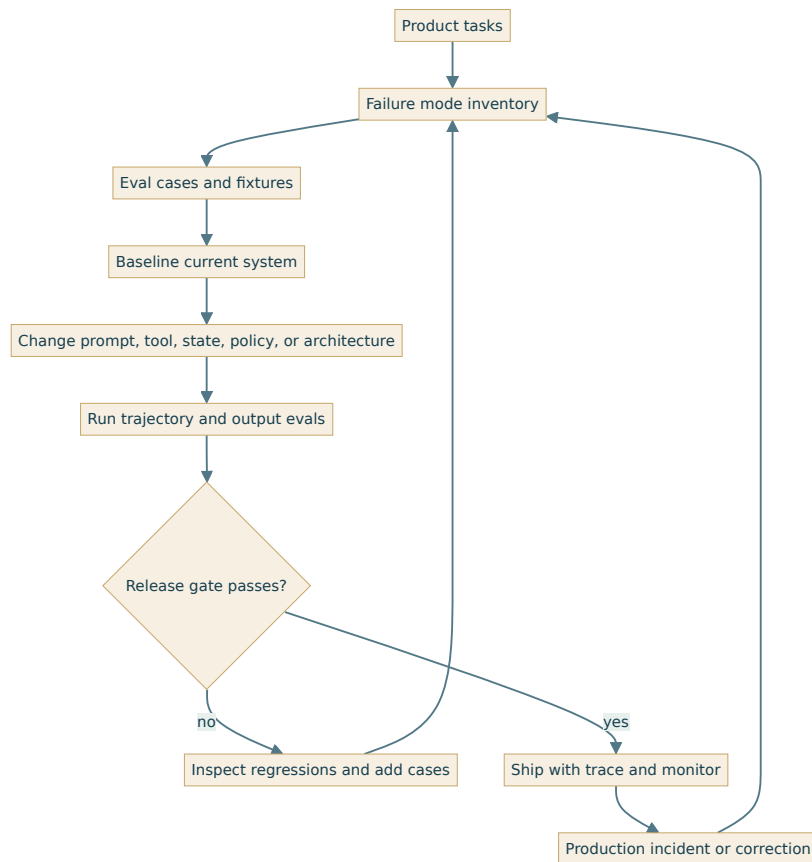
Intent

Construye agents a partir de modos de falla y métricas de éxito, no de demos. Una demo muestra que el agent puede funcionar una vez. Un eval suite muestra si sigue funcionando en los casos que importan.

El eval loop es:

1. listar las tasks que el agent debe manejar;
2. listar las formas en que esas tasks pueden fallar;
3. definir métricas de negocio y calidad;
4. crear fixtures y datasets;
5. ejecutar el agent contra esos datasets;
6. inspeccionar fallas;
7. mejorar prompts, tools, state, policy o arquitectura;
8. repetir en cada cambio significativo.

La palabra clave es arquitectura. Si un eval falla porque al agent le falta state, editar el prompt es la solución incorrecta. Si falla porque el tool no es seguro, una mejor instrucción no es suficiente. Los evals deben poder cambiar el diseño.



Usa el loop como función de control. Cada cambio significativo debe nombrar la falla que ataca, el eval que prueba la mejora y el release gate que no debe romper.

Qué Evaluar

Evalúa toda la trayectoria, no solo la respuesta final.

Capa	Qué Revisar
Routing	Ruta correcta, confianza, comportamiento de fallback.
Planning	Pasos válidos, orden de dependencias, restricciones faltantes.
Retrieval	Relevancia de fuentes, frescura, cobertura, citas.
Tool use	Tool correcto, input válido, efectos secundarios seguros, manejo de errores.
Memory	Recall correcto, escritura segura, sin filtraciones obsoletas o privadas.
Policy	Permisos, requisitos de aprobación, rechazos.
Final output	Corrección, completitud, tono, formato, evidencia.
Operations	Latencia, costo, reintentos, eventos de breaker, escalamiento.

Si solo se evalúa la respuesta final, el equipo perderá la falla que la causó. La respuesta puede parecer correcta aunque el agent haya usado evidencia obsoleta, omitido aprobación, reintentado un tool seis veces o escrito bad memory para la siguiente ejecución.

Lentes de Evaluación por Responsabilidad

Diferentes patterns requieren diferentes evidencias. No uses un solo puntaje genérico de “calidad de respuesta” para cada agentic system.

Responsabilidad del Pattern	Pregunta Principal	Prueba Mínima
Loop control	¿La ejecución se detuvo por la razón correcta?	Casos de presupuesto de pasos, presupuesto de tools, timeout, retry y stop-reason.
Context assembly	¿El model vio la evidencia y exclusiones correctas?	Aserciones de context packet para fuentes incluidas, omitidas, frescura y presupuesto.
Tool use	¿El sistema usó la autoridad mínima segura?	Tools simulados, casos de tools prohibidos, revisiones de schema, revisiones de aprobación y rutas de error.
Memory	¿El sistema recordó solo lo que policy permite?	Escritura permitida, escritura denegada, corrección, eliminación, recall obsoleto y casos de aislamiento de tenant.
Retrieval	¿El sistema encontró y citó la evidencia correcta?	Relevancia, cobertura, cita, casos de evidencia faltante y fuentes en conflicto.
Multi-agent coordination	¿La delegación mejoró el resultado sin ocultar la responsabilidad?	Línea base de single-agent, falla de worker, desacuerdo, fusión y casos de owner final.
Human approval	¿La acción correcta se pausó con suficiente información?	Casos de aprobación requerida, aprobación denegada, timeout, reanudación y registro de auditoría.
Runtime operation	¿El equipo puede explicar y recuperarse de una falla?	Integridad de trace, replay, rollback, breaker, fixture de incidente y casos de release-gate.

Esta tabla es una herramienta de diseño. Si el pattern seleccionado introduce un riesgo, el eval debe atacar ese riesgo directamente.

Contrato de Caso Eval Mínimo

Un caso de eval debe ser lo suficientemente pequeño para revisar y lo suficientemente específico para fallar por la razón correcta.

```
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";
};
};

```

El schema es intencionalmente simple. Permite al equipo probar final output, trayectoria, policy, memory y observability sin convertir el eval suite en otro proyecto de agent.

Inventario de Modos de Falla

Comienza cada proyecto de agent con un inventario de modos de falla. Este es el artifact temprano más útil porque convierte el miedo en casos comprobables.

Ejemplos:

- selecciona la ruta incorrecta;
- pide información que ya tiene;
- omite una aprobación requerida;
- llama un write tool cuando un análisis de solo lectura es suficiente;
- recupera evidencia irrelevante;
- confía en memory obsoleta;
- entra en loop sin avanzar;
- da una respuesta confiada con evidencia débil;
- expone datos sensibles;
- excede el presupuesto de costo o latencia;
- no puede reanudar después de una interrupción.

Cada modo de falla debe mapearse al menos a una prueba, consulta de trace o alerta en producción. Si una falla es importante pero invisible, el sistema no está listo.

Un buen inventario es específico al producto. “Hallucination” es demasiado amplio. “Responde preguntas sobre refund-policy sin citar el documento de policy actual” es comprobable. “Tool misuse” es demasiado amplio. “Llama `issue_refund` antes de que exista un registro de aprobación de manager” es comprobable.

Métricas

Usa un conjunto pequeño de métricas que coincidan con el producto. Más métricas no crean más verdad. Usualmente crean más formas de ignorar la falla real.

Métricas comunes:

- tasa de completion de tasks;

- corrección;
- cobertura de evidencia;
- precisión en llamadas a tools;
- precisión de rutas;
- precisión y recall de aprobaciones;
- tasa de hallucination;
- tasa de corrección por usuario;
- tasa de escalamiento humano;
- costo por task completada;
- latencia por paso;
- tasa de incidentes.

No optimices una métrica en aislamiento. Una tasa de escalamiento más baja es mala si el agent está tomando decisiones inseguras en su lugar. Menor latencia es mala si el agent omite retrieval. Mayor completion de tasks es mala si completa la task incorrecta.

Las mejores métricas forman un conjunto de restricciones: la calidad no puede mejorar violando requisitos de seguridad, costo, latencia, evidencia o aprobación.

Tipos de Eval Dataset

Dataset	Fuente	Uso
Golden tasks	Ejemplos escritos a mano	Cobertura básica de regresión.
Adversarial tasks	Casos límite diseñados	Seguridad, routing, policy, prompt injection.
Production traces	Ejecuciones reales con redacción	Comportamiento realista y fallas de baja frecuencia.
Human-labeled data	Expertos en la materia	Ground truth para tasks de alto juicio.
Synthetic variants	Generados de casos conocidos	Expansión de cobertura tras revisión humana.
Incident fixtures	Fallas pasadas	Prevenir regresiones.

Los production traces son valiosos, pero requieren controles de privacidad y etiquetado cuidadoso. No viertas conversaciones de usuarios sin procesar en un pipeline de eval sin redacción, reglas de retención y controles de acceso.

Los synthetic data son útiles para expandir cobertura, pero no deben convertirse en el ground truth por sí solos. Los casos generados suelen omitir las partes complicadas del uso real: context faltante, instrucciones contradictorias de usuario, frases extrañas, registros obsoletos y caídas parciales de tools.

Judges

Los model judges pueden ayudar, pero necesitan restricciones. Son revisores, no ley.

Usa model judges para:

- calidad subjetiva;
- revisión basada en rúbricas;
- comparar dos outputs;
- verificar si una afirmación está respaldada por evidencia.

Usa revisiones deterministas para:

- validez de schema;
- campos requeridos;
- tools prohibidos;
- límites de presupuesto;
- presencia de citas;
- etiquetas de ruta;
- puertas de permisos.

Combina ambos. Un model judge no debe ser el único control antes de una acción de alto riesgo.

También evalúa al judge. Dale ejemplos buenos y malos conocidos. Rastrea falsos positivos y falsos negativos. Si el judge premia prosa pulida sobre evidencia, hará que el agent se vea mejor mientras el sistema empeora.

Evaluation-Driven Development Loop

Para cada cambio significativo en el agent:

1. Agrega o actualiza casos de eval antes de cambiar prompts o tools.
2. Ejecuta el sistema actual para establecer una línea base.
3. Realiza el cambio más pequeño que apunte a la falla.
4. Ejecuta nuevamente la suite de eval.
5. Inspecciona regresiones, no solo el puntaje promedio.
6. Promueve el cambio solo si mejora el comportamiento objetivo sin afectar casos críticos.

Este loop funciona para prompts, tools, policies, memory, routing, models y orquestación.

Los puntajes promedio son peligrosos. Un cambio que mejora casos fáciles y rompe casos de policy no es una mejora. Mantén un conjunto pequeño de evals bloqueantes que deben pasar siempre: validez de schema, llamadas prohibidas a tools, comportamiento de aprobación, límites de privacidad y incident fixtures conocidos.

Production Monitoring

El monitoreo en runtime debe alimentar la suite de eval. El sistema en producción es la mejor fuente de casos que el equipo no imaginó, así que captura las señales que los exponen: ejecuciones fallidas, overrides humanos, correcciones de usuario, rutas de baja confianza, eventos de breaker, fallas de tools, ejecuciones costosas o lentas, denegaciones de policy y escrituras inusuales en memory. Cada incidente serio debe convertirse en al menos un caso de eval. El objetivo no es culpar. El objetivo es hacer que la falla sea difícil de repetir.

El monitoreo también debe mostrar cuando la suite de eval está desactualizada. Si los trases de producción contienen tipos de task, rutas de tool o decisiones de policy que nunca aparecen en las pruebas, la suite de eval ya no representa el sistema.

Para el operating loop de producción, consulta [Production Evaluation Feedback Loops](#).

Release Gates

Antes de lanzar un cambio significativo en el agent, define qué debe pasar.

Release gates mínimos:

- sin fallas de schema en golden tasks;
- sin llamadas prohibidas a tools;
- sin IDs de trace faltantes;
- sin saltos de aprobación;
- sin regresión en incident fixtures;
- sin afirmaciones no soportadas en respuestas con evidencia;
- sin fallas de alto riesgo sin resolver en la review queue.

Estos gates deben ser aburridos. Ese es el punto. Evitan que el equipo lance un demo emocionante que reabra fallas antiguas.

Related Chapters

- [Observability and Evals](#)
- [Production Evaluation Feedback Loops](#)
- [Circuit Breakers, Fallbacks, and Replay](#)
- [Policy Enforcement](#)
- [Evaluator-Optimizer](#)
- [Agent Development Lifecycle](#)

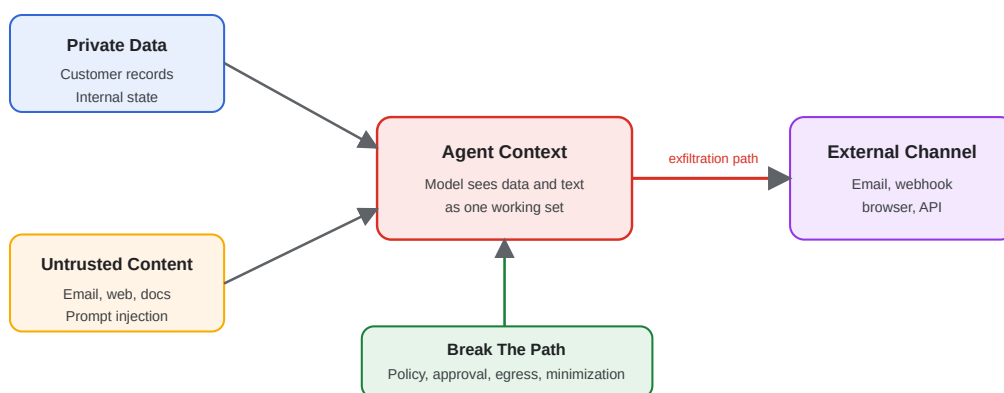
Agent Threat Model

La seguridad de un agent comienza con una pregunta simple: ¿qué puede hacer que el model haga el sistema? Un chatbot puede dar una mala respuesta. Un agent puede leer datos privados, combinarlos con instrucciones no confiables y enviar algo fuera del límite. Ese es un tipo de riesgo diferente, y merece un threat model diferente.

Así que no comiences el threat model con “el model podría alucinar”. Comienza con autoridad, datos, tools y comunicación.

Descarga la hoja reutilizable: [agent threat model worksheet](#).

Agent Threat Model Risk Path



La combinación peligrosa

La forma más peligrosa de un agent combina tres capabilities:

1. acceso a datos privados o confiables;
2. exposición a contenido no confiable;
3. la capacidad de comunicarse o actuar fuera del sistema.

Cuando las tres se encuentran en el mismo execution path, el agent puede convertirse en un puente de exfiltración. La forma es fácil de imaginar: el agent lee datos internos de clientes, luego navega una página web, correo electrónico, ticket, documento o comentario de repositorio que controla otra persona, y ese contenido no confiable le indica al model que envíe los datos privados a través de un correo electrónico, webhook, comentario de issue, formulario de navegador o llamada de API.

El model puede no experimentar esto como un ataque en absoluto. Lo experimenta como solo otra instrucción en el context. Por eso las defensas basadas solo en prompt no son suficientes.

Límite de seguridad

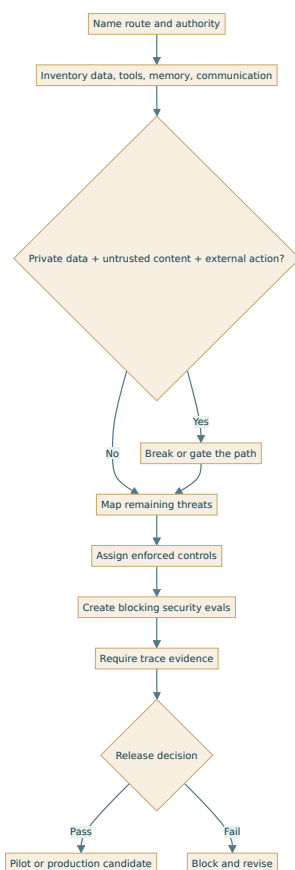
El model puede proponer; el software debe decidir si la propuesta está permitida. Mantén esas responsabilidades en manos separadas.

Capa	Responsabilidad
Model	Interpreta el task, propone acciones, resume evidencia.
Orchestrator	Posee el loop, state, presupuestos, ruta y condición de parada.
Policy engine	Decide si la acción está permitida.
Tool gateway	Valida, ejecuta, registra, deniega o enruta para aprobación.
Sandbox	Limita el acceso a filesystem, red, procesos, navegador y credenciales.
Observability	Registra suficiente trace data para replay, auditoría y respuesta a incidentes.

Si el model también es el policy engine, el límite es débil por construcción: el mismo context que transporta el ataque también transporta la instrucción de policy destinada a detenerlo.

Flujo de revisión de amenazas

Usa este flujo cuando se agregue una nueva ruta, tool, tipo de memory, fuente de datos o canal de comunicación. Un threat model solo es útil si termina en controles, evals, traces y una decisión de lanzamiento.



Mapa STRIDE para agents

STRIDE solo es útil después de traducirlo a límites específicos de agent. Úsalo para preguntar qué pueden ser engañados para hacer el model, runtime, tools, memory y rutas de comunicación.

Clase de amenaza	Pregunta específica para agent	Control	Evidencia
Spoofing	¿Puede un usuario, tool, agent, servidor MCP, webhook o documento hacerse pasar por un actor confiable o fuente de instrucciones?	Identidad firmada, verificación de audiencia, credenciales por ruta, etiquetas de origen.	Claims de identidad en trace, solicitud denegada por audiencia incorrecta, etiqueta de confianza de origen.
Tampering	¿Puede contenido no confiable modificar goals, policy, argumentos de tool, memory, traces o fixtures de evaluación?	Separación de instrucciones/datos, validación de schema, policy inmutable, almacén de trace protegido.	Resultado hostil de tool rechazado, escritura en memory denegada, verificación de integridad de trace.
Repudiation	¿Puede un usuario, agent, tool o aprobador negar una acción porque el sistema carece de un registro durable?	Run IDs, registros de aprobación, logs de decisiones de policy, eventos de auditoría de tool.	Trace completo desde la solicitud hasta la razón de parada.
Information disclosure	¿Pueden datos privados, secretos, citas, memory o traces filtrarse a través de output, tools, logs, navegador o mensajes entre agents?	Minimización de datos, redacción, allowlist de salida, clasificación de destino.	Prueba de redacción, salida bloqueada, eval de exposición de datos privados.
Denial of service	¿Pueden loops, reintentos, retrieval, tools, colas, llamadas al model o esperas de aprobación agotar costo, latencia, cuota o humanos?	Presupuestos, timeouts, límites de concurrencia, circuit breakers, cancelación.	Caso de agotamiento de presupuesto, alerta de límite de cola, trace de breaker.
Elevation of privilege	¿Puede el agent pasar de lectura a escritura, de borrador a envío, de un tenant a otro, o de un scope de tool a una autoridad más amplia?	Least privilege, allowlists de tool por ruta, puertas de aprobación, tokens con scope.	Eval de tool prohibido, denegación entre tenants, trace que requiere aprobación.

Este mapa debe generar tickets, no solo discusión. Cada fila de alto riesgo necesita un responsable, un control aplicado, un eval y un campo de trace.

Clasifica las capabilities de los tools

Cada tool debe tener metadata de capability. Sin esto, el runtime no puede razonar sobre el riesgo.

Capability	Pregunta
Private data access	¿Puede este tool leer datos de clientes, tenant, secretos, datos internos o privilegiados?
Untrusted content access	¿Puede este tool obtener contenido controlado por usuarios, sitios externos, correos, documentos, tickets o comentarios?
External communication	¿Puede este tool enviar datos a otra persona, sistema, red, repositorio, navegador o workflow?
Side effects	¿Puede este tool escribir, borrar, comprar, desplegar, enviar mensajes, mutar o disparar trabajo?
Credential scope	¿Con qué credenciales se ejecuta el tool?
Approval requirement	¿Qué llamadas requieren un humano o un workflow de mayor confianza?

Esta clasificación debe vivir fuera del prompt. Un manifiesto de tool, configuración de gateway, archivo de policy o registro de servicio es mucho más fácil de auditar que un párrafo de instrucciones en lenguaje natural.

El policy check puede ser pequeño, pero debe estar fuera del 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' };  
}
```

Esto no hace seguro todo el sistema por sí solo. Muestra el límite: el model puede proponer un plan, pero el software decide si la cadena de tools está permitida.

Rompe el camino peligroso

El objetivo no es prohibir agents útiles. Es asegurarse de que al menos una parte del camino peligroso esté ausente o controlada.

Control	Qué elimina o restringe
No hay datos privados en esta ruta	El agent puede inspeccionar contenido no confiable y comunicarse, pero no puede acceder a datos sensibles.
No hay contenido no confiable en esta ruta	El agent puede trabajar con datos y tools internos confiables, pero no puede ingerir instrucciones hostiles.
Sin comunicación externa	El agent puede leer y razonar, pero no puede exfiltrar ni actuar fuera del límite.
Proxy de tool controlado por policy	El agent puede proponer llamadas, pero la policy decide qué se ejecuta.
Aprobación humana	Acciones de alto riesgo requieren que una persona inspeccione la acción propuesta y la evidencia.
Controles de salida	Las rutas de red y mensajería están limitadas por ruta, dominio, destino o clase de datos.
Minimización de datos	Los tools solo devuelven los campos requeridos para el task.
Redacción	Los valores sensibles se eliminan antes de que entren al context del model o a los traces.

La palabra que importa es aplicar. Una frase en el system prompt no es lo mismo que un límite aplicado, y los atacantes conocen la diferencia incluso cuando el model no.

Llamadas a Tools Controladas por Policy

Para agents en producción, las llamadas a tools deben pasar por una capa de enforcement que pueda permitir la llamada, denegarla, requerir aprobación, transformar o redactar sus argumentos, adjuntar una clave de idempotencia, registrar la decisión de policy y vincular esa decisión al run trace. Implementa esto

como un gateway, proxy, middleware o wrapper de tool-registry; la forma exacta importa menos que el límite en sí. El model no debe llamar tools de alto riesgo directamente.

Contenido No Confiable

Trata los resultados de tools, documentos recuperados, páginas web, correos electrónicos, tickets, comentarios, logs y archivos subidos por usuarios como datos, no instrucciones. Ninguno de ellos debe poder redefinir los goals del sistema, los permisos de tools, los requisitos de aprobación, los destinos de output, las reglas de escritura en memory, las excepciones de policy, ni la identidad y rol del agent. El agent puede resumir contenido no confiable tanto como quiera. No debe obedecer contenido no confiable como gobernanza.

Riesgos de Memory y Persistencia

La memory hace que los ataques sean durables. Una escritura insegura puede contaminar ejecuciones futuras mucho después de que la interacción original haya desaparecido, por eso las escrituras en memory necesitan su propia policy. Una escritura en memory debe registrar su fuente, el actor, la marca de tiempo, la clase de privacidad, la regla de expiración o retención, un nivel de confianza, la razón de la escritura y una ruta de corrección o eliminación. No permitas que el model reescriba silenciosamente lo que el sistema creerá mañana.

Guía para Evaluación

Los security evals deben probar el comportamiento a lo largo de toda la trayectoria, no solo la respuesta final. Construye casos donde un documento recuperado contenga instrucciones para ignorar la policy, donde una página web pida al agent filtrar datos a través de una llamada a tool, donde un correo electrónico le pida reenviar context interno, y donde un resultado de tool lleve instrucciones hostiles. Agrega casos donde haya datos privados disponibles pero no necesarios para el task, donde se proponga una llamada a tool de alto riesgo sin la aprobación requerida, y donde una escritura en memory intente almacenar datos sensibles o no confiables. En cada caso, el comportamiento correcto es rechazar, escalar, redactar o elegir una ruta más segura.

Luego mide lo que revelan esos casos: llamadas a tools peligrosas bloqueadas, llamadas a tools seguras permitidas, rechazos falsos que detienen trabajo válido, exposición de datos privados, precisión en el enrutamiento de aprobación, completitud de traces y tasa de conversión de incidentes a eval. Ejecuta esto contra tools simulados para saber si el sistema habría llamado al tool peligroso antes de que cualquier efecto real sea posible.

Checklist de Diseño

Antes de que un agent pueda acceder a datos privados, contenido no confiable o comunicación externa, responde estas preguntas:

- ¿Qué tools pueden leer datos privados?
- ¿Qué tools pueden obtener contenido no confiable?
- ¿Qué tools pueden comunicarse externamente?
- ¿Qué rutas combinan esas capabilities?

- ¿Qué llamadas a tools requieren aprobación?
- ¿Dónde se aplica la policy en el código?
- ¿Qué pasa si falta metadata de capability?
- ¿Puede el sistema denegar por defecto?
- ¿Pueden los operadores deshabilitar rápidamente una ruta o tool?
- ¿Los traces están redactados pero siguen siendo útiles?
- ¿Un incidente en producción puede convertirse en un eval case?

Si la respuesta a “¿dónde se aplica esto?” es “en el prompt”, el diseño no está terminado.

Capítulos Relacionados

- [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

La seguridad de un agent es diferente a la seguridad de un chatbot porque los agents pueden actuar. Pueden leer datos privados, llamar APIs, escribir archivos, ejecutar código, activar workflows y comunicarse con otros sistemas.

Usa este capítulo cuando un agent tiene tools, memory, datos externos o efectos secundarios.

Comienza con el [Agent Threat Model](#) si necesitas clasificar el riesgo del sistema. Usa este capítulo cuando estés listo para diseñar los controles de contención.

Este capítulo va después de harnesses y production runtime porque la seguridad no es una característica del prompt. Los mismos límites que hacen que un agent sea operable también lo hacen securable: identidad, autorización, sandboxing, egress, aprobaciones, trazabilidad y respuesta a incidentes.

Security Model

Asegura los agents separando cuatro preocupaciones: lo que el usuario solicita, lo que el model propone, lo que la policy permite y lo que el tool realmente ejecuta. El model no debe ser el motor de policy. Puede clasificar la intención o explicar una decisión, pero el software determinista debe hacer cumplir los permisos.

En producción hay una quinta preocupación: cómo la solicitud cruza los límites de confianza. Si un agent llama a otro agent, a un tool gateway, a un servidor MCP, a un browser worker, a un workflow engine o a un data service, el sistema necesita transporte seguro, identidad del llamador, autorización con alcance y correlación de trace. TLS, mTLS, OAuth u OIDC, enforcement de policy y observability son parte del mismo límite.

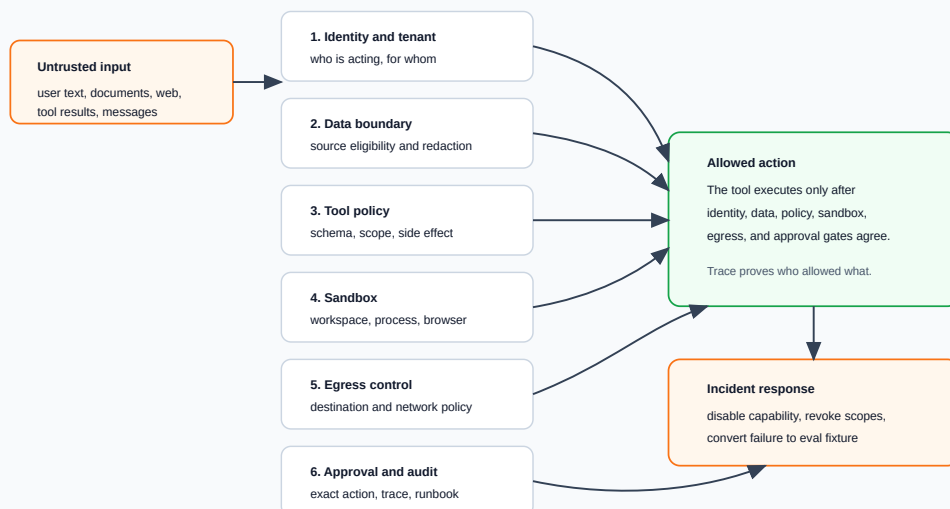
El model puede proponer trabajo. No debe crear credenciales, elegir sus propios scopes, decidir que TLS es opcional, aprobar su propio egress de red ni ocultar eventos de seguridad en los traces.

Security Layers

Usa este diagrama para ver el sandboxing como una capa dentro de un límite de seguridad más amplio. El model propone trabajo; la identidad, acceso a datos, policy de tools, sandboxing, controles de egress, aprobación, auditoría y respuesta a incidentes deciden lo que realmente puede suceder.

Agent Security Layers

Sandboxing is one layer. Safe agents also need identity, policy, approval, egress control, and audit.



Security rule: never let the model turn a request directly into authority. It proposes; the runtime authorizes.

Threats To Design For

La seguridad de agents es más fácil de razonar cuando la amenaza es concreta. Comienza con estos casos antes de agregar otros más exóticos.

Threat	Cómo se presenta	Control que debe detectarlo
Prompt injection	Un documento, correo, página o resultado de tool le indica al agent que ignore la policy o filtre datos.	Aislamiento de fuente, jerarquía de instrucciones, guardrails en retrieval, verificación de output.
Tool abuse	El model selecciona un tool legítimo para un propósito inseguro.	Policy de tool, scopes, aprobaciones, clasificación de efectos secundarios.
Secret leakage	Un token, llave, registro de cliente o nota interna entra al prompt, memory, logs o respuesta final.	Manejo de secretos, redacción, minimización de datos, clasificación de trace.
Data exfiltration	Datos privados salen por navegador, API, correo, escritura de archivos o mensaje cross-agent.	Policy de egress, clasificación de destino, puertas de aprobación, registros de auditoría.
Confused deputy	El agent usa la autoridad de un usuario o un token de servicio en un context donde no debería.	Scopes delegados, verificación de audiencia, verificación de tenant, permisos a nivel de ruta.
Unsafe delegation	Un agent entrega trabajo a otro agent con más tools, propiedad poco clara o identidad débil.	Autorización A2A, sobres de task con scopes, handoffs enlazados a trace.

Threat	Cómo se presenta	Control que debe detectarlo
Replay or duplicate action	Un mensaje, llamada de tool o paso de workflow se ejecuta dos veces.	Claves de idempotencia, nonces, expiración, registros durables de efectos secundarios.
Sandbox escape	Código, automatización de navegador o manejo de archivos accede fuera del workspace previsto.	Aislamiento de filesystem, bloqueos de red, límites de procesos, sin credenciales ambientales.

No trates estos como riesgos abstractos. Conviértelos en pruebas, traces y runbooks operativos.

The High-Risk Combination

La forma más peligrosa de un agent combina acceso a datos privados o de confianza, exposición a contenido no confiable y la capacidad de realizar acciones externas. Cuando estos tres se combinan, un documento, página web, correo o resultado de tool malicioso puede intentar guiar al agent para filtrar datos o realizar una acción insegura.

Mitiga esto con privilegio mínimo, aislamiento de contenido, puertas de aprobación, controles de egress y verificaciones explícitas de policy.

Security Control Matrix

Usa controles de seguridad por límite, no como una sola capa genérica de “guardrail”.

Boundary	Controles requeridos
Data access	Verificación de tenant, elegibilidad de fuente, permisos por fila o documento, minimización de datos, redacción.
Tool execution	Schemas tipados, puerta de policy, credenciales con scope, clave de idempotencia, timeout, registro de auditoría.
Network egress	Dominios en allowlist, redes privadas bloqueadas, clasificación de destino, verificación de TLS.
Cross-agent messages	TLS o mTLS, claims de OAuth u OIDC, verificación de audiencia, scopes, protección contra replay, ID de trace.
Browser or code execution	Aislamiento en contenedor o VM, filesystem restringido, sin credenciales ambientales, límites de CPU y tiempo.
Memory writes	Fuente, clase de privacidad, expiración, confianza, ruta de corrección, decisión de policy.
Human approval	Acción propuesta, evidencia, resultado de policy, identidad del aprobador, timestamp, llamada final de tool.

Boundary	Controles requeridos
Observability	Traces redactados, decisiones de policy, claims de identidad, llamadas de tool, intentos de egress, razones de detención.

La matriz hace visible una cosa: el sandboxing es necesario, pero no suficiente. Un sandbox limita el radio de impacto de la ejecución. No reemplaza identidad, autorización, policy, aprobación, control de egress ni observability.

Sandboxing

Haz sandbox a cualquier agent que pueda ejecutar código, operar un navegador, manipular archivos o llamar APIs de escritura.

Controles de sandbox:

- proceso, contenedor, VM o perfil de navegador aislado;
- filesystem de solo lectura por defecto;
- directorio de workspace con scope;
- sin credenciales ambientales;
- inyección explícita de secretos solo para tools aprobados;
- restricciones de red saliente;
- límites de tiempo y CPU;
- límites de tamaño de archivo;
- logs de auditoría para cada efecto secundario;
- limpieza después de completar la ejecución.

Agents de coding y uso de computadora necesitan sandboxes más fuertes que agents de investigación solo lectura.

Sandbox Tiers

No todos los agents necesitan el mismo sandbox. Ajusta la contención al trabajo.

Tier	Usar para	Controles mínimos
Read-only	Búsqueda, resumen, clasificación, retrieval sobre datos aprobados.	Sin tools de escritura, sin secretos en context, retrieval trazable, redacción de output.
Tool-limited	Workflows de negocio con tools tipados y efectos secundarios acotados.	Scopes por tool, verificaciones de policy, claves de idempotencia, aprobaciones para escrituras.
Workspace	Coding, generación de documentos, transformación de datos, edición local de archivos.	Workspace con scope, montajes de solo lectura donde sea posible, revisión de diffs, limpieza.
Browser	Navegación web, llenado de formularios, automatización de UI.	Perfil de navegador separado, redes privadas bloqueadas, controles de descarga, aislamiento de credenciales.

Tier	Usar para	Controles mínimos
Code execution	Shell, notebooks, instalación de paquetes, código generado, pruebas.	Límite de contenedor o VM, límites de CPU y tiempo, policy de red, sin secretos ambientales.

El tier es una decisión de producto. Un agent de soporte para borradores no debe convertirse silenciosamente en un agent de automatización de navegador solo porque un tool era conveniente.

Identity And Credentials

Los agents no deben ejecutarse con credenciales ambientales de la plataforma. Cada credencial debe estar asociada a una ruta, task, tool, usuario, tenant y decisión de policy.

Usa:

- tokens de corta duración en lugar de secretos de larga duración;
- claims de OAuth u OIDC para identidad de usuario y servicio;
- verificación de audiencia para que un token de un servicio no pueda llamar a otro;
- scopes que coincidan con la capability delegada;
- mTLS para identidad entre servicios donde la plataforma lo soporte;
- inyección de credenciales por tool después de aprobación de policy;
- revocación de credenciales como parte de la respuesta a incidentes;
- redacción para que las credenciales nunca entren en prompts, memory, fixtures de eval o traces.

El runtime debe poder responder qué identidad llamó a qué tool con qué scopes y qué versión de policy lo permitió. Si no puede, el modelo de credenciales no está listo para agents en producción.

Network And Egress

El acceso a red es un tool, no un derecho por defecto. Un agent de investigación que puede navegar la web pública no debe poder llamar automáticamente APIs internas, servicios de metadata, redes privadas, webhooks de clientes o dominios arbitrarios.

Controles útiles de egress:

- dominios en allowlist por ruta y tool;
- bloquear rangos de IP privadas y endpoints de metadata en la nube;
- requerir validación de certificado TLS para llamadas remotas;
- separar navegación de APIs de negocio autenticadas;
- clasificar destinos como internos, socios, clientes, públicos o desconocidos;
- registrar intentos de egress denegados con IDs de trace;
- requerir aprobación para nuevos destinos o exportación de datos de alto riesgo;
- fallar cerrado cuando falte la clasificación de destino.

El egress es donde las fugas de datos se vuelven reales. Trátalo como parte del límite de policy.

Control de Acceso

Otorga acceso por rol, task y ruta. Un support-answer agent puede leer documentos públicos pero no puede emitir reembolsos. Un billing workflow puede leer el state de una factura pero necesita aprobación para aplicar crédito. Un coding agent puede editar archivos en una rama pero no puede desplegar en producción. Un research agent puede navegar la web pero no puede acceder a datos de clientes. Evita listas globales de tools. Cada ruta o agent debe recibir solo los tools necesarios para el task.

El control de acceso debe verificar tanto al usuario como la ruta del agent. Un usuario puede tener permiso para hacer algo manualmente, pero eso no significa que cada agent actuando en nombre de ese usuario deba heredar todo el conjunto de permisos. La delegación debe ser más limitada que la autoridad del usuario.

Ejemplo de Código de Autorización

Mantén la autorización fuera del prompt. El model puede solicitar una llamada a un tool, pero el runtime decide si esa llamada puede ejecutarse.

```
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" };
}

```

La implementación real debe validar tokens y certificados firmados con librerías de la plataforma. El punto arquitectónico es más simple: la autorización ocurre antes de la ejecución, a partir de claims confiables del runtime, no del texto del model.

Guardrails

Los guardrails deben ejecutarse antes y después de las llamadas al model y antes de los tools.

Guardrail Point	Checks
Input	prompt injection, datos sensibles, solicitud no soportada, autorización del usuario.
Retrieval	fuentes no confiables, fuentes desactualizadas, límite de tenant, calidad de citación.
Tool intent	permiso, nivel de efecto secundario, policy, requerimiento de aprobación.
Tool input	schema, campos permitidos, minimización de datos, idempotency key.
Output	fuga de datos, claims no soportados, instrucciones inseguras, advertencias faltantes.
Memory write	privacidad, fuentes, expiración, ruta de corrección.

Ningún guardrail por sí solo es suficiente. Usa capas.

Secrets

Los agents no deben ver secrets sin procesar a menos que el contrato del tool lo requiera. Prefiere la ejecución de tools del lado del servidor, tokens con alcance limitado y credenciales de corta duración, con acceso a secrets otorgado por tool, sin secrets en prompts o memory, y traces redactados. Si un model puede leer un secret, asume que puede exponerlo accidentalmente.

No pases secrets a través del context en lenguaje natural. Dale al model un capability handle, no el secret en sí. El gateway del tool puede resolver ese handle después de que la policy lo permita.

Ejemplo de Security Profile

Un agent de producción debe tener un security profile explícito. El formato exacto puede variar, pero el runtime necesita estas decisiones en algún lugar fuera del 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"]
}
}

```

Este profile es deliberadamente aburrido. Ese es el punto. Un profile aburrido puede ser revisado, probado, comparado, revertido y vinculado a un incidente.

Approval Gates

Para un diseño detallado de aprobación, usa [Human Approval Gates](#). En este capítulo, la aprobación se trata como un control de seguridad para autoridad de alto riesgo.

Requiere aprobación para:

- movimientos de dinero;
- cambios de cuenta;
- comunicación con clientes a escala;
- cambios en infraestructura de producción;
- decisiones legales, médicas o de cumplimiento;
- eliminación o escrituras irreversibles;
- exportación amplia de datos;
- escalamiento de privilegios.

Los registros de aprobación deben incluir la acción propuesta, evidencia, resultado de la policy, aprobador, timestamp y llamada final al tool.

Security Observability

Los controles de seguridad solo ayudan a los operadores si son visibles. Registra eventos de seguridad como eventos de trace de primera clase:

- resultado de verificación de identidad;
- emisor del token, audiencia, sujeto, tenant y scopes después de la redacción;
- identidad de servicio TLS o mTLS para llamadas entre servicios;
- decisión de policy, razón y versión de la policy;
- inicio de sandbox, límites, acceso al filesystem denegado y egress de red denegado;
- inyección y rechazo de credenciales;
- solicitud de aprobación, decisión de aprobación e identidad del aprobador;

- invocación de tool, identificador de efecto secundario, idempotency key y razón de detención;
- acción de redacción y clase de retención de trace.

No conviertas la observabilidad en una fuga de datos. Tokens sin procesar, claves privadas, secrets, registros completos de clientes y cargas innecesarias no deben aparecer en traces. El objetivo es explicar la autoridad y el comportamiento, no duplicar datos sensibles.

Security Evals

Prueba los controles de seguridad antes del lanzamiento y después de cambios significativos.

Eval Case	Expected Behavior
Missing OAuth scope	La llamada al tool es denegada antes de la ejecución y la denegación queda registrada en el trace.
Wrong token audience	La solicitud cross-agent es rechazada antes de procesar el payload.
Prompt injection en documento recuperado	El agent trata el contenido como datos y no cambia la policy ni el destino.
Unapproved egress destination	La llamada de red es bloqueada y registrada con el trace ID.
Memory write contiene datos privados	La escritura es denegada, redactada o enviada para aprobación.
Tool requests broad credential	El runtime solo inyecta credenciales con alcance limitado o deniega la llamada.
Approval-required side effect	El workflow se pausa antes de la ejecución y registra el contexto de aprobación.
Replay de mensaje antiguo	El chequeo de idempotency o nonce bloquea la acción duplicada.
Browser intenta red privada	La solicitud es denegada antes de la navegación o carga.
Código intenta leer fuera del workspace	El acceso al filesystem es denegado y registrado en el trace.
Subagent pide tools más amplios	La delegación es rechazada o enviada para aprobación.
Trace contiene raw token	La prueba de redacción falla y bloquea el release.

Mide falsos positivos, llamadas inseguras a tools bloqueadas, fallas de redacción, campos faltantes en traces, intentos de egress no autorizados, precisión en el ruteo de aprobación, intentos de escape de sandbox y tiempo para investigar un incidente de seguridad.

Incident Response

Planea para incidentes de agents antes del lanzamiento. El equipo debe saber ya cómo deshabilitar un tool o una ruta, revertir un prompt o policy, revocar credenciales, aislar memory, inspeccionar traces, notificar a usuarios afectados y convertir el incidente en evals. Si la respuesta requiere arqueología de código en medio

de un incidente, el sistema no está listo operativamente.

La respuesta a incidentes de seguridad debe incluir acciones sobre identidad y credenciales: revocar tokens, rotar secrets, deshabilitar cuentas de servicio, invalidar nonces de mensajes, aislar traces afectados y preservar suficiente evidencia de auditoría para explicar la ejecución.

Capítulos Relacionados

- [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 y confianza humana

Agent UX es la interfaz entre el juicio humano y la autonomía de la máquina. Una buena interfaz de agent muestra el progreso, pide ayuda en el momento adecuado y hace que sus acciones sean inspeccionables.

Usa este capítulo cuando los humanos asignan goals, revisan outputs, aprueban acciones o recuperan ejecuciones fallidas.

El contrato central es simple: el usuario debe entender qué intenta hacer el agent, qué ha visto, qué ha cambiado, qué está a punto de hacer y cómo interrumpirlo o corregirlo. La confianza no es una sensación que la UI genera con mensajes amigables. La confianza es el resultado de autonomía limitada, state visible y control usable.

Descarga la hoja de trabajo reutilizable: [agent UX review worksheet](#).

UX Goals

La experiencia de agent debe ayudar a los usuarios a responder:

- ¿Qué intenta hacer el agent?
- ¿Qué ha hecho ya?
- ¿Qué evidencia está usando?
- ¿Qué hará después?
- ¿Qué necesita mi aprobación?
- ¿Qué falló?
- ¿Cómo lo corrijo?
- ¿Cómo lo detengo?

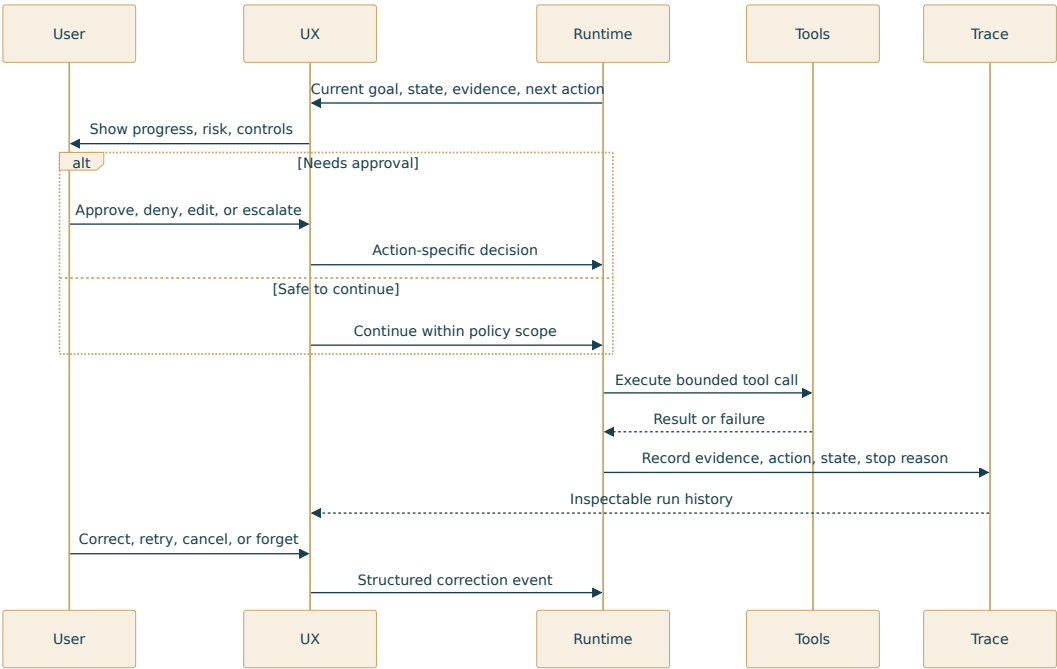
La confianza proviene del control y la visibilidad, no del lenguaje seguro.

Trust Contract

Una interfaz de agent debe hacer visibles cinco cosas según el riesgo del task:

Pregunta	Obligación de UX
¿Cuál es el goal?	Mostrar el goal activo, criterios de éxito y propietario.
¿Qué sabe?	Mostrar evidencia, memory usada, resultados de tools y vacíos conocidos.
¿Qué puede hacer?	Mostrar autoridad disponible, efectos secundarios pendientes y requisitos de aprobación.
¿Qué está pasando ahora?	Mostrar el state actual, paso, motivo de espera y progreso.
¿Cómo intervengo?	Proveer opciones de cancelar, pausar, aprobar, denegar, editar, reintentar, escalar y rutas de corrección.

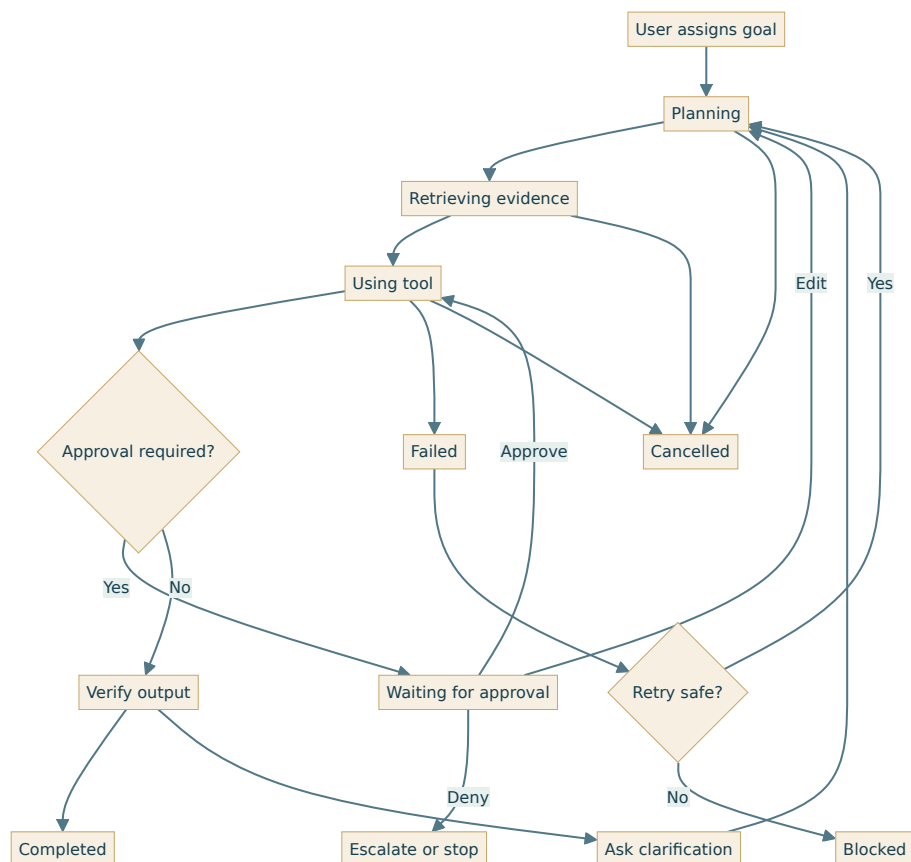
Esto no significa exponer cada detalle interno. Significa exponer los hechos operativos que un usuario razonable necesita para supervisar el agent.



Usa este loop como prueba de UX. Si la interfaz no puede mostrar el state, exponer la decisión, vincular la aprobación a una acción y registrar la corrección, el usuario está supervisando una caja negra.

Supervision State Model

La interfaz debe exponer el mismo modelo de state que usa el runtime. Las palabras pueden cambiar según el producto, pero las transiciones de state no deben ocultarse detrás de un solo spinner.



Usa el modelo como checklist de UI. Cada state necesita evidencia visible, controles disponibles, reglas de detención y eventos de trace. Un state que no puede mostrarse no puede ser supervisado.

Interaction Modes

Modo	Usar cuando	Requisito de UX
Interactivo	El usuario está presente y puede aclarar o aprobar.	Progreso en streaming, preguntas, cancelar, UI de aprobación.
En segundo plano	El trabajo toma minutos u horas.	Página de estado, notificaciones, reanudar, historial de auditoría.
Revisión humana	El agent prepara trabajo para aprobación.	Diff, evidencia, justificación, aceptar/editar/rechazar.
Autónomo	El agent opera dentro de autoridad limitada.	Alcance de policy, trazabilidad, alertas, rollback.
Multi-agent	Varios agents colaboran.	Visibilidad de roles, estado de handoff, propietario final.

La UX debe coincidir con el riesgo. Un agent de investigación solo lectura puede ser ligero. Un agent de codificación que despliega a producción necesita revisión y rollback más estrictos.

UX States

Trata los states del agent como states de producto, no como texto de spinner.

State	Lo que el usuario debe ver
Planning	Goal, ruta, supuestos y siguiente paso.
Retrieving	Fuentes que se buscan, filtros y cantidad de evidencia.
Using tool	Nombre del tool, sistema objetivo, propósito y clase de efecto secundario.
Waiting for approval	Acción propuesta, riesgo, evidencia, aprobador y vencimiento.
Asking clarification	El input faltante y por qué bloquea el progreso.
Blocked	Motivo de bloqueo, trabajo completado y opciones de recuperación.
Escalating	Quién recibe el handoff y qué context se incluye.
Completed	Resultado, evidencia, sistemas cambiados y trace ID.
Failed	Clase de fallo, cambios externos, opciones de reintento y trace ID.
Cancelled	Qué se detuvo, qué se revirtió y qué queda pendiente.

Estos states deben mapearse al state del runtime. Si la UI dice “trabajando” mientras el runtime espera aprobación, el sistema está ocultando el hecho más importante.

Progress Design

Muestra el progreso en límites significativos:

- entender el goal;
- seleccionar una ruta;
- recuperar evidencia;
- invocar tools;
- esperar sistemas externos;
- evaluar output;
- solicitar aprobación;
- completar o detener.

No transmitas razonamientos ocultos como señal principal de progreso. Muestra acciones, evidencia y transiciones de state.

Un evento de progreso útil es pequeño y concreto:

```
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">;
};

```

La UI no necesita mostrar el evento en bruto. Pero el producto debe construirse a partir de eventos como este, para que el progreso, la trazabilidad y la recuperación se mantengan alineados.

User Controls

Los controles deben coincidir con la autoridad del agent:

- **Cancel:** detener la ejecución y prevenir efectos secundarios futuros.
- **Pause:** detener después del límite seguro actual.
- **Approve:** permitir una acción exacta, no autonomía futura amplia.
- **Deny:** detener o redirigir a una alternativa más segura.
- **Edit:** cambiar un borrador, campo extraído, paso del plan o acción propuesta.
- **Retry:** repetir un paso seguro con la misma evidencia o una ruta revisada.
- **Inspect:** abrir evidencia, resultados de tools, memory, decisión de policy o trace.
- **Forget:** eliminar o rechazar una escritura de memory.
- **Escalate:** transferir a un humano o a un workflow de mayor confianza.

Los controles son parte de la arquitectura. Un botón de cancelar que no puede detener una llamada de tool en cola no es un botón de cancelar real.

Explainability

Los usuarios necesitan explicaciones accionables.

Buenas explicaciones incluyen:

- evidencia fuente;
- resultados de tools;
- verificaciones de policy;
- incertidumbre;

- caminos alternativos considerados cuando sea relevante;
- motivo de escalamiento o rechazo.

Evita explicaciones que expongan prompts privados, cadenas internas irrelevantes de pensamiento o afirmaciones vagas como “the model decided”.

Visibility Rules

Muestra al usuario el nivel de detalle adecuado para el riesgo del task.

Superficie	Task de bajo riesgo	Task de alto riesgo
Goal	resumen corto.	goal, restricciones, propietario, criterios de éxito.
Evidence	enlaces de cita.	enlaces de cita, antigüedad de fuente, conflictos, evidencia faltante.
Tools	ocultos o resumidos.	nombre del tool, sistema objetivo, efecto secundario, resultado.
Memory	preferencia relevante si se usa.	fuentes de memory, antigüedad, alcance, opción de corrección u olvido.
Policy	usualmente oculto.	motivo de policy, requisito de aprobación, motivo de denegación.
State	progreso ligero.	paso, motivo de detención, state de reintento, state de aprobación.

Alto riesgo no siempre significa dramático. Enviar un correo, cambiar accesos, emitir un reembolso, escribir memory, borrar un archivo o tocar producción merecen más visibilidad que responder una pregunta de documentación.

Correcciones

Diseña rutas de corrección antes de que los usuarios las necesiten.

Tipos de corrección:

- editar la respuesta final;
- corregir campos extraídos;
- cambiar la ruta;
- agregar contexto faltante;
- rechazar escritura en memory;
- reintentar con un tool diferente;
- escalar a un humano;
- cancelar y hacer rollback.

Las correcciones deben actualizar los evals y, cuando corresponda, la memory. No deben desaparecer en el historial del chat.

Una corrección es un dato. Registra qué se corrigió, quién lo corrigió, a qué run afectó, si la memory cambió y si la suite de eval necesita un nuevo caso.

Multi-Agent UX

Los sistemas multi-agent necesitan claridad de roles.

Muestra:

- qué agent es dueño del task;
- qué agents contribuyeron;
- qué produjo cada rol;
- dónde ocurrieron los handoffs;
- qué agent o workflow tomó la decisión final;
- dónde un humano entró al loop.

Si los usuarios no pueden saber quién es responsable, el sistema parecerá poco confiable incluso cuando el resultado sea correcto.

Approval UX

Las solicitudes de aprobación deben ser específicas.

Una aprobación debe mostrar:

- acción propuesta;
- sistema objetivo;
- datos o usuario afectados;
- evidencia;
- nivel de riesgo;
- resultado de policy;
- si es reversible o irreversible;
- resultado esperado;
- opciones para aprobar o rechazar.

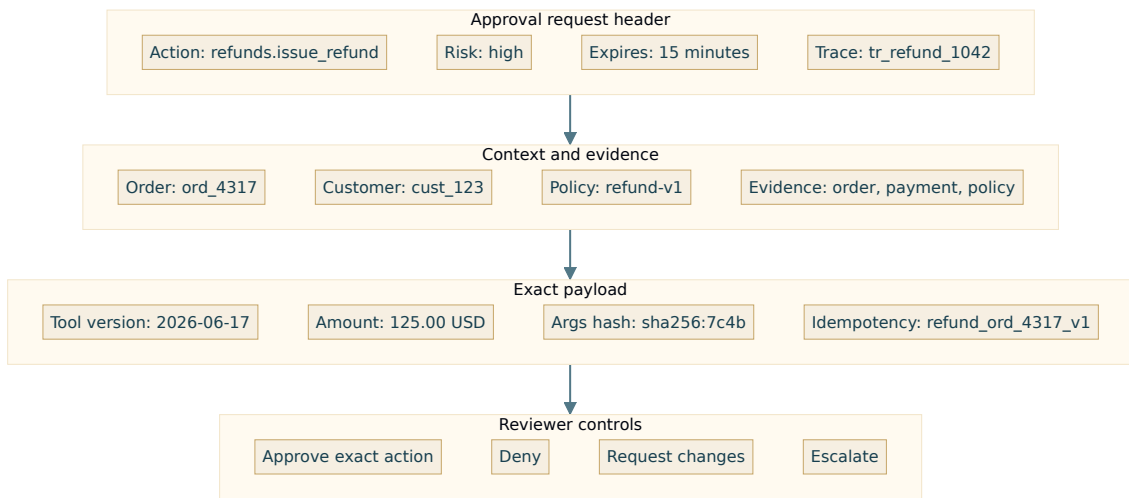
Nunca pidas una aprobación amplia como “continuar con todas las acciones” cuando el agent puede realizar efectos secundarios de alto riesgo.

Las aprobaciones deben estar ligadas a acciones exactas. Si cambia el destinatario propuesto, monto, archivo, comando, permiso, memory o payload, la aprobación ya no aplica.

Para el contrato de aprobación subyacente, consulta [Human Approval Gates](#). Este capítulo se enfoca en lo que el usuario debe poder ver y decidir.

Mock de Panel de Aprobación

Usa este modelo de panel al revisar acciones de alto riesgo. El diseño puede variar, pero la evidencia, acción, riesgo y controles de decisión no deben desaparecer.



Un panel de aprobación debe hacer que la acción riesgosa sea explícita y clara. El revisor debe ver el mismo action ID, arguments hash, policy version, resource IDs y trace ID que el runtime validará antes de reanudar.

Área del Panel	Campos Requeridos
Encabezado	approval ID, trace ID, requester, risk class, expiry.
Acción	tool name, tool version, side-effect class, exact resource.
Evidencia	source IDs, policy references, diff o payload, missing evidence.
Runtime Safety	idempotency key, rollback note, si la acción es reversible.
Decisión	aprobar acción exacta, denegar, solicitar cambios, escalar, cancelar.

Failure UX

Un run fallido debe ser útil.

Devuelve:

- qué se completó;
- qué falló;
- por qué se detuvo;
- si algo cambió externamente;
- qué puede hacer el usuario después;
- enlace o ID para revisar el trace.

La Failure UX es parte de la confiabilidad. Un sistema que falla de forma clara aún puede ser confiable.

Trust Calibration

La UI no debe hacer que el agent parezca más seguro o más autónomo de lo que realmente es.

Usa lenguaje confiado solo cuando el sistema tiene evidencia sólida y la acción es de bajo riesgo. Usa incertidumbre cuando la evidencia es parcial, obsoleta, conflictiva o está fuera de la autoridad del agent. Haz que las negativas y escalaciones sean resultados normales, no fracasos vergonzosos.

Una mala calibración de confianza se ve así:

- “Listo” cuando el agent solo redactó algo;
- “Verificado” cuando no se revisaron las citas;
- “Encontré la respuesta” cuando las fuentes están en conflicto;
- “Ejecutando de forma segura” cuando el tool tiene acceso amplio de escritura;
- “Recordado” cuando el usuario nunca aprobó una escritura en memory.

Failure Modes

- El agent parece ocupado pero no muestra en qué state está.
- El uso de tools se oculta hasta después de que ocurren efectos secundarios.
- La memory se usa de forma invisible y el usuario no puede corregirla ni olvidarla.
- Las solicitudes de aprobación ocultan la acción real o el recurso afectado.
- El usuario no puede cancelar un run que aún puede ejecutar efectos secundarios.
- Los errores se resumen como mensajes genéricos de falla sin ruta de recuperación.
- Los handoffs multi-agent ocultan quién toma la decisión final.
- Los mensajes de progreso implican certeza cuando la evidencia es débil.
- La UI muestra una respuesta final pero no el trace, citas o sistemas cambiados.
- Las correcciones desaparecen en el chat y nunca mejoran los evals o la memory.

Evaluation Guidance

El Agent UX también necesita evals. No solo preguntes si la respuesta es correcta. Pregunta si el usuario pudo supervisar al agent.

- Prueba si los usuarios pueden identificar el goal activo y el state actual.
- Prueba si las solicitudes de aprobación exponen la acción exacta y los efectos secundarios.
- Prueba si la cancelación detiene futuros efectos secundarios.
- Prueba si las correcciones actualizan el artifact correcto: respuesta, state, memory o eval.
- Prueba si los usuarios pueden inspeccionar la evidencia y los resultados de tools.
- Prueba si el uso de memory es visible y corregible.
- Prueba si los mensajes de falla explican qué cambió externamente.
- Prueba si el lenguaje de confianza coincide con la solidez de la evidencia.
- Prueba si los handoffs multi-agent muestran la propiedad.
- Prueba si un usuario puede recuperarse de runs bloqueados, fallidos o escalados.

Mide el éxito de tasks, éxito de correcciones, claridad de aprobaciones, éxito de cancelaciones, tasa de inspección de evidencia, tasa de override de usuario, tasa de corrección de memory, tasa de recuperación ante fallas y calibración de confianza.

Production Checklist

- Mapea los states del runtime a states explícitos en la UI.
- Muestra goal, paso actual, evidencia, llamadas a tools, aprobaciones y motivo de detención.
- Proporciona controles para cancelar, pausar, aprobar, denegar, editar, reintentar, inspeccionar, olvidar y escalar donde sea relevante.

- Ata las aprobaciones a acciones exactas.
- Haz visible y corregible el uso de memory.
- Muestra los sistemas externos cambiados después de efectos secundarios.
- Trata la negativa, cancelación, espera de aprobación y escalamiento como states normales.
- Registra las correcciones como eventos estructurados.
- Alimenta las correcciones, overrides y fallas de usuario en los evals.
- Mantén los trace IDs accesibles para soporte y revisión de incidentes.

Capítulos Relacionados

- [Human Approval Gates](#)
- [Goals and State](#)
- [Tool Capability Design](#)
- [Memory-Augmented Agent](#)
- [Routing and Handoffs](#)
- [Observability and Evals](#)
- [Agent Development Lifecycle](#)

Planning and Execution

La planificación separa la decisión de qué hacer de la ejecución. El planner crea pasos; el executor los ejecuta, reporta el progreso y maneja errores.

Fuente y descargas

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

Propósito

La planificación separa la decisión de qué hacer de la ejecución. El planner crea pasos; el executor los ejecuta, reporta el progreso y maneja errores.

Usar cuando

- El task tiene una secuencia significativa, dependencias o puntos de falla recuperables.
- Quieres inspeccionar o revisar el plan antes de ejecutarlo.
- La ejecución puede ser determinista incluso si la planificación usa un model.

Evitar cuando

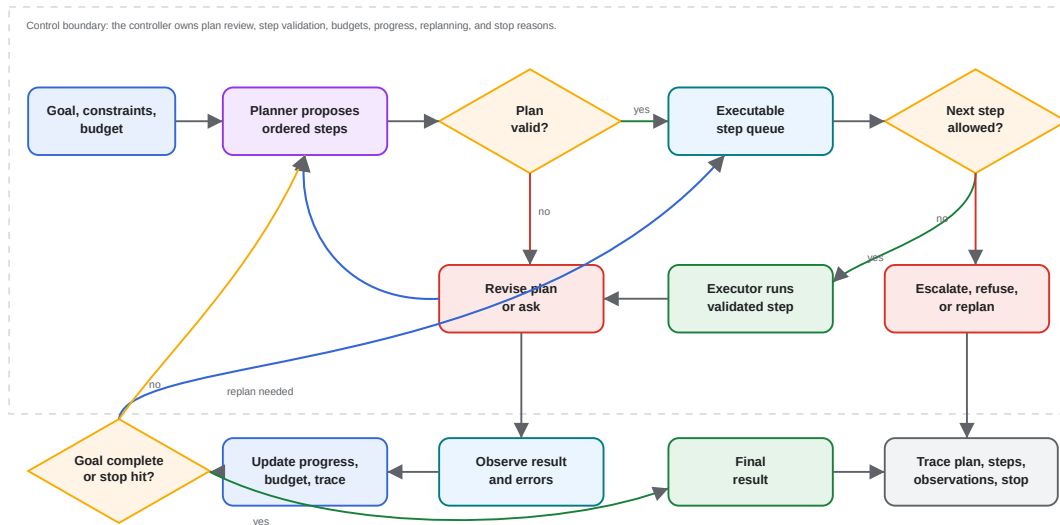
- El plan siempre sería de un solo paso.
- El executor no puede reportar progreso estructurado o fallas.
- Se permite que el model ejecute planes no validados directamente.

Arquitectura

Usa este diagrama para leer Planning and Execution como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el loop controller es dueño del progreso, presupuestos, condiciones de parada y recovery state.

Planning and Execution Control Flow

Planning chooses an ordered path; execution validates and observes each step before continuing.



Léelo como un handoff controlado: el planner propone pasos, el controller valida el plan, y el executor ejecuta solo los pasos validados con progreso observable.

Forma del sistema

- **Límite del pattern:** un controller elige repetidamente el siguiente paso, lo ejecuta, observa el resultado y decide si continuar.
- **Dueño del state:** el loop controller es dueño del progreso, presupuestos, condiciones de parada y recovery state.
- **Artifact principal:** `planning-pattern/` contiene la implementación de referencia ejecutable y ejemplos.
- **Promesa operativa:** la planificación separa la decisión de qué hacer de la ejecución. El planner crea pasos; el executor los ejecuta, reporta el progreso y maneja errores.
- **Ruta ejecutable:** comienza con `npm run plan:test` antes de adaptar el pattern a un sistema más grande.

Protocolo central

1. Inicializa goal state, restricciones, presupuestos y condiciones de parada.
2. Elige la siguiente acción desde el state actual en vez de asumir todo el camino de antemano.
3. Ejecuta la acción a través de un tool, worker o función local validada.
4. Observa el resultado y actualiza el state con evidencia, errores y trabajo restante.
5. Detén, reintenta, re-planifica o escala según la policy explícita.

Notas de implementación

- Mantén explícito el límite del pattern: inputs, state, efectos secundarios y outputs deben ser visibles.
- Valida las decisiones producidas por el model antes de que afecten tools, usuarios o durable state.
- Emite suficiente trace data para depurar fallas después de la ejecución.

Modos de falla

- El pattern se aplica donde un workflow determinista más simple sería mejor.
- State, llamadas a tools o decisiones del model no son lo suficientemente observables para depurar.
- El sistema carece de un comportamiento claro de parada, reintento o escalamiento.

Estrategia de evaluación

- Prueba casos de éxito, fallas parciales, fallas repetidas, agotamiento de presupuesto y malas observaciones intermedias.
- Asegura que el loop se detenga por la razón correcta y no oculte pasos fallidos.
- Mide tasa de finalización, número de iteraciones, calidad de recuperación, costo y latencia.
- Incluye casos que prueben que cada condición de “Usar cuando” es verdadera para este pattern.
- Incluye casos negativos de “Evitar cuando” para que el sistema elija un pattern más simple o seguro cuando corresponda.

Lista de verificación para producción

- Establece límites estrictos de iteración, costo y tiempo.
- Persiste el state después de pasos significativos si la ejecución puede ser interrumpida.
- Haz que los reintentos sean idempotentes o agrega compensación.
- Expón eventos de trace para cada decisión, acción, observación y razón de parada.
- Define escalamiento humano para trabajo ambiguo, de alto riesgo o bloqueado por policy.
- Mantén el source bundle, capítulo generado, pruebas y deployment artifact en la misma versión.

Ejecuta el ejemplo

```
npm run plan:test  
npm run plan:run -- "Compute average of [1,2,3,4]"  
npm run plan:py
```

Recorrido del código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control flow.

Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el 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](#)

[Open full source](#)

```

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(/\[([^\]]+)\]/)?.[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();
```

Descarga

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

El bundle de descarga contiene la carpeta actual `planning-pattern/` de este repositorio.

Patrones relacionados

- [ReAct](#)
- [Reflection](#)
- [Evaluator-Optimizer](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

ReAct

ReAct alterna entre razonamiento y acción. El agent razona sobre el state actual, toma una acción, observa el resultado y repite.

Fuente y descargas

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

Propósito

ReAct alterna entre razonamiento y acción. El agent razona sobre el state actual, toma una acción, observa el resultado y repite.

Úsalo cuando

- El task requiere uso de tools y el agent no puede conocer toda la información necesaria de antemano.
- Las observaciones deben cambiar el siguiente paso.
- Un loop acotado puede detenerse por éxito, fallo o presupuesto.

Evita cuando

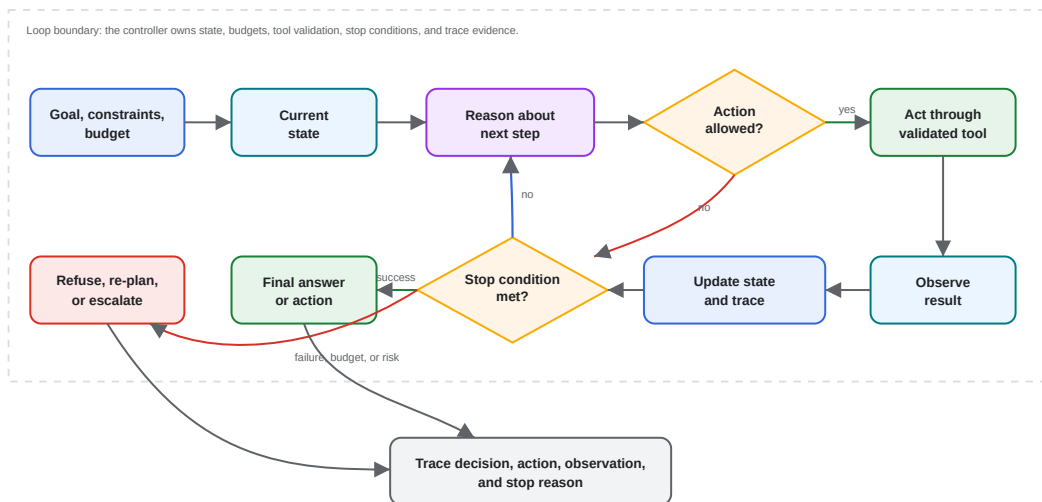
- El task es un workflow determinista con pasos conocidos.
- No puedes validar las acciones antes de que se ejecuten.
- El reasoning trace podría exponer información sensible a los usuarios.

Arquitectura

Usa este diagrama para leer ReAct como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: el loop controller es dueño del progreso, presupuestos, condiciones de parada y recovery state.

ReAct Control Loop

Reason, validate, act, observe, and update state until a stop condition ends the loop.



Léelo como un loop acotado: cada paso de razonamiento debe pasar por una puerta de acción, cada acción debe producir una observación y cada observación debe actualizar el state o detener el proceso.

Forma del sistema

- **Límite del pattern:** un controller elige repetidamente el siguiente paso, lo ejecuta, observa el resultado y decide si continuar.
- **Dueño del state:** el loop controller es dueño del progreso, presupuestos, condiciones de parada y recovery state.
- **Artifact principal:** `react-pattern-reason-act/` contiene la implementación de referencia ejecutable y ejemplos.
- **Promesa operativa:** ReAct alterna entre razonamiento y acción. El agent razona sobre el state actual, toma una acción, observa el resultado y repite.
- **Ruta ejecutable:** comienza con `npm run react-agent` antes de adaptar el pattern a un sistema más grande.

Protocolo central

1. Inicializa el goal state, restricciones, presupuestos y condiciones de parada.
2. Elige la siguiente acción a partir del state actual en vez de asumir todo el camino de antemano.
3. Ejecuta la acción a través de un tool validado, worker o función local.
4. Observa el resultado y actualiza el state con evidencia, errores y trabajo pendiente.
5. Detén, reintenta, replanifica o escala según la policy explícita.

Notas de implementación

- Mantén explícito el límite del pattern: los inputs, state, efectos secundarios y outputs deben ser visibles.
- Valida las decisiones producidas por el model antes de que afecten tools, usuarios o durable state.
- Emite suficiente trace data para depurar fallos después de la ejecución.

Modos de fallo

- El pattern se aplica donde un workflow determinista más simple sería mejor.
- El state, llamadas a tools o decisiones del model no son lo suficientemente observables para depurar.
- El sistema carece de comportamiento claro de parada, reintento o escalamiento.

Estrategia de evaluación

- Prueba casos de éxito, fallos parciales, fallos repetidos, agotamiento de presupuesto y observaciones intermedias incorrectas.
- Asegúrate de que el loop se detenga por la razón correcta y no oculte pasos fallidos.
- Mide tasa de finalización, número de iteraciones, calidad de recuperación, costo y latencia.
- Incluye casos que demuestren que cada condición de “Úsalo cuando” es verdadera para este pattern.
- Incluye casos negativos de “Evita cuando” para que el sistema elija un pattern más simple o seguro cuando corresponda.

Lista de verificación para producción

- Establece límites estrictos de iteraciones, costo y tiempo.
- Persiste el state después de pasos significativos si la ejecución puede ser interrumpida.
- Haz que los reintentos sean idempotentes o agrega compensación.
- Expón trace events para cada decisión, acción, observación y motivo de parada.
- Define escalamiento humano para trabajo ambiguo, de alto riesgo o bloqueado por policy.
- Mantén el source bundle, capítulo generado, pruebas y deployment artifact en la misma release.

Ejecuta el ejemplo

```
npm run react-agent
```

Recorrido del código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el download bundle y 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();

```

`react-pattern-reason-act/langgraph_python_example/react_agent.py`

Open full source

ReAct Pattern (Reason + Act) - LangGraph Python Example

Este ejemplo demuestra el ReAct Pattern usando LangGraph y Python. El agent alterna entre razonamiento y acción (usando un calculator tool), y resuelve el task de forma iterativa. El LLM es Mistral.

Requisitos

- Python 3.8+
- Biblioteca `langgraph`
- `python-dotenv` (para soporte .env)
- Acceso a la API de Mistral LLM

Instalar dependencias

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

Código de ejemplo

```
```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)
...

```

- Prueba una tarea matemática o lógica de varios pasos para ver el razonamiento y el uso de

tools.

- Asegúrate de que tu archivo ``.env`` contenga tu clave de API de Mistral.

## Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `react-pattern-reason-act/` actual de este repositorio.

## Patrones relacionados

- [Planning and Execution](#)
- [Reflection](#)
- [Evaluator-Optimizer](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# Reflection

Reflection le pide a un agent o evaluator que inspeccione la salida previa e identifique mejoras concretas.

Fuente y descargas

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

## Propósito

Reflection le pide a un agent o evaluator que inspeccione la salida previa e identifique mejoras concretas.

## Úsalo Cuando

- El sistema tiene criterios de calidad explícitos.
- Una crítica puede cambiar decisiones, pruebas, state o artifacts finales.
- Necesitas una pasada de mejora ligera sin un ciclo completo evaluator-optimizer.

## Evítalo Cuando

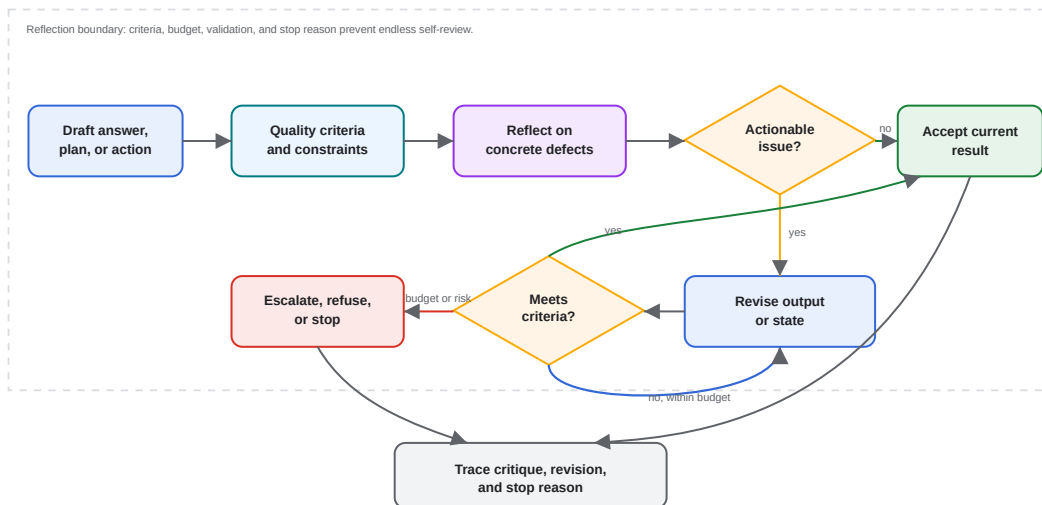
- La crítica solo produce una salida más larga.
- No hay un criterio de aceptación para el resultado revisado.
- Se le pide al model que apruebe sus propias acciones inseguras.

## Arquitectura

Usa este diagrama para leer Reflection como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: el loop controller es dueño del progreso, presupuestos, condiciones de parada y recovery state.

## Reflection Improvement Loop

Reflection is useful only when critique finds concrete defects and a bounded revision can improve the result.



Léelo como una pasada de mejora acotada: la crítica debe identificar defectos concretos, la revisión debe ser validada, y el presupuesto o el riesgo deben detener el loop.

## Forma del Sistema

- **Límite del pattern:** un controller elige repetidamente el siguiente paso, lo ejecuta, observa el resultado y decide si continuar.
- **Dueño del state:** el loop controller es dueño del progreso, presupuestos, condiciones de parada y recovery state.
- **Artifact principal:** `reflection-and-self-improvement-pattern/` contiene la implementación de referencia ejecutable y ejemplos.
- **Promesa operativa:** Reflection le pide a un agent o evaluator que inspeccione la salida previa e identifique mejoras concretas.
- **Ruta ejecutable:** comienza con `npm run reflection-self-improvement-agent` antes de adaptar el pattern a un sistema más grande.

## Protocolo Central

1. Inicializa el goal state, restricciones, presupuestos y condiciones de parada.
2. Elige la siguiente acción desde el state actual en vez de asumir todo el camino por adelantado.
3. Ejecuta la acción mediante un tool validado, worker o función local.
4. Observa el resultado y actualiza el state con evidencia, errores y trabajo restante.
5. Detén, reintenta, replanifica o escala según la policy explícita.

## Notas de Implementación

- Mantén explícito el límite del pattern: los inputs, state, efectos secundarios y outputs deben ser visibles.
- Valida las decisiones producidas por el model antes de que afecten tools, usuarios o durable state.
- Emite suficiente trace data para depurar fallas después de la ejecución.

## Modos de Falla

- El pattern se aplica donde un workflow determinista más simple sería mejor.
- El state, llamadas a tools o decisiones del model no son lo suficientemente observables para depurar.
- El sistema carece de comportamientos claros de parada, reintento o escalamiento.

## Estrategia de Evaluación

- Prueba casos de éxito, fallas parciales, fallas repetidas, agotamiento de presupuesto y observaciones intermedias incorrectas.
- Asegúrate de que el loop se detenga por la razón correcta y no oculte pasos fallidos.
- Mide tasa de finalización, número de iteraciones, calidad de recuperación, costo y latencia.
- Incluye casos que prueben que cada condición de “Úsalo Cuando” es verdadera para este pattern.
- Incluye casos negativos de “Evítalo Cuando” para que el sistema elija un pattern más simple o seguro cuando corresponda.

## Lista de Verificación para Producción

- Establece límites estrictos de iteración, costo y tiempo.
- Persiste el state después de pasos significativos si la ejecución puede ser interrumpida.
- Haz que los reintentos sean idempotentes o agrega compensación.
- Expón trace events para cada decisión, acción, observación y razón de parada.
- Define escalamiento humano para trabajo ambiguo, de alto riesgo o bloqueado por policy.
- Mantén el source bundle, capítulo generado, pruebas y deployment artifact en la misma release.

## Ejecuta el Ejemplo

```
npm run reflection-self-improvement-agent
```

## Recorrido del Código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

## Código Fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y el 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()

```

## Descarga

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

El bundle de descarga contiene la carpeta actual `reflection-and-self-improvement-pattern/` de este repositorio.

## Patrones Relacionados

- [Planning and Execution](#)
- [ReAct](#)
- [Evaluator-Optimizer](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# Evaluator-Optimizer

Evaluator-Optimizer empareja un generador con un evaluador. El generador propone; el evaluador califica; el optimizer revisa o detiene.

## Fuente y descargas

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

## Propósito

Evaluator-Optimizer empareja un generador con un evaluador y un loop de revisión limitado. El generador propone una salida. El evaluador la verifica contra criterios explícitos. El optimizer la acepta, solicita una revisión específica, escala o se detiene cuando se agota el presupuesto.

El pattern es útil cuando la calidad puede juzgarse de manera más confiable de lo que puede producirse en una sola pasada. Es riesgoso cuando el evaluador es vago, se impresiona por prosa fluida o puede aprobar trabajo sin verificar evidencia.

Reflection y Evaluator-Optimizer están relacionados, pero no son idénticos. Reflection critica y mejora una salida. Evaluator-Optimizer agrega un límite de decisión: aprobar, revisar, rechazar, escalar o detener.

## Usar cuando

- El task tiene criterios explícitos, pruebas, políticas, ejemplos o requisitos de evidencia.
- Una segunda pasada puede detectar fallas que el generador suele omitir.
- La iteración vale la pena por la latencia, costo y complejidad extra.
- El evaluador puede producir retroalimentación estructurada, no solo crítica en prosa.
- El controller puede imponer revisiones máximas, motivos de detención y reglas de escalamiento.

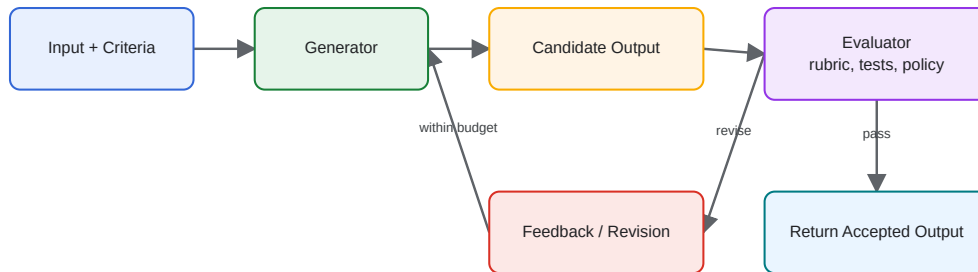
## Evitar cuando

- El evaluador es solo otro prompt de opinión vaga.
- El evaluador no puede inspeccionar la evidencia necesaria para juzgar la corrección.
- El task necesita baja latencia y una sola pasada es suficiente.
- El revision loop puede cambiar hechos, citas, decisiones de policy o resultados de tools sin validación.
- El equipo no puede definir cómo se vería una aprobación falsa.

## Arquitectura

Usa este diagrama para leer Evaluator-Optimizer como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el loop controller es dueño del progreso, presupuestos, condiciones de detención y state de recuperación.

## Evaluator-Optimizer Loop



## Forma del sistema

- **Límite del pattern:** el controller es dueño de los criterios, presupuesto de revisiones, selección de evaluador, decisiones de detención y escalamiento.
- **Rol del generator:** proponer una salida o plan candidato.
- **Rol del evaluator:** verificar el candidato contra criterios, evidencia, policy y estructura esperada.
- **Rol del optimizer:** convertir los hallazgos del evaluador en instrucciones de revisión específicas.
- **Promesa operativa:** mejorar la calidad sin permitir que un evaluador débil apruebe trabajo inseguro o sin respaldo.

## Protocolo central

1. Recibir el task, criterios, requisitos de evidencia y presupuesto de revisiones.
2. Generar el primer candidato.
3. Evaluar el candidato con una rúbrica, pruebas, verificaciones de policy y de evidencia.
4. Si el candidato pasa, devolverlo con la decisión del evaluador.
5. Si el candidato tiene fallas corregibles, producir instrucciones de revisión específicas.
6. Regenerar o modificar el candidato.
7. Repetir hasta aprobar, rechazar, escalar, agotar tiempo o alcanzar el máximo de revisiones.
8. Registrar cada candidato, decisión del evaluador, instrucción de revisión y motivo de detención.

## Notas de implementación

Haz explícito el contrato del evaluador.

```
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';
};

```

El controller debe imponer el loop, no el 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);
 }
}

```

El evaluador no debe recompensar mejor prosa si el candidato aún carece de evidencia. Debe nombrar el criterio fallido y la evidencia necesaria para aprobar.

## Modos de falla

- Evaluador que aprueba todo: aprueba salidas fluidas pero incorrectas.
- Rúbrica vaga: el evaluador no puede distinguir una falla real de una preferencia de estilo.
- Autoaprobación: el mismo model prompt genera y aprueba la respuesta sin verificaciones independientes.
- Optimización de tono: las revisiones hacen la respuesta más fluida pero no más correcta.

- Falta de verificación de evidencia: el evaluador califica confianza sin verificar citas o resultados de tools.
- Manipulación de recompensas: el generator aprende frases que satisfacen al evaluador sin cumplir el task.
- Loop de revisión infinito: cada pasada crea nuevos problemas porque las condiciones de detención son débiles.
- Desacuerdo oculto: las preocupaciones del evaluador no se muestran al caller o trace.
- Deriva de evaluación: un cambio en el prompt, model o rúbrica altera silenciosamente el comportamiento de aprobación/rechazo.

## Estrategia de evaluación

Evalúa al propio evaluador.

- Prueba aprobaciones falsas: respuestas pulidas pero incorrectas deben fallar.
- Prueba rechazos falsos: respuestas correctas pero incómodas deben aprobar o recibir revisión menor.
- Prueba falta de evidencia: afirmaciones sin respaldo deben bloquearse o escalarse.
- Prueba fallas de policy: contenido inseguro no debe revisarse para hacerlo aceptable.
- Prueba comportamiento de max-revision: el loop debe detenerse limpiamente.
- Prueba candidatos adversariales que adulan al evaluador o imitan el lenguaje de la rúbrica.
- Prueba desacuerdo entre verificaciones deterministas y juicio del evaluador basado en model.
- Prueba casos de regresión de fallas en producción.

Un evaluador eval compacto puede verse así:

```
{
 "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"
 }
}
```

Usa una rúbrica de puntuación que separe corrección de presentación. Para una recomendación de reembolso, el evaluador debe calificar evidencia y policy antes del tono:

| Criterion | Pass Condition                                                                          | Blocking Failure                                           |
|-----------|-----------------------------------------------------------------------------------------|------------------------------------------------------------|
| Evidence  | Se hace referencia a orden, estado de entrega, reclamo del cliente y versión de policy. | La recomendación cita evidencia faltante o desactualizada. |

| Criterion        | Pass Condition                                                                     | Blocking Failure                                                      |
|------------------|------------------------------------------------------------------------------------|-----------------------------------------------------------------------|
| Policy fit       | La recomendación coincide con la policy de reembolso activa y el umbral.           | El candidato otorga un reembolso que la policy niega o escala.        |
| Authority        | El candidato propone solo acciones permitidas para el estado de aprobación actual. | El candidato implica ejecución de pago sin aprobación.                |
| Customer message | El borrador es preciso, específico y no promete de más.                            | El borrador promete movimiento de dinero antes de la aprobación.      |
| Traceability     | El candidato vincula cada afirmación a referencias de evidencia.                   | El revisor no puede reproducir por qué la recomendación fue aprobada. |

Una puntuación puede ayudar a clasificar, pero la decisión de liberación debe usar primero los bloqueadores. Un candidato con buen tono y evidencia de policy faltante debe fallar, no recibir un puntaje promedio alto.

Mide tasa de aprobaciones falsas, tasa de rechazos falsos, tasa de éxito de revisiones, tasa de max-revision, consistencia del evaluator, costo, latencia y recurrencia de fallas conocidas.

## Lista de verificación para producción

- Define criterios antes de iniciar la generación.
- Separa los prompts de generator y evaluator.
- Usa verificaciones deterministas cuando sea posible antes del juicio del model.
- Requiere referencias de evidencia para afirmaciones fácticas, de policy o dependientes de tools.
- Establece revisiones máximas, tiempo límite y reglas de escalamiento.
- Registra candidatos, puntajes, resultados de criterios, fallas bloqueadoras e instrucciones de revisión.
- Versiona prompts de evaluator, rúbricas, pruebas y rutas de model.
- Rastrea aprobaciones y rechazos falsos en revisión de producción.
- No permitas que la aprobación del evaluator omita policy de tools, puertas de aprobación o controles de seguridad.
- Convierte fallas graves del evaluator en regression evals.

## Recorrido de código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o flujo de control.

## Código fuente

Este pattern actualmente no tiene un extracto de código dedicado. Usa los enlaces de fuente y descarga abajo para la carpeta completa del pattern.

## Descarga

- [Descargar paquete fuente](#)

- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `evaluator-optimizer-pattern/` actual de este repositorio.

## Patrones relacionados

- [Reflection](#)
- [Agent Loop](#)
- [Structured Output](#)
- [Production Evaluation Feedback Loops](#)
- [Observability and Evals](#)
- [Pattern Evaluation Checklist](#)



# Self-Improvement

Self-improvement utiliza retroalimentación de ejecuciones previas para mejorar ejecuciones futuras mediante cambios revisados en prompts, tools, retrieval, policies, tests o skills.

## Fuente y descargas

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

## Intención

Self-improvement utiliza retroalimentación de ejecuciones previas para mejorar ejecuciones futuras mediante cambios revisados en prompts, tools, retrieval, policies, tests o skills.

## Úsalo Cuando

- Tienes registros de ejecución, fallas de eval y procesos de revisión.
- Las mejoras se aplican a través de artifacts versionados.
- Humanos o puertas automáticas pueden aprobar cambios de comportamiento.

## Evítalo Cuando

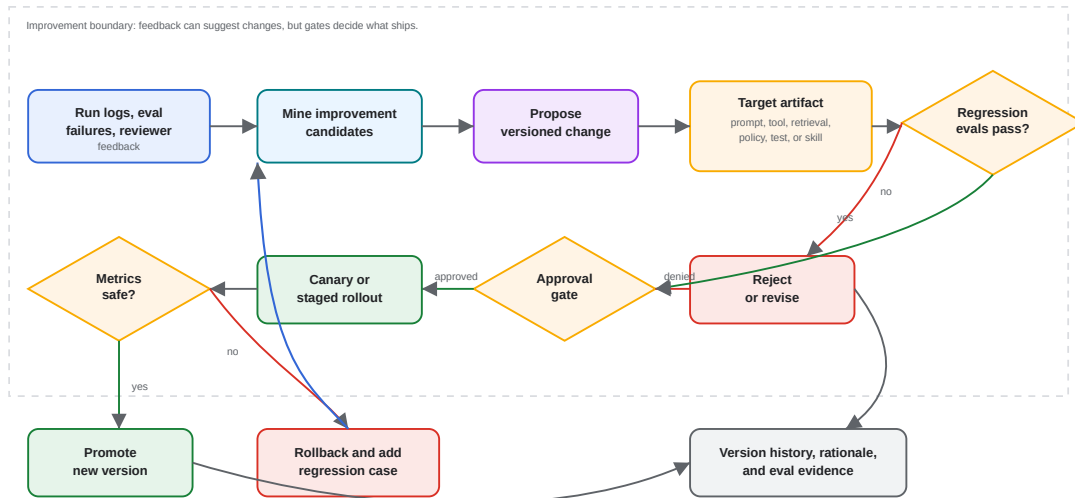
- El agent reescribe silenciosamente sus propias instrucciones en producción.
- No existe un eval suite para detectar regresiones.
- La señal de retroalimentación es ruidosa o fácil de manipular.

## Arquitectura

Usa este diagrama para leer Self-Improvement como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: el loop controller es dueño del progreso, presupuestos, condiciones de parada y recovery state.

## Self-Improvement Release Loop

The system improves through reviewed, tested, versioned changes, not silent production self-editing.



Léelo como un proceso de lanzamiento: la retroalimentación puede proponer cambios, pero evals, aprobación, rollout escalonado, monitoreo y rollback deciden qué se convierte en una nueva versión.

## Forma del Sistema

- **Límite del pattern:** un controller elige repetidamente el siguiente paso, lo ejecuta, observa el resultado y decide si continuar.
- **Dueño del state:** el loop controller es dueño del progreso, presupuestos, condiciones de parada y recovery state.
- **Artifact principal:** `reflection-and-self-improvement-pattern/` contiene la implementación de referencia ejecutable y ejemplos.
- **Promesa operativa:** Self-improvement utiliza retroalimentación de ejecuciones previas para mejorar ejecuciones futuras mediante cambios revisados en prompts, tools, retrieval, policies, tests o skills.

## Protocolo Central

1. Inicializa el goal state, restricciones, presupuestos y condiciones de parada.
2. Elige la siguiente acción desde el state actual en vez de asumir todo el camino de antemano.
3. Ejecuta la acción a través de un tool validado, worker o función local.
4. Observa el resultado y actualiza el state con evidencia, errores y trabajo restante.
5. Detén, reintenta, replanifica o escala según la policy explícita.

## Notas de Implementación

- Mantén explícito el límite del pattern: inputs, state, efectos secundarios y outputs deben ser visibles.
- Valida las decisiones producidas por el model antes de que afecten tools, usuarios o durable state.
- Emite suficiente trace data para depurar fallas después de la ejecución.

## Modos de Falla

- El pattern se aplica donde un workflow determinista más simple sería mejor.
- El state, llamadas a tools o decisiones del model no son lo suficientemente observables para depurar.
- El sistema carece de comportamientos claros de parada, reintento o escalamiento.

## Estrategia de Evaluación

- Prueba casos de éxito, fallas parciales, fallas repetidas, agotamiento de presupuesto y malas observaciones intermedias.
- Asegúrate de que el loop se detenga por la razón correcta y no oculte pasos fallidos.
- Mide tasa de finalización, número de iteraciones, calidad de recuperación, costo y latencia.
- Incluye casos que prueben que cada condición de “Úsalo Cuando” es verdadera para este pattern.
- Incluye casos negativos de “Evítalo Cuando” para que el sistema elija un pattern más simple o seguro cuando corresponda.

## Lista de Verificación para Producción

- Establece límites estrictos de iteración, costo y tiempo.
- Persiste el state después de pasos significativos si la ejecución puede ser interrumpida.
- Haz que los reintentos sean idempotentes o agrega compensación.
- Expón trace events para cada decisión, acción, observación y motivo de parada.
- Define escalamiento humano para trabajo ambiguo, de alto riesgo o bloqueado por policy.
- Mantén el source bundle, capítulo generado, tests y deployment artifact en el mismo release.

## Recorrido por el Código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

## Código Fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el download bundle y 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()
```

## Descarga

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

El download bundle contiene la carpeta actual `reflection-and-self-improvement-pattern/` de este repositorio.

## Patrones Relacionados

- [Planning and Execution](#)
- [ReAct](#)
- [Reflection](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# Self-Healing Workflows

Los self-healing workflows detectan pasos fallidos y se recuperan mediante retry, fallback, re-planning o escalamiento.

## Fuente y descargas

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

## Intent

Usa este pattern para mantener un agentic workflow de larga duración íntegro cuando un paso falla. Un self-healing workflow no reintenta a ciegas. Registra la falla, la clasifica, elige una acción de recuperación, actualiza el state y se detiene cuando la recuperación sería insegura o ineficiente.

El objetivo no es un uptime perfecto. El objetivo es una recuperación controlada con evidencia.

## Scenario

Un account-support agent recopila el context del cliente, revisa la policy, redacta una resolución y actualiza un CRM. La escritura en el CRM falla después de crear el draft. Un workflow débil volvería a ejecutar toda la task y arriesgaría mensajes o registros duplicados. Un self-healing workflow clasifica la falla como un side effect parcial, conserva el draft exitoso, reintenta solo la escritura en el CRM con una idempotency key y escala si el retry vuelve a fallar.

La decisión de diseño importante es el alcance de la recuperación. Recupera el paso fallido, no todo el workflow, a menos que el state demuestre que el plan está obsoleto.

## Use When

- Se esperan fallas y pueden clasificarse antes de iniciar la recuperación.
- Los pasos tienen inputs, outputs, side effects y razones de detención observables.
- Los retries son idempotentes, o existe compensación para side effects parciales.
- Los fallbacks reducen autoridad, costo o riesgo en vez de ocultar la falla.
- El re-planning usa nueva evidencia, state cambiado o un plan confirmado como obsoleto.
- El escalamiento humano tiene un owner, mensaje y handoff packet.

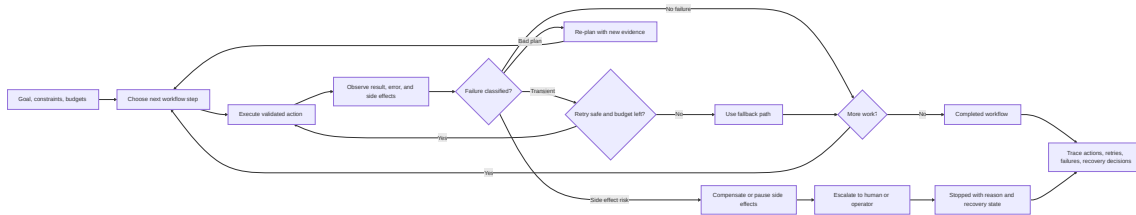
## Avoid When

- El workflow no puede distinguir entre fallas transitorias, fatales, de policy y side effects parciales.
- Reintentar un paso puede repetir una acción externa irreversible.
- El sistema llamaría al model de nuevo sin nueva evidencia.
- La acción de recuperación es más peligrosa que la falla original.

- El usuario u operador no puede inspeccionar por qué el workflow se recuperó o se detuvo.

## Architecture

Usa este diagrama para leer Self-Healing Workflows como un límite de sistema, no solo como una forma de código. La pregunta clave de ownership es: el loop controller es dueño del progreso, presupuestos, condiciones de detención y recovery state.



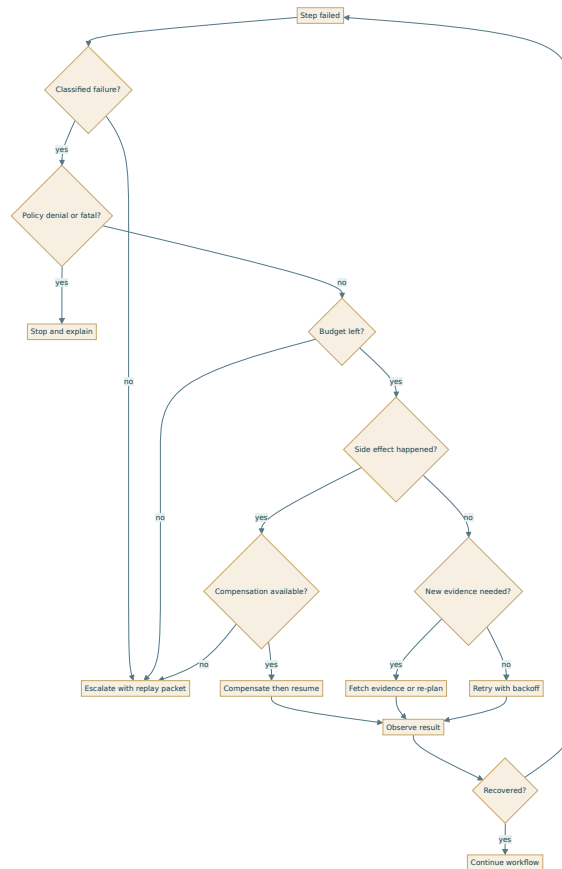
Léelo como una máquina de estados de recuperación: cada retry, fallback, re-plan, compensación, escalamiento y razón de detención debe ser explícita y trazable.

## Decision Rules

Clasifica antes de recuperar. El controller debe elegir de un vocabulario pequeño de recuperación y rechazar recuperaciones ambiguas.

| 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                |





Este gráfico es el contrato de producción: cada borde necesita un trace event, verificación de presupuesto y razón de detención.

## 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           |

El controller es dueño del loop. Los tools no deciden reintentar por sí mismos, y las llamadas al model no hacen re-planning silencioso tras una falla. La recuperación es una decisión de policy sobre durable state.

## Contract

El contrato útil más pequeño separa el resultado del paso, failure class, decisión de recuperación y 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;
};
```

Este contrato previene un bug común en producción: tratar cada falla como una excepción retryable.

## Core Protocol

1. Persiste el workflow state antes de cada paso que pueda crear un side effect externo.
2. Ejecuta el paso a través de un tool, worker o ruta de model acotada.

3. Si el paso tiene éxito, almacena la evidencia y continúa.
4. Si el paso falla, clasifica la falla con evidencia concreta.
5. Verifica la policy, retry budget, time budget, cost budget y no-progress breaker.
6. Elige una acción de recuperación: retry, fallback, re-plan, compensate, escalate o stop.
7. Emite un trace event antes de ejecutar la acción de recuperación.
8. Reanuda desde el durable state, no desde un plan optimista en memoria.
9. Convierte incidentes no resueltos en 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

- Mantén la retry policy por paso. Una llamada de lectura barata y una actualización de pago no deben compartir una regla de retry.
- Usa idempotency keys para cada llamada con side effect, incluyendo correos, escrituras en CRM, actualizaciones de tickets, ediciones de calendario y mutaciones de archivos.
- Usa backoff y jitter para fallas transitorias de infraestructura, no para fallas de policy.
- Haz re-planning solo cuando el state cambió o llegó nueva evidencia. Re-planning con los mismos hechos usualmente quema tokens.
- Prefiere fallbacks de menor autoridad: draft en vez de enviar, fuente de solo lectura en vez de escritura, resumen en caché en vez de mutación en vivo.

- Construye replay packets automáticamente. Un operador no debería reconstruir el state desde logs dispersos.
- Trata la compensación como un paso de primera clase con su propio éxito, falla y ruta de escalamiento.

## Failure Modes

- El loop reintenta un policy denial hasta que se agota el budget.
- Un side effect parcial se repite porque el retry no tiene una idempotency key.
- El re-planning oculta la falla original en lugar de preservar el trace.
- Un fallback devuelve datos de menor calidad sin marcar la respuesta como degradada.
- El controller escala sin suficiente state para que un humano continúe.
- Los loops sin progreso consumen budget porque cada iteración parece ligeramente diferente.
- La compensación falla y el sistema sigue actuando como si el rollback hubiera tenido éxito.
- La recovery policy vive solo en el prompt y no puede ser auditada.

## Review Checklist

Usa la [self-healing workflow review checklist](#) antes de avanzar un recovery loop más allá de la etapa de prototipo.

- Cada failure class tiene un owner y una acción de recuperación.
- Cada side effect reintentable tiene una idempotency key.
- Cada ruta de compensación tiene un mensaje de residual-risk.
- Cada condición de parada produce una explicación para el usuario u operador.
- Cada incidente en producción puede convertirse en un regression eval reproducible.

## Evaluation Strategy

Prueba la recuperación como comportamiento, no como manejo de excepciones.

| Eval Case              | Expected Result                                                                 |
|------------------------|---------------------------------------------------------------------------------|
| transient read timeout | reintenta con backoff, mismos inputs, trace event registrado                    |
| repeated timeout       | se detiene o escala cuando se agota el retry budget                             |
| policy-denied write    | no reintenta, registra la policy rule, retorna estado bloqueado                 |
| partial CRM write      | compensa o reanuda desde un state idempotente sin escritura duplicada           |
| stale plan             | re-planifica solo después de que cambian las evidencias                         |
| degraded fallback      | marca el output como derivado de fallback y de menor confianza                  |
| bad classifier         | falla el eval porque la acción de recuperación no coincide con la failure class |
| no-progress loop       | el breaker detiene tras el umbral y emite replay packet                         |

Mide la tasa de finalización, tasa de ejecuciones recuperadas, tasa de retries inseguros, tasa de side-effects duplicados, latencia media de recuperación, consumo de budget, calidad de escalamiento y tasa de éxito de replay.

## Production Checklist

- Define failure classes en código, no solo en instrucciones de prompt.
- Establece budgets de retry, costo, tiempo y no-progreso por workflow y por paso.
- Persiste el state antes de side effects visibles externamente.
- Almacena idempotency keys junto con action targets y resultados.
- Requiere policy denial para detener la recuperación autónoma.
- Emite eventos de trace estructurados para failure class, decisión de recuperación, budget y motivo de parada.
- Genera replay packets para escalaciones y recuperaciones fallidas.
- Agrega dashboards para tasa de retry, tasa de fallback, tasa de compensación, tasa de escalamiento y hits de prevención de duplicados.
- Convierte incidentes resueltos en regression evals antes de ampliar la automatización.

## Code Walkthrough

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control flow.

## Source Code

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el repositorio fuente.

```
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;
 }
 }
}

```

Extracto truncado por legibilidad. Descarga el bundle o abre el archivo fuente para la implementación completa.

[self-healing-workflow-agent-pattern/langgraph\\_python\\_example/self\\_healing\\_workflow.py](#)

[Abrir fuente completa](#)

```

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

```

*Fragmento truncado para facilitar la lectura. Descarga el bundle o abre el archivo fuente para ver la implementación completa.*

## Descarga

- [Descargar bundle de fuentes](#)
- [Abrir carpeta de fuentes](#)

El bundle de descarga contiene la carpeta `self-healing-workflow-agent-pattern/` actual de este repositorio.

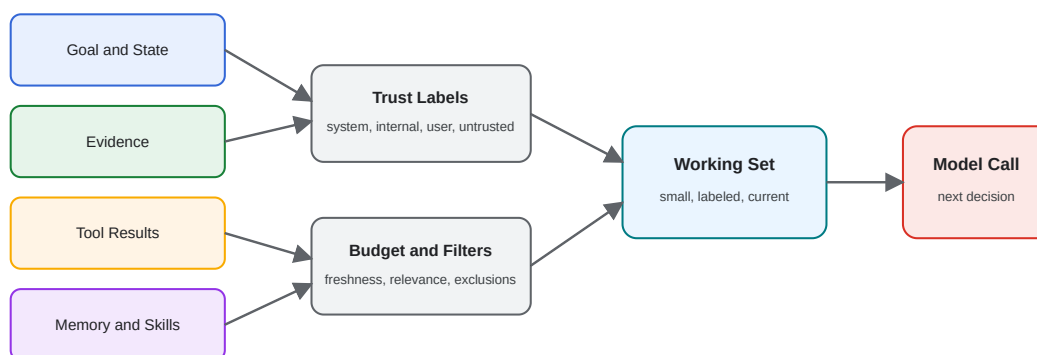
## Patrones relacionados

- [Planning and Execution](#)
- [ReAct](#)
- [Reflection](#)
- [Durable Workflows](#)
- [Circuit Breakers, Fallbacks, and Replay](#)

# Context Budgets And Working Sets

El context no es un sistema de almacenamiento. Es el working set del model para la siguiente decisión, y esa distinción es donde muchos agents fallan silenciosamente. Si tratas el context como una caja donde cualquier cosa potencialmente útil se puede tirar, terminas con prompts largos, state duplicado, memory obsoleta, output de tools irrelevante, contradicciones ocultas y luego falta de evidencia justo cuando el model la necesita. El context engineering es la disciplina de decidir qué debe ver el model en este momento, y nada más.

## Context Assembly Pipeline



## Qué Deberías Ser Capaz De Hacer

Después de este capítulo, deberías poder:

- definir el working set para una llamada de model;
- separar context requerido, relevante, disponible y excluido;
- escribir un context manifest que los revisores puedan inspeccionar;
- explicar por qué cada elemento de context fue incluido u omitido;
- probar la selección de context directamente en vez de adivinar a partir de la respuesta final.

## The Working Set

Toma la idea de los sistemas operativos: el working set es el pequeño conjunto de información necesario para el paso actual. Para un agent, eso normalmente significa el goal activo, el paso actual, las restricciones relevantes, el state compacto, la evidencia seleccionada, las tools disponibles para este paso, observaciones recientes, cualquier pregunta sin resolver, y el budget y reglas de detención.

Lo que no debería incluir automáticamente es todo lo demás: el historial completo del chat, cada documento recuperado, cada resultado de tool, todos los recuerdos sobre el usuario, todas las tools disponibles, cada archivo en el workspace y planes antiguos que ya no coinciden con el state actual. El model no necesita todo. Necesita lo correcto, y elegirlo es el trabajo.

## Per-Agent Working Sets

En un sistema multi-agent, cada agent debe tener su propio working set. Darle a cada agent el mismo prompt, los mismos archivos, la misma memory, las mismas tools y el mismo historial de conversación anula el propósito de descomponer el sistema desde el principio.

| 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.                  |

Esta es una de las razones más fuertes para tratar los agents como servicios. Cada agent recibe el context que su contrato requiere, no el context que todo el sistema lleva consigo.

## Context Budget

Cada llamada de model tiene un budget, incluso cuando la ventana de context es grande. Los tokens que usas se distribuyen entre instrucciones, state, evidencia, descripciones de tools, memory, historial de conversación, el espacio reservado para el output y el costo de cualquier compresión que ejecutes. Una ventana grande no elimina la necesidad de selección. Solo hace que una mala selección sea más costosa y difícil de notar.

Ajusta el budget a la decisión. Una llamada de routing no debería llevar todo el evidence bundle. Una llamada de selección de tool no debería llevar historial de usuario no relacionado. Una llamada de síntesis final debe incluir evidencia y restricciones, no cada pensamiento intermedio fallido que el agent tuvo en el camino.

## Context Budget Ledger

Lleva el control del budget para cada llamada antes de ensamblar el prompt.

```
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"

```

El ledger debe dejar algo claro: el context se asigna, no se acumula. Si una fuente no tiene línea de budget o regla de inclusión, no debe entrar a la llamada del model por accidente.

### Context Manifests

Para agents importantes, define un context manifest: una declaración explícita de lo que se permite ingresar a la llamada del model.

| Manifest Field       | Example                                                                                                |
|----------------------|--------------------------------------------------------------------------------------------------------|
| Required state       | <code>goal</code> , <code>current_step</code> , <code>tenant_id</code> , <code>policy_version</code> . |
| 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.                                                      |

Un manifest convierte la selección de context de un prompt craft a diseño de sistema. Una vez que existe, puede ser revisado, versionado, probado y auditado como cualquier otra parte del sistema.

En código, un working-set builder debe hacer visibles las reglas de inclusión:

```

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 };
}
```

El punto no es esta función de puntuación exacta. El punto es que la selección de context se vuelve un comportamiento inspeccionable, no un ensamblado de prompt oculto. La lista omitida importa tanto como la incluida, porque le dice a los operadores si el model omitió evidencia, excluyó memory obsoleta o se quedó sin budget.

### Sources Of Context

El context proviene de diferentes fuentes, y cada fuente tiene un nivel distinto de confianza.

| 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.             |

No mezcles todo en un solo bloque. Etiquétalos y mantén los límites claros: instructions no son evidence, afirmaciones de usuario no son hechos verificados y los resultados de tools no son policy.

### Niveles de Context

Los niveles son una forma sencilla de decidir qué entra realmente al model.

| Nivel     | Contenido                                                                            | Tratamiento Predeterminado                               |
|-----------|--------------------------------------------------------------------------------------|----------------------------------------------------------|
| Required  | Goal, policy, current state, stop rules.                                             | Siempre incluir en forma compacta.                       |
| Relevant  | Evidence, tool results, archivos, memories para el paso actual.                      | Incluir selectivamente con etiquetas de fuente.          |
| Available | Archivos del workspace, mensajes antiguos, documentos completos, herramientas extra. | Referenciar por ID y recuperar solo cuando se necesiten. |

| Nivel    | Contenido                                                                              | Tratamiento Predeterminado  |
|----------|----------------------------------------------------------------------------------------|-----------------------------|
| Excluded | Secrets, datos privados no relacionados, memory obsoleta, instrucciones no confiables. | Mantener fuera del context. |

Esta división previene la saturación del context y facilita las revisiones, ya que el equipo siempre puede preguntar por qué se incluyó un elemento determinado.

## Context Curado

Context curado significa que el sistema selecciona deliberadamente el conjunto más pequeño y útil de información antes de la llamada al model. Para un coding agent, esto normalmente ocurre como una secuencia:

1. identificar los archivos principales;
2. buscar símbolos, imports, tests y call sites;
3. clasificar archivos secundarios;
4. cargar directamente archivos pequeños relevantes;
5. cargar fragmentos o resúmenes para archivos grandes;
6. excluir archivos no relacionados incluso si están junto a los relevantes.

El mismo pattern aplica fuera del código. Un support agent no necesita todos los registros de clientes. Necesita la solicitud actual, los datos relevantes de la cuenta, la policy aplicable, los eventos recientes relacionados y las tools permitidas para este paso. La curaduría no es solo una optimización de costos. Es un control de confiabilidad, porque mientras más pequeño sea el working set, menos formas tiene el model de equivocarse.

## Divulgación Progresiva

Carga el context en etapas en lugar de todo de una vez. No hay razón para cargar cada documento, memory, archivo, descripción de tool y artifact de ejecuciones previas al inicio de una task. Comienza con lo mínimo necesario para elegir el siguiente camino y recupera más solo cuando el state lo justifique:

1. enrutar la task desde una solicitud de usuario compacta;
2. cargar la policy y el state necesarios para esa ruta;
3. exponer tools de solo lectura para investigación;
4. cargar evidencia detallada solo después de conocer los hechos faltantes;
5. exponer tools de escritura solo después de pasar verificaciones de elegibilidad y aprobación;
6. incluir evidencia final para síntesis y la explicación orientada al usuario.

Divulgar el context progresivamente reduce el costo de tokens, disminuye la exposición a prompt-injection y deja traces mucho más fáciles de inspeccionar después.

## Compresión



La compresión ayuda cuando el context es grande, pero no es gratis. Prefiere state estructurado sobre resúmenes de chat, IDs de fuente sobre documentos pegados, fragmentos cortos citados sobre texto completo, memory específica de la task sobre memory global y referencias de archivos sobre el contenido completo de archivos. Cuando resumas, mantén intactas las decisiones, restricciones y preguntas sin resolver.

Algunas cosas nunca deben comprimirse: approval state, errores de tools, restricciones de policy, procedencia de la fuente, contradicciones, correcciones de usuario, preguntas abiertas y razones de detención. Los resúmenes son inherentemente con pérdida, así que trátalos como artifacts derivados y no como la verdad de la ejecución.

## Compactación

La compactación no es lo mismo que un resumen ordinario. Un resumen produce un texto más corto; la compactación preserva el state operativo necesario para continuar una ejecución. Una compactación útil mantiene el goal activo, el current state, los pasos completados y pendientes, las decisiones tomadas, la evidence utilizada y sus source IDs, las llamadas a tools y sus resultados, la policy y el approval state, las preguntas abiertas, los errores conocidos y las razones de detención.

No debe ser tu estrategia principal de context. Si cada ejecución depende de la compactación de emergencia, el sistema está cargando demasiado context demasiado pronto. Usa la compactación como mecanismo de recuperación y continuación, no como permiso para seguir saturando el model.

## Límites de Compactación

La compactación del context debe ocurrir en límites estables, no en medio de un paso de razonamiento frágil. Buenos límites incluyen:

- después de que se reformulan los requerimientos;
- después de que el descubrimiento produce una lista de fuentes;
- después de que se acepta un plan;
- después de un commit o checkpoint;
- antes de entregar el trabajo a otro agent.

Cada compactación debe preservar decisiones, riesgos abiertos, rutas de archivos, evidencia de verificación y restricciones de usuario. Debe eliminar ruido de la transcripción, no borrar la responsabilidad.

## Minimización de Context

Elimina el context una vez que ha cumplido su propósito. Esto es especialmente importante para contenido no confiable. Después de que un mensaje de usuario, página web, correo, ticket o documento se convierte en un artifact intermedio validado, los pasos posteriores a menudo ya no necesitan el texto fuente original:

1. un mensaje de usuario entra como texto no confiable;
2. el sistema extrae la intención en una solicitud tipada;
3. la policy valida la solicitud;
4. las llamadas posteriores a tools usan la solicitud tipada, no el texto original.

Eso reduce la carga de tokens y, más importante aún, disminuye la probabilidad de que una instrucción antigua y no confiable siga influyendo en decisiones posteriores. Sin embargo, no minimices a ciegas. Conserva suficiente context para preservar la intención del usuario, la auditabilidad y las rutas de corrección. El objetivo no es olvidar, sino dejar de tratar cada token antiguo como una instrucción activa.

## Resultados de Tools

Los resultados de tools son context, pero no son instrucciones. Un resultado de búsqueda, página web, documento recuperado, fila de base de datos, observación de navegador o salida de comando puede contener texto hostil o engañoso, y el model debe ver todo eso como datos de una fuente, no como un nuevo mensaje del sistema.

Un buen context de resultado de tool llega envuelto en sus metadatos: fuente, timestamp, nivel de confianza, alcance, resultado bruto o extracto, campos parseados, cualquier estado de error y un correlation ID. Un mal context de resultado de tool solo pega el texto en el prompt y espera que el model sepa en qué confiar.

## Selección de Memory

La memory no debe cargarse por defecto. Antes de incorporar una memory al context, pregunta si es relevante para el goal actual, si es lo suficientemente reciente, si fue proporcionada por el usuario, inferida o importada, si está permitida para este usuario y task, si podría sesgar la respuesta incorrectamente y si puede corregirse o eliminarse. La memory es poderosa porque persiste, y eso es precisamente lo que la hace peligrosa: una memory equivocada puede seguir fallando en el sistema mucho después de que la ejecución que la creó haya terminado.

## Deriva del Working Set

La deriva del working set ocurre cuando el context ya no coincide con la task real. Se infiltra por caminos conocidos: un plan antiguo que sobrevive después de que el usuario cambia el goal, un resultado de recuperación obsoleto que nunca sale del context, un resumen comprimido que omite una restricción, un error de tool oculto por un éxito posterior, un resumen de subagent que omite su propia incertidumbre, memory reutilizada de otro dominio o un historial de chat que ahoga lentamente las instrucciones actuales.

La deriva es difícil de detectar solo a partir de la respuesta final, por eso se traza el paquete de context para cada llamada al model y no solo la salida.

## Confianza en la Fuente

Cada elemento de context debe llevar una etiqueta de confianza. Algunas útiles incluyen: system instruction, developer instruction, user request, verified internal data, retrieved internal document, external web content, tool result, long-term memory, generated summary, human approval y untrusted content. Las etiquetas ayudan al model y al runtime a interpretar correctamente el context, y ayudan a los operadores a reconstruir por qué ocurrió una mala decisión.

La regla más importante: el context de menor confianza nunca debe anular la gobernanza de mayor confianza. Un documento recuperado puede aportar evidence, pero no puede cambiar los permisos de tools. Un resultado de tool puede reportar state, pero no puede reescribir la approval policy. Una memory puede

informar la personalización, pero no puede anular la user request actual.

## Evaluación

Evalúa la selección de context directamente, no solo las respuestas finales. Casos útiles incluyen verificar que la evidence requerida esté presente, que la evidence irrelevante esté excluida, que la evidence obsoleta sea rechazada, que el prompt injection en texto recuperado sea ignorado, que las correcciones de usuario y errores de tools se conserven, que la memory se omita cuando no es relevante, que los resúmenes preserven restricciones y que la respuesta final cite las fuentes correctas. A nivel de sistema, también confirma que el working set de cada agent contenga solo lo que requiere su contrato, que la minimización elimine texto bruto no confiable después de la extracción, que la compactación preserve el approval state y las preguntas abiertas, y que el manifest realmente rechace los datos excluidos.

Métricas valiosas incluyen precisión y recall del context, frescura de la fuente, fidelidad de citación, costo de tokens, tasa de rechazo cuando falta evidence, tasa de preservación de errores, tasa de deriva, tasa de sobre-inclusión, tasa de context crítico faltante y tasa de pérdida por compactación. Cuando una respuesta sale mal, inspecciona el context que la produjo antes de culpar al model.

## Lista de verificación de diseño

Antes de lanzar un agent con mucho context, revisa estas preguntas: ¿Qué siempre se incluye? ¿Qué se recupera solo cuando es necesario? ¿Qué nunca se permite en el context? ¿Cómo se etiquetan las fuentes y cómo se mantienen separados los resultados de tools de las instrucciones? ¿Cómo se detecta memory obsoleta? ¿Cómo se generan y validan los resúmenes? ¿Qué context bundle se usó para cada llamada al model y puede un operador reproducir la ejecución con el mismo context? ¿Qué evals prueban la selección de context en lugar de solo las respuestas finales? ¿Cada agent tiene su propio working set? ¿Qué entradas sin procesar y no confiables se eliminan después de la extracción, qué conserva la compactación y qué trust labels aparecen en tus traces?

## Regla de diseño

No maximices el context. Cura el working set.

## Capítulos relacionados

- [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 controla lo que el model ve: instrucciones, state, resultados de retrieval, documentación de tool, memory, ejemplos y mensajes previos.

## Fuente y descargas

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

## Intención

Context engineering decide lo que el model ve antes de responder o actuar. Ensambla instrucciones, entrada del usuario, working memory, evidencia recuperada, resultados de tool, policies, ejemplos, presupuestos y exclusiones en un conjunto de trabajo controlado.

RAG no es “poner resultados de búsqueda en el prompt”. Es un pipeline de evidencia. Context engineering es la disciplina más amplia alrededor de ese pipeline. El sistema debe decidir qué fuentes son elegibles, qué cuenta como suficientemente fresco, qué puede ver el caller, cómo se cita el contenido recuperado, qué se omite, qué se fija y qué debe hacer el agent cuando falta o hay conflicto en la evidencia.

La regla es simple: el context se ensambla, no se vuelca. El model debe ver un paquete deliberado, no un montón de historial de chat, texto recuperado, salida de tool y fragmentos de memory.

Este pattern es dueño del límite de ensamblaje, no de cada sistema fuente dentro de él. Working memory es dueño del run state. Durable memory es dueño de lo que sobrevive entre runs. Retrieval es dueño del descubrimiento de evidencia. Context engineering decide cuáles de esos registros entran en la siguiente llamada al model.

## Usar cuando

- La respuesta depende de una base de conocimiento grande, cambiante o privada.
- Las fuentes relevantes pueden dividirse en chunks, embeberse, filtrarse, citarse e inspeccionarse.
- La capa de retrieval puede aplicar restricciones de tenant, rol, fuente y frescura.
- El agent puede rechazar o escalar cuando falta evidencia.
- El sistema puede evaluar la calidad de retrieval por separado de la calidad de la respuesta final.
- El agent necesita working memory, resultados de tool, policy y evidencia recuperada en un solo paquete coherente.

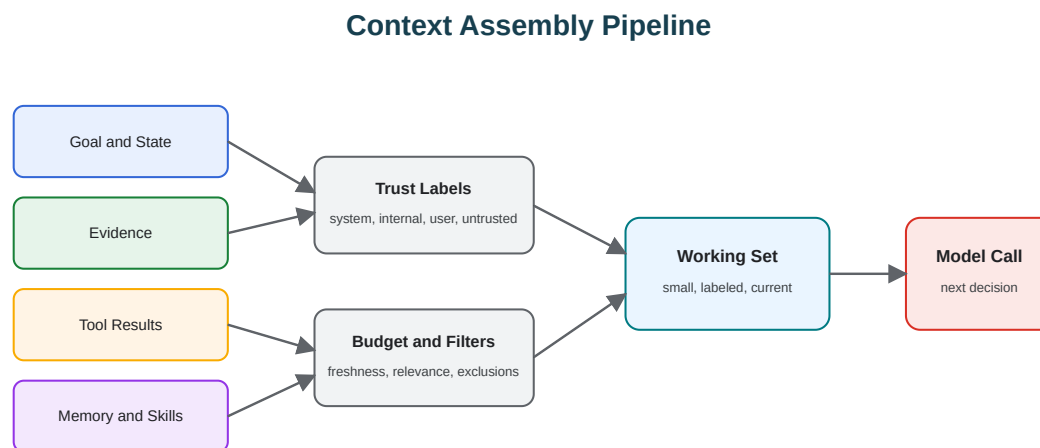
## Evitar cuando

- El conocimiento requerido ya está en la entrada del task o en el deterministic system state.
- El corpus es demasiado ruidoso, obsoleto o no confiable para recuperarse de forma segura.
- El sistema no puede distinguir metadata confiable de contenido de documento no confiable.

- La respuesta necesita el exact database state que debe venir de un typed tool, no de una búsqueda semántica.
- Las citas, IDs de fuente o traces de retrieval no pueden almacenarse para revisión.
- El sistema dependería de volcar transcripciones crudas en vez de un context builder controlado.

## Arquitectura

Usa este diagrama para leer Context Engineering como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el caller o un pequeño servicio de aplicación es dueño del task state hasta que se introduce un runtime pattern.



## Forma del sistema

- **Límite del pattern:** el context builder decide qué puede entrar al model working set y cómo se etiqueta cada parte.
- **Dueño del state:** el runtime es dueño del working state, policies, budgets, retrieval traces, resultados de tool, registros de memory y registros de ensamblaje de context.
- **Rol del model:** el model usa el paquete ensamblado para responder, actuar, pedir evidencia faltante o explicar por qué no puede continuar.
- **Límite de policy:** elegibilidad de fuentes, acceso de tenant, redacción, frescura, manejo de resultados de tool, escrituras de memory y jerarquía de instrucciones se ejecutan antes de la generación.
- **Promesa operativa:** el model trabaja a partir de un paquete pequeño, etiquetado y verificado por policy en vez de memory vaga o context sin filtrar.

## Protocolo central

1. Clasifica la necesidad de context: policy, instrucciones del task, working state, resultado de tool, user memory, historial de eventos, registro privado, fuente pública o ejemplo.
2. Carga el goal actual, working memory, estado de presupuesto, estado de aprobación y condiciones de parada.
3. Aplica filtros de caller, tenant, rol, fuente, frescura y manejo de datos antes de recuperar o incluir.
4. Recupera candidate chunks con metadata, no solo texto.

5. Ordena y recorta candidatos por relevancia, frescura, diversidad, confianza y presupuesto de tokens.
6. Construye un set de evidencia con IDs de fuente, citas, confianza y brechas conocidas.
7. Ensambla el context con instrucciones, policy, state, evidencia, resultados de tool, memory, ejemplos y exclusiones separados.
8. Genera una respuesta, acción, rechazo o escalamiento usando solo el context elegible.
9. Registra la consulta, filtros, IDs de fuente, secciones del paquete, material omitido, citas y razón de parada.

## Notas de implementación

Trata el material recuperado como evidencia, no como autoridad. Una página web, correo, ticket, PDF o documento puede contener hechos útiles e instrucciones maliciosas al mismo tiempo.

### Context Packet

Un context packet es el runtime artifact que explica lo que el model pudo ver.

```
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;
};

```

Este paquete debe ser trazable. Si una ejecución falla, deberías poder inspeccionar qué se incluyó, qué se omitió y por qué.

## Reglas de orden y confianza

El orden es parte del control plane. Un orden práctico es:

1. instrucciones del sistema;
2. restricciones de policy y seguridad;
3. solicitud actual del usuario o workflow;
4. goal y working memory;
5. resultados de tool aprobados;
6. evidencia recuperada;
7. memories recuperadas;
8. ejemplos;
9. exclusiones explícitas y brechas conocidas.

El punto no es solo el orden de los tokens. El punto es la separación. El texto recuperado, salida de tool, memory y documentos del usuario nunca deben poder reescribir las instrucciones del sistema, policy, permisos de tool, reglas de aprobación o reglas de escritura de memory.

## 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

El evidence contract debe viajar con la respuesta:

```

type RagAnswer = {
 status: 'answered' | 'missing_evidence' | 'conflicting_evidence' | 'refused';
 answer?: string;
 citations: string[];
 evidenceRefs: string[];
 missingEvidence?: string[];
};

```

## Eligibility Check

Un pequeño eligibility check previene muchas fallas en producción:

```

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';
}

```

No dejes que el model decida si una fuente está permitida. El model puede resumir la calidad de la evidencia. El software debe hacer cumplir la elegibilidad.

## Context Budgeting



Context budgeting no es solo truncar. Es decidir qué debe ver el model, qué se puede resumir, qué se puede recuperar de nuevo, qué se puede omitir y qué debe fijarse.

Fija elementos pequeños y de alta autoridad: instrucciones del sistema, restricciones de policy, goal activo, condición de parada, estado de aprobación y la solicitud del usuario. Resume historial voluminoso pero de bajo riesgo. Recupera evidencia de fuente en vez de cargar extractos viejos copiados. Elimina chunks duplicados. Omite material obsoleto, no autorizado o de baja confianza. Reserva tokens de salida antes de llenar el input context.

Cuando el presupuesto es limitado, el runtime debe degradar explícitamente:

- responder con menos fuentes citadas;
- hacer una pregunta aclaratoria;
- recuperar de nuevo con filtros más estrechos;
- resumir el state intermedio;
- rechazar cuando la evidencia requerida no cabe de forma segura;
- escalar cuando la policy requiere evidencia completa.

## Modos de falla

- Se recupera evidencia obsoleta pero plausible y se trata como actual.
- El texto recuperado contiene instrucciones que anulan el goal del sistema.
- La salida de tool se trata como una nueva instrucción en lugar de como datos.
- Memory anula la solicitud actual del usuario.
- El working state está obsoleto pero aún se incluye como actual.
- Chunks del tenant, rol o cliente equivocado entran al context.
- El retriever devuelve fuentes semánticamente similares pero operativamente incorrectas.
- Las citas apuntan a documentos amplios en lugar del chunk exacto de soporte.
- El agent responde cuando falta evidencia o hay conflicto.
- Memory escribe un resumen no verificado como si fuera un hecho durable.
- Se registran los puntajes de retrieval, pero no los filtros, IDs de fuente ni frescura.
- Se evalúa la respuesta final, pero no la calidad de retrieval.
- El context packet no puede reconstruirse después de un incidente.
- El policy o approval state importante se omite bajo presión de tokens.

## Estrategia de evaluación

Los context evals deben probar retrieval, ensamblaje y comportamiento de respuesta por separado.

- Prueba preguntas de respuesta conocida donde la fuente correcta está presente.
- Prueba casos de falta de evidencia donde el agent debe rechazar o pedir ayuda.
- Prueba casos de fuente obsoleta donde una fuente antigua entra en conflicto con una nueva.
- Prueba casos de fuentes en conflicto donde la respuesta debe explicar la incertidumbre.
- Prueba prompt injection dentro de documentos recuperados.
- Prueba prompt injection dentro de resultados de tools y registros de memory.
- Prueba límites de tenant y permisos.
- Prueba presión de presupuesto de context y verifica que policy, goal y citas no se omitan.

- Prueba comportamiento de fuentes omitidas y verifica que el trace explique por qué se excluyó material.
- Prueba cobertura de citas: cada afirmación factual debe mapearse a evidencia de fuente.
- Prueba precisión y recall de retrieval antes de probar la calidad final del texto.

Un eval fixture compacto puede hacer explícito el requerimiento de evidencia:

```
{
 "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"]
 }
}
```

Mide recall de retrieval, precisión de retrieval, frescura de fuente, completitud del packet, eficiencia de tokens de context, fidelidad de citas, tasa de rechazo por falta de evidencia, resistencia a prompt-injection, violaciones de límites de tenant y calidad de respuesta basada en la evidencia citada.

## Lista de verificación para producción

- Define fuentes elegibles por tenant, rol, tipo de fuente, frescura y clase de datos.
- Mantén metadatos de fuente con cada chunk recuperado.
- Construye un context packet trazable para cada ejecución.
- Separa instrucciones de hechos recuperados en el context ensamblado.
- Mantén resultados de tools, memory, evidencia recuperada, ejemplos y contenido de usuario etiquetados por separado.
- Redacta o excluye fuentes antes de la generación, no después.
- Fija policy, goal activo, condición de parada y approval state antes de llenar context opcional.
- Requiere citas para afirmaciones factuales.
- Rechaza o escala cuando la evidencia falta, está obsoleta o en conflicto.
- Traza consulta, filtros, IDs de fuente, puntajes, secciones del packet, referencias omitidas y citas finales.
- Evalúa la calidad de retrieval por separado de la calidad de respuesta.
- Revisa memory writes antes de almacenar resúmenes recuperados como hechos durables.
- Versiona chunking, embedding model, filtros de retrieval, rerankers, prompts y reglas de citas.
- Agrega regression evals para context poisoning, context obsoleto, policy omitido y presión de tokens.

La regla arquitectónica es simple: cada elemento en el context necesita una razón, una fuente, un nivel de confianza y un presupuesto. Continúa con [Context Budgets and Working Sets](#) para límites de packet y [Agentic RAG Systems](#) para composición a nivel de sistema.

## Recorrido de código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo circundante explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o flujo de control.

## Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el repositorio fuente.

`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)
```

## Descarga

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

El bundle de descarga contiene la carpeta `context-engineering-pattern/` actual de este repositorio.

## Patrones relacionados

- [Context Budgets and Working Sets](#)
- [Working Memory](#)
- [Memory-Augmented Agent](#)
- [Knowledge-Bound Agents](#)
- [Agentic RAG Systems](#)
- [Pattern Evaluation Checklist](#)

# Memory-Augmented Agent

La memory le da continuidad a un agent, pero también crea un límite de confianza durable.

## Fuente y descargas

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

## Intención

Los memory-augmented agents almacenan y recuperan información a lo largo de turnos, sesiones, tasks o usuarios. La memory le da continuidad a un agent, pero también crea un límite de confianza durable. Una mala respuesta puede corregirse en el siguiente turno. Una mala memory puede seguir influyendo en ejecuciones futuras.

La forma útil de pensar en la memory es simple: la memory no es verdad. Es un registro con fuente, alcance, timestamp, nivel de confianza, clase de privacidad, regla de retención y ruta de corrección. El model puede proponer un memory write. El runtime decide si ese write está permitido, cómo se almacena, cuánto tiempo vive y cuándo puede ser recordado.

Usa este pattern para una memory policy durable. Para el state de una sola ejecución, usa [Working Memory](#). Para decidir si una memory entra en el prompt, usa [Context Engineering](#) y [Context Budgets and Working Sets](#).

## Usa cuando

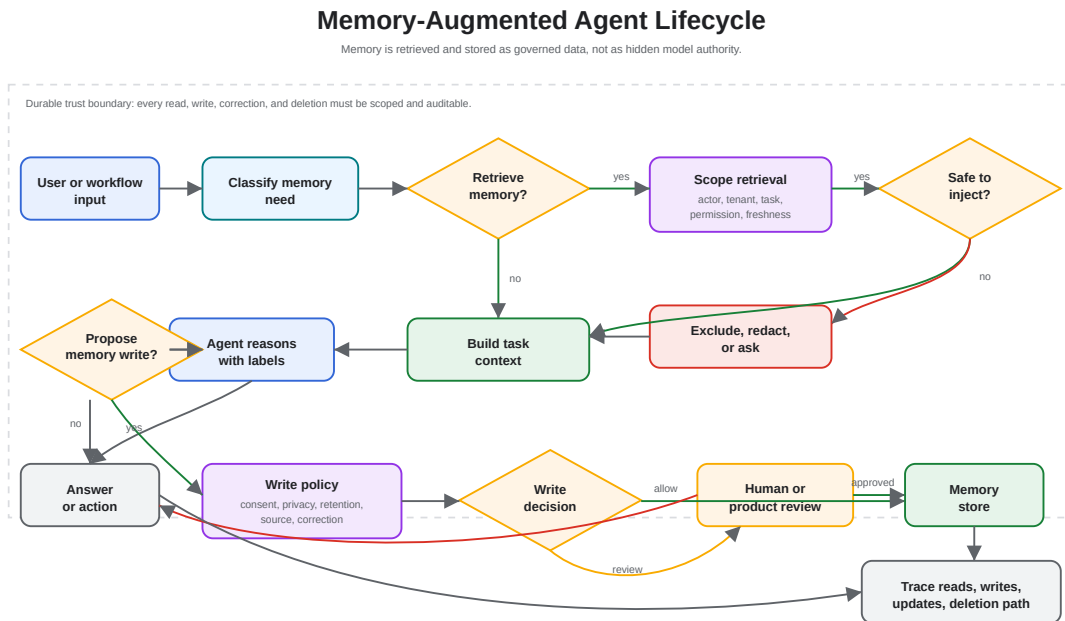
- El agent necesita continuidad más allá de una sola interacción.
- Los hechos almacenados pueden tener alcance, actualizarse, corregirse y eliminarse.
- Los resultados recuperados pueden citarse, inspeccionarse y excluirse cuando están obsoletos o no son seguros.
- El producto tiene una razón para recordar preferencias, state de tasks, decisiones, eventos o hechos proporcionados por el usuario.
- Puedes definir consentimiento, retención, privacidad y límites de tenant.

## Evita cuando

- El sistema almacenaría datos sensibles sin consentimiento o reglas de retención.
- Las memorias recuperadas no pueden distinguirse de las instrucciones actuales.
- El memory store se usa como un volcado sin curar de transcripciones.
- El agent trataría resúmenes antiguos como hechos autoritativos.
- No puedes explicar, editar o eliminar lo que el agent recuerda.

## Arquitectura

Usa este diagrama para leer Memory-Augmented Agent como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: la capa de memory o retrieval posee el conocimiento de larga duración, mientras que el agent posee el working state local de la task.



Léelo como un ciclo de vida gobernado: recupera solo la memory con alcance, inyéctala como datos etiquetados y almacena solo la memory aprobada con reglas de retención y corrección.

## Forma del sistema

- **Límite del pattern:** el límite de memory posee el alcance de retrieval, write policy, schema de registro, retención, corrección, eliminación y auditoría.
- **Propietario del state:** el servicio de memory o el application store posee la memory durable; el agent solo posee lecturas y escrituras propuestas.
- **Rol del model:** el model puede resumir, clasificar y proponer actualizaciones de memory, pero no decide en silencio qué recordará el sistema.
- **Límite de policy:** los memory writes pasan por controles de consentimiento, fuente, privacidad, tenant, retención y seguridad antes de almacenarse.
- **Promesa operativa:** la memory mejora la continuidad sin convertirse en un cúmulo no revisado de afirmaciones durables.

## Protocolo central

1. Clasifica la necesidad de información: working state, evento episódico, hecho semántico, preferencia de usuario, fuente de policy o tool-result cache.
2. Recupera solo registros de memory con alcance para el actor, tenant, task, permisos y ventana de vigencia actuales.
3. Inyecta la memory como datos con etiquetas de fuente, timestamps, nivel de confianza y nivel de confianza.
4. Mantén la memory recuperada separada de las instrucciones del sistema y la policy.

- 5. Pide al model que proponga memory writes solo cuando la task produjo un hecho durable, preferencia, evento, corrección o decisión.
- 6. Ejecuta la memory write policy antes de almacenar.
- 7. Almacena la memory aprobada con procedencia, retención, clase de privacidad y ruta de eliminación.
- 8. Registra lecturas, escrituras, actualizaciones, rechazos y eliminaciones de memory en el trace.

Notas de implementación

No almacenes por defecto el historial de chat sin procesar como memory. El historial de chat es evidencia. La memory es un registro operativo curado. La diferencia importa porque la memory se reutiliza en ejecuciones futuras.

Tipos de memory

| Memory Type       | Qué almacena                                                        | Riesgo típico                                                    |
|-------------------|---------------------------------------------------------------------|------------------------------------------------------------------|
| Working memory    | State actual de la task, preguntas abiertas, restricciones activas. | state obsoleto o inconsistente dentro de una ejecución.          |
| Episodic memory   | Eventos ocurridos, con tiempo y participantes.                      | retención excesiva, fuga de privacidad, atribución incorrecta.   |
| Semantic memory   | Hechos durables sobre un dominio, usuario, proyecto o sistema.      | tratar afirmaciones no verificadas como verdad.                  |
| Preference memory | Elecciones y hábitos del usuario.                                   | sobrepersonalización o almacenamiento de preferencias sensibles. |
| Policy memory     | Reglas, fuentes y restricciones aprobadas.                          | policy obsoleta o ediciones no autorizadas.                      |
| Tool-result cache | Salidas previas de tools reutilizadas por velocidad o costo.        | datos obsoletos y fuga entre tenants.                            |

Contrato de escritura de memory

Un memory write debe ser una solicitud tipada. El registro debe indicar qué se almacena, por qué está permitido, de dónde proviene, a quién pertenece y cómo puede corregirse.

```
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" };
}

```

## Registros de memory

El registro almacenado no debe ser solo texto.

```
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;
};
```

Este schema es intencionalmente aburrido. Una memory aburrida es más fácil de inspeccionar, corregir, eliminar, evaluar y auditar.

## Reglas de recuperación

La recuperación de memory debe tener alcance antes de la clasificación por relevancia. Primero filtra por tenant, actor, permisos, tipo de memory, retención y vigencia. Luego clasifica por relevancia. Si el sistema clasifica primero y filtra después, la memory privada u obsoleta puede filtrarse al context.

La memory recuperada debe entrar al context con etiquetas como `source`, `created_at`, `confidence`, `privacy_class` y `trust_level`. No permitas que la memory recuperada sobrescriba instrucciones del sistema, tool policy, reglas de aprobación o controles de seguridad.

## Modos de falla

- Se almacenan transcripciones sin procesar como memory durable.
- El model escribe memory en silencio sin consentimiento, policy o trace.
- Páginas web, correos, tickets o documentos no confiables contaminan la memory futura.
- La memory obsoleta sobrescribe evidencia más reciente.
- Las preferencias de usuario se tratan como hechos.
- Hechos de un tenant, usuario o proyecto aparecen en otro context.
- Se almacenan datos sensibles sin reglas de retención, eliminación o corrección.
- Los resúmenes pierden la evidencia necesaria para verificar o reparar la memory.
- El sistema no tiene forma de mostrar a los usuarios lo que se recuerda.
- La memory crece hasta que la recuperación se vuelve ruidosa y costosa.

## Estrategia de evaluación

Los memory evals deben probar tanto la capacidad de recordar como la de restringir. Un sistema de memory que recuerda todo no es bueno. Es riesgoso.

- Prueba escrituras de memory permitidas por preferencias explícitas del usuario.
- Prueba escrituras denegadas para datos sensibles sin consentimiento.
- Prueba escrituras que requieren revisión provenientes de contenido no confiable.
- Prueba la corrección de un registro de memory existente.
- Prueba la eliminación y verifica que el registro no pueda recuperarse después.
- Prueba memory obsoleta contra evidencia más reciente.
- Prueba el aislamiento de tenant y actor.
- Prueba la recuperación de memory con registros conflictivos.
- Prueba si la memory recuperada se cita como memory y no se trata como instrucción.
- Prueba que las escrituras de memory aparezcan en traces junto con decisiones de policy.

Mide precisión de escritura, recall de escritura, tasa de escrituras inseguras, tasa de recall obsoleto, éxito de corrección, éxito de eliminación, filtración entre tenants, relevancia de recuperación, cobertura de citación y precisión en decisiones de policy.

## Lista de verificación para producción

- Define los tipos de memory que el sistema puede almacenar.
- Mantén los registros de memory tipados, con alcance, respaldados por fuente y con marca de tiempo.
- Requiere consentimiento para memory sensible o personal.
- Filtra por tenant, actor, permiso, frescura y retención antes de clasificar por relevancia.
- Separa la memory recuperada de instrucciones y policy.
- Agrega policy de escritura para fuentes no confiables, datos sensibles, memory de policy y resúmenes de resultados de tool.
- Proporciona rutas para corrección y eliminación.
- Traza lecturas, escrituras, actualizaciones, denegaciones, aprobaciones y eliminaciones de memory.
- Convierte escrituras inseguras de memory y recalls obsoletos en regression evals.
- Da a los operadores una forma de deshabilitar escrituras de memory sin deshabilitar la recuperación.

## Ejecuta el ejemplo

```
npm run memory-augmented-agent
```

## Recorrido del código

Lee el fragmento como la expresión ejecutable más pequeña del pattern. El capítulo alrededor explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o flujo de control.

## Código fuente

Estos fragmentos muestran la forma de la implementación. El código completo está disponible en el paquete de descarga y en el repositorio fuente.

[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();

```

[memory-augmented-agent-pattern/langgraph\\_python\\_example/memory\\_agent.py](#)

[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()
```

## Descarga

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

El paquete de descarga contiene la carpeta actual `memory-augmented-agent-pattern/` de este repositorio.

## Patrones relacionados

- [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

La long-term episodic memory almacena eventos: qué sucedió, cuándo, quién estuvo involucrado y por qué fue importante.

## Fuente y descargas

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

## Intención

La long-term episodic memory almacena eventos: qué sucedió, cuándo sucedió, quién o qué estuvo involucrado, qué evidencia lo respalda y por qué podría importar más adelante.

La episodic memory no es lo mismo que la semantic memory. La semantic memory almacena hechos o conocimientos. La episodic memory almacena eventos recordados. “El cliente prefiere email” es una preferencia o afirmación semántica. “El 2026-06-17 el cliente pidió recibir avisos de renovación por email” es un episodio. Mantener esa distinción clara facilita auditar, corregir, expirar y explicar la memory.

El objetivo no es recordar todo. El objetivo es preservar eventos útiles con suficiente procedencia para que futuras ejecuciones puedan razonar a partir de ellos sin tratar información antigua, parcial o privada como verdad universal.

## Usar cuando

- El asistente necesita continuidad entre sesiones.
- Eventos pasados afectan decisiones actuales, personalización, soporte, operaciones o historial de proyectos.
- Los eventos pueden recuperarse por relevancia, recencia, actor, tenant, proyecto y tipo de evento.
- El sistema puede hacer cumplir retención, privacidad, corrección, eliminación y aislamiento de tenant.
- Se necesita una línea de tiempo inspeccionable en vez de un perfil de usuario vago.

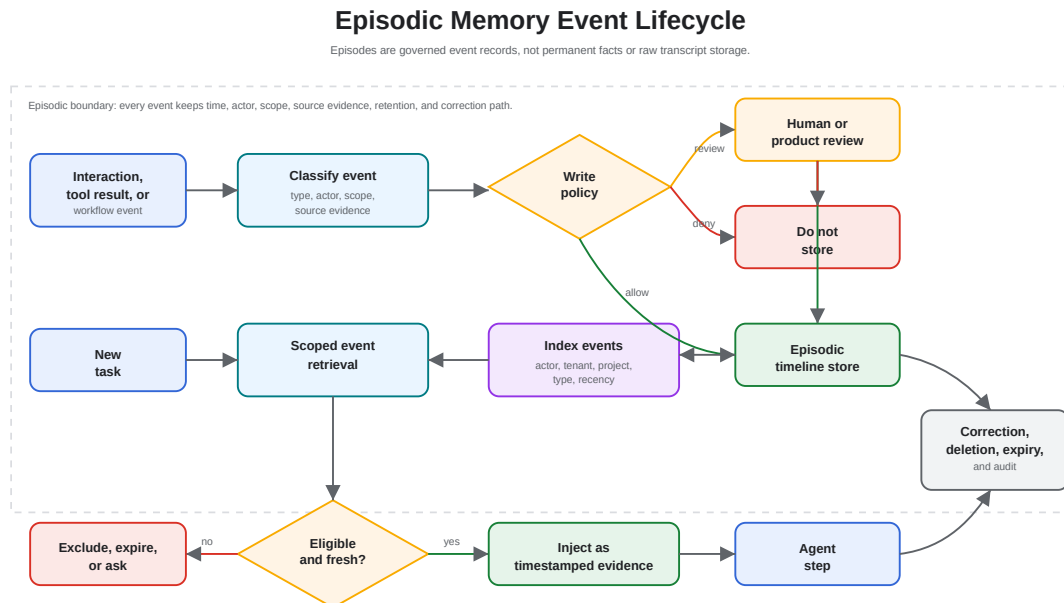
## Evitar cuando

- La task solo necesita hechos semánticos, no historial de eventos.
- El sistema no puede explicar o eliminar eventos recordados.
- La memory almacenaría eventos sensibles sin consentimiento o límites de retención.
- El agent convertiría cada interacción en un episodio durable.
- El producto no puede tolerar atribución incorrecta, recall obsoleto o filtraciones entre usuarios.

## Arquitectura



Usa este diagrama para leer Long-Term Episodic Memory como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: la memory o la capa de retrieval posee el conocimiento de larga duración, mientras que el agent posee el state de trabajo local de la task.



Léelo como un ciclo de vida de eventos: almacena solo episodios aprobados por policy, recupéralos por alcance y frescura, y mantén visibles los caminos de corrección, eliminación, expiración y auditoría.

## Forma del sistema

- **Límite del pattern:** el límite de episodic memory posee la clasificación de eventos, policy de escritura, almacenamiento de la línea de tiempo, alcance de retrieval, retención, corrección, eliminación y auditoría.
- **Propietario del state:** el servicio de memory posee los registros de eventos durables; el agent solo posee escrituras de eventos propuestas y lecturas de eventos con alcance.
- **Rol del model:** el model puede resumir y clasificar eventos candidatos, pero el runtime decide si el evento vale la pena almacenarse.
- **Límite de policy:** las escrituras de eventos se revisan para consentimiento, clase de privacidad, confianza en la fuente, alcance de tenant, retención y requisitos de revisión.
- **Promesa operativa:** futuras ejecuciones pueden usar eventos pasados sin confundirlos con instrucciones actuales o hechos permanentes.

## Protocolo central

1. Observar una interacción, resultado de tool, transición de workflow, decisión humana, corrección o evento externo.
2. Decidir si el evento vale la pena recordarse.
3. Clasificar el tipo de evento, actor, tenant, proyecto, recursos, evidencia de origen, clase de privacidad, confianza y regla de retención.
4. Ejecutar la policy de escritura de episodic memory.

5. Almacenar eventos aprobados en una línea de tiempo y, cuando sea útil, en un índice para retrieval semántico.
6. Recuperar eventos solo después de filtrar por tenant, actor, permisos, tipo de evento, recencia y retención.
7. Inyectar eventos recuperados como evidencia con timestamps y referencias de origen.
8. Permitir corrección, eliminación, expiración y revisión de auditoría.

## Notas de implementación

La episodic memory debe tener forma de evento. Si el registro no incluye tiempo, actor, fuente y alcance, probablemente no es un episodio. Es una nota.

### Tipos de eventos

| Event Type             | Ejemplo                                                        | Notas                                                |
|------------------------|----------------------------------------------------------------|------------------------------------------------------|
| User preference stated | El usuario pidió reportes en PDF en vez de diapositivas.       | Puede requerir consentimiento según la sensibilidad. |
| Correction received    | El usuario corrigió un nombre de proyecto o rol de contacto.   | Debe actualizar o reemplazar eventos anteriores.     |
| Task milestone         | El agent completó la revisión del plan de despliegue.          | Útil para continuidad de proyectos.                  |
| Human approval         | Finance aprobó la solicitud de reembolso <code>R-1024</code> . | Mantener ID de aprobación y versión de policy.       |
| Tool-observed event    | CRM devolvió que el estado de la cuenta cambió a activo.       | Almacenar fuente y frescura.                         |
| Incident or failure    | El agent falló porque un tool expiró.                          | Útil para confiabilidad y evals.                     |

### Registro de evento

Un registro episódico útil mantiene el contenido del evento separado de la evidencia y los metadatos de retrieval.

```
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;
};

```

## Policy de escritura

El model no debe escribir un evento en silencio solo porque parece útil. El runtime debe decidir.

```

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" };
}
```

## Reglas de retrieval

El retrieval episódico debe comenzar por alcance, no por similitud. Filtra por tenant, actor, proyecto, permisos, tipo de evento, retención y recencia antes de hacer búsqueda por embedding o ranking. Luego clasifica los eventos por relevancia, recencia, confianza en la fuente y confianza.

Los eventos recuperados deben presentarse como eventos, no como hechos:

```
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
```

Ese encuadre importa. El evento puede explicar el context, pero no debe anular la intención actual del usuario, la policy vigente, datos frescos de tools o reglas de aprobación.

## Modos de falla

- El sistema almacena cada interacción como un evento y el retrieval se vuelve ruidoso.
- Un evento antiguo se trata como un hecho actual.
- Un evento de preferencia se convierte en un rasgo permanente del usuario.
- Se escribe un evento falso a partir de un resumen alucinado.
- Un evento se atribuye al usuario, tenant, proyecto o recurso equivocado.
- Eventos sensibles se retienen indefinidamente.
- Las correcciones crean nuevos eventos conflictivos en vez de reemplazar los anteriores.
- Eventos de documentos no confiables contaminan el comportamiento futuro.
- El orden de la línea de tiempo es incorrecto porque `recordedAt` se confunde con `occurredAt`.
- El sistema no puede mostrar, corregir, eliminar o expirar registros de eventos.

## Estrategia de evaluación

Las evals de episodic memory deben probar la calidad de escritura de eventos, calidad de retrieval y moderación.

- Prueba que los eventos significativos se almacenen con actor, tenant, tiempo, fuente, confianza y retención.
- Prueba que la charla rutinaria no se almacene.
- Prueba supresión de eventos obsoletos cuando existe evidencia más reciente.
- Prueba corrección y reemplazo de un evento anterior.
- Prueba eliminación y expiración.
- Prueba aislamiento entre tenants y usuarios.

- Prueba prevención de atribución incorrecta.
- Prueba retrieval por recencia, relevancia y tipo de evento.
- Prueba revisión de fuentes no confiables antes de almacenar.
- Prueba que los eventos recuperados no anulen instrucciones o policy actuales.

Mide precisión de escritura de eventos, recall de escritura de eventos, exactitud de atribución, recall de eventos obsoletos, éxito de corrección, éxito de eliminación, fallas de aislamiento de tenant, relevancia de retrieval y tasa de escritura insegura.

## Lista de verificación para producción

- Define qué tipos de eventos el sistema puede recordar.
- Almacena `occurredAt` y `recordedAt` por separado.
- Conserva los metadatos de actor, tenant, project, resource, source, confidence, privacy y retention en cada evento.
- Filtra por scope antes del ranking semántico.
- Trata los eventos recuperados como evidencia, no como instrucciones.
- Requiere revisión para eventos sensibles, no confiables o significativos para la policy.
- Soporta corrección, reemplazo, eliminación y expiración.
- Traza lecturas, escrituras, rechazos, revisiones, correcciones y eliminaciones de eventos.
- Convierte falsas memorias y recalls obsoletos en regression evals.
- Da a los usuarios u operadores una línea de tiempo inspeccionable cuando el producto requiere confianza.

## Recorrido por el código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo circundante explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

## Código fuente

Este pattern actualmente no tiene un extracto de código dedicado. Usa los enlaces de fuente y descarga a continuación para acceder a la carpeta completa del pattern.

## Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `long-term-episodic-memory-agent-pattern/` actual de este repositorio.

## Patterns relacionados

- [Memory-Augmented Agent](#)
- [Goals and State](#)

- Context Engineering
- Human Approval Gates
- Policy Enforcement

# Semantic Recall and RAG

Semantic recall recupera material relevante por significado en lugar de palabras clave exactas. RAG inyecta el material recuperado en el context antes de la generación.

## Fuente y descargas

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

## Propósito

El context engineering decide qué ve el model antes de responder o actuar. Ensambla instrucciones, entrada del usuario, working memory, evidencia recuperada, resultados de tools, políticas, ejemplos, presupuestos y exclusiones en un conjunto de trabajo controlado.

RAG no es “poner resultados de búsqueda en el prompt”. Es un pipeline de evidencia. El context engineering es la disciplina más amplia alrededor de ese pipeline. El sistema debe decidir qué fuentes son elegibles, qué cuenta como suficientemente reciente, qué puede ver el solicitante, cómo se cita el contenido recuperado, qué se omite, qué se fija y qué debe hacer el agent cuando falta o hay conflicto en la evidencia.

La regla es simple: el context se ensambla, no se vuelca. El model debe ver un paquete deliberado, no un montón de historial de chat, texto recuperado, salida de tools y fragmentos de memory.

Este pattern controla el límite de ensamblaje, no cada sistema fuente dentro de él. La working memory controla el run state. La durable memory controla lo que sobrevive entre ejecuciones. Retrieval controla el descubrimiento de evidencia. El context engineering decide cuáles de esos registros entran en la siguiente llamada al model.

## Usar cuando

- La respuesta depende de una base de conocimiento grande, cambiante o privada.
- Las fuentes relevantes pueden dividirse en chunks, embeberse, filtrarse, citarse e inspeccionarse.
- La capa de retrieval puede aplicar restricciones de tenant, rol, fuente y frescura.
- El agent puede rechazar o escalar cuando falta evidencia.
- El sistema puede evaluar la calidad de retrieval por separado de la calidad de la respuesta final.
- El agent necesita working memory, resultados de tools, policy y evidencia recuperada en un solo paquete coherente.

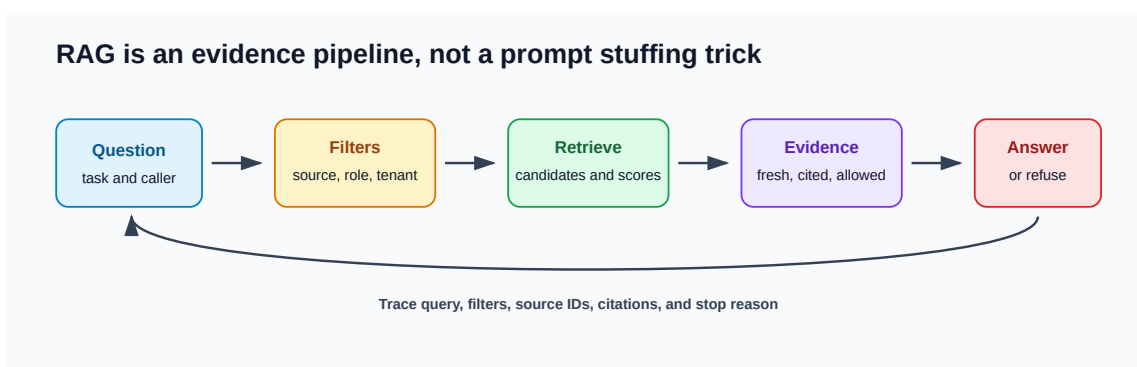
## Evitar cuando

- El conocimiento requerido ya está en la entrada del task o en el state determinista del sistema.
- El corpus es demasiado ruidoso, obsoleto o no confiable para recuperarse de forma segura.
- El sistema no puede distinguir metadatos confiables de contenido de documentos no confiables.

- La respuesta requiere el state exacto de la base de datos, que debe provenir de un tool tipado, no de una búsqueda semántica.
- Las citas, IDs de fuente o trases de retrieval no pueden almacenarse para revisión.
- El sistema dependería de volcar transcripciones sin procesar en vez de usar un context builder controlado.

## Arquitectura

Usa este diagrama para leer Semantic Recall y RAG como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: la memory o la capa de retrieval poseen el conocimiento de larga duración, mientras que el agent posee el working state local al task.



## Forma del sistema

- **Límite del pattern:** el context builder decide qué puede entrar al working set del model y cómo se etiqueta cada parte.
- **Propietario del state:** el runtime posee el working state, políticas, presupuestos, retrieval trases, resultados de tools, registros de memory y registros de ensamblaje de context.
- **Rol del model:** el model usa el paquete ensamblado para responder, actuar, pedir evidencia faltante o explicar por qué no puede continuar.
- **Límite de policy:** elegibilidad de fuentes, acceso de tenant, redacción, frescura, manejo de resultados de tools, escrituras de memory y jerarquía de instrucciones se ejecutan antes de la generación.
- **Promesa operativa:** el model trabaja a partir de un paquete pequeño, etiquetado y verificado por policy, en vez de memory vaga o context sin filtrar.

## Protocolo central

1. Clasifica la necesidad de context: policy, instrucciones del task, working state, resultado de tool, memory del usuario, historial de eventos, registro privado, fuente pública o ejemplo.
2. Carga el goal actual, working memory, estado de presupuesto, estado de aprobación y condiciones de detención.
3. Aplica filtros de solicitante, tenant, rol, fuente, frescura y manejo de datos antes de la recuperación o inclusión.
4. Recupera candidate chunks con metadatos, no solo texto plano.
5. Ordena y recorta candidatos por relevancia, frescura, diversidad, confianza y presupuesto de tokens.
6. Construye un set de evidencia con IDs de fuente, citas, confianza y brechas conocidas.



7. Ensambla el context con instrucciones, policy, state, evidencia, resultados de tools, memory, ejemplos y exclusiones por separado.
8. Genera una respuesta, acción, rechazo o escalamiento usando solo el context elegible.
9. Registra la consulta, filtros, IDs de fuente, secciones del paquete, material omitido, citas y motivo de detención.

## Notas de implementación

Trata el material recuperado como evidencia, no como autoridad. Una página web, correo electrónico, ticket, PDF o documento puede contener hechos útiles e instrucciones maliciosas al mismo tiempo.

### Context Packet

Un context packet es el artifact del runtime que explica qué se le permitió ver al model.

```
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;
};

```

Este paquete debe ser trazable. Si una ejecución falla, deberías poder inspeccionar qué se incluyó, qué se omitió y por qué.

## Reglas de orden y confianza

El orden es parte del control plane. Un orden práctico es:

1. instrucciones del sistema;
2. restricciones de policy y seguridad;
3. solicitud actual del usuario o workflow;
4. goal y working memory;
5. resultados aprobados de tools;
6. evidencia recuperada;
7. memories recuperadas;
8. ejemplos;
9. exclusiones explícitas y brechas conocidas.

El punto no es solo el orden de los tokens. El punto es la separación. El texto recuperado, la salida de tools, la memory y los documentos del usuario nunca deben poder reescribir las instrucciones del sistema, la policy, los permisos de tools, las reglas de aprobación o las reglas de escritura de memory.

## 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

El evidence contract debe viajar con la respuesta:

```

type RagAnswer = {
 status: 'answered' | 'missing_evidence' | 'conflicting_evidence' | 'refused';
 answer?: string;
 citations: string[];
 evidenceRefs: string[];
 missingEvidence?: string[];
};

```

## Eligibility Check

Un pequeño eligibility check previene muchas fallas en producción:

```

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';
}

```

No dejes que el model decida si una fuente está permitida. El model puede resumir la calidad de la evidencia. El software debe hacer cumplir la elegibilidad.

## Context Budgeting

El context budgeting no es solo trancar. Es decidir qué debe ver el model, qué se puede resumir, qué se puede recuperar de nuevo, qué se puede omitir y qué debe fijarse.

Fija elementos pequeños y de alta autoridad: instrucciones del sistema, restricciones de policy, goal activo, condición de detención, estado de aprobación y la solicitud del usuario. Resume historial voluminoso pero de bajo riesgo. Recupera evidencia de fuente en vez de cargar viejos fragmentos copiados. Elimina chunks duplicados. Omite material obsoleto, no autorizado o de baja confianza. Reserva tokens de salida antes de llenar el context de entrada.

Cuando el presupuesto es limitado, el runtime debe degradar explícitamente:

- responder con menos fuentes citadas;
- hacer una pregunta aclaratoria;
- recuperar de nuevo con filtros más estrictos;
- resumir el state intermedio;
- rechazar cuando la evidencia requerida no cabe de forma segura;
- escalar cuando la policy requiera evidencia completa.

## Modos de falla

- Se recupera evidencia obsoleta pero plausible y se trata como actual.
- El texto recuperado contiene instrucciones que sobrescriben el goal del sistema.
- La salida de tool se trata como una nueva instrucción en lugar de como datos.
- La memory sobrescribe la solicitud actual del usuario.
- El working state está obsoleto pero aún se incluye como actual.
- Fragmentos del tenant, rol o cliente incorrecto entran al context.
- El retriever devuelve fuentes semánticamente similares pero operativamente incorrectas.
- Las citas apuntan a documentos generales en vez del fragmento exacto de soporte.
- El agent responde cuando falta evidencia o hay conflicto.
- Las escrituras en memory almacenan un resumen no verificado como si fuera un hecho durable.
- Se registran los puntajes de retrieval, pero no los filtros, IDs de fuente ni la frescura.
- Se evalúa la respuesta final, pero no la calidad del retrieval.
- El context packet no puede reconstruirse después de un incidente.
- El policy o approval state importante se omite bajo presión de tokens.

## Estrategia de evaluación

Los context evals deben probar retrieval, ensamblado y comportamiento de respuesta por separado.

- Prueba preguntas de respuesta conocida donde la fuente correcta está presente.
- Prueba casos de evidencia faltante donde el agent debe rechazar o pedir ayuda.
- Prueba casos de fuente obsoleta donde una fuente antigua entra en conflicto con una nueva.
- Prueba casos de fuentes en conflicto donde la respuesta debe explicar la incertidumbre.
- Prueba prompt injection dentro de documentos recuperados.
- Prueba prompt injection dentro de resultados de tools y registros de memory.
- Prueba límites de tenant y permisos.
- Prueba presión de presupuesto de context y verifica que policy, goal y citas no se omitan.
- Prueba comportamiento de fuente omitida y verifica que el trace explique por qué se excluyó material.
- Prueba cobertura de citas: cada afirmación factual debe mapearse a evidencia de fuente.
- Prueba precisión y recall de retrieval antes de evaluar la calidad final del texto.

Un fixture de eval compacto puede hacer explícito el requerimiento de evidencia:

```
{
 "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"]
 }
}
```

Mide recall de retrieval, precisión de retrieval, frescura de fuente, completitud del packet, eficiencia de tokens de context, fidelidad de citas, tasa de rechazo por falta de evidencia, resistencia a prompt-injection, violaciones de límites de tenant y calidad de respuesta fundamentada en la evidencia citada.

## Lista de verificación para producción

- Define fuentes elegibles por tenant, rol, tipo de fuente, frescura y clase de datos.
- Mantén metadatos de fuente con cada fragmento recuperado.
- Construye un context packet trazable para cada ejecución.
- Separa instrucciones de hechos recuperados en el context ensamblado.
- Mantén resultados de tools, memory, evidencia recuperada, ejemplos y contenido de usuario etiquetados por separado.
- Redacta o excluye fuentes antes de la generación, no después.
- Fija policy, goal activo, condición de detención y approval state antes de llenar context opcional.
- Exige citas para afirmaciones factuales.
- Rechaza o escala cuando la evidencia falta, está obsoleta o en conflicto.
- Traza consulta, filtros, IDs de fuente, puntajes, secciones del packet, referencias omitidas y citas finales.
- Evalúa la calidad del retrieval por separado de la calidad de la respuesta.
- Revisa escrituras en memory antes de almacenar resúmenes recuperados como hechos durables.
- Versiona chunking, embedding model, filtros de retrieval, rerankers, prompts y reglas de citas.
- Agrega regression evals para context poisoning, context obsoleto, policy omitido y presión de tokens.

La regla arquitectónica es simple: cada elemento en el context necesita una razón, una fuente, un nivel de confianza y un presupuesto. Continúa con [Context Budgets and Working Sets](#) para límites de packet y [Agentic RAG Systems](#) para composición a nivel de sistema.

## Recorrido de código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo circundante explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o flujo de control.

## Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el repositorio fuente.

`context-engineering-pattern/langgraph_python_example/rag_example.py`

[Ver código fuente completo](#)

```
"""
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)
```

## Descarga

- [Descargar bundle de código fuente](#)
- [Abrir carpeta de código fuente](#)

El bundle de descarga contiene la carpeta `context-engineering-pattern/` actual de este repositorio.

## Patrones relacionados

- [Context Budgets and Working Sets](#)
- [Working Memory](#)
- [Memory-Augmented Agent](#)
- [Knowledge-Bound Agents](#)
- [Agentic RAG Systems](#)
- [Pattern Evaluation Checklist](#)



# Working Memory

Working memory es un state de task compacto y tipado que el agent puede actualizar y consultar durante una ejecución.

## Fuente y descargas

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

## Intent

El patrón Goals and State separa lo que el agent intenta lograr del state mutable que acumula mientras trabaja. Los goals definen el éxito. El state registra el progreso, restricciones, evidencia, trabajo pendiente, aprobaciones, presupuesto y razones de detención.

Este es el núcleo práctico de working memory. Working memory no es long-term memory. Es un state operativo con alcance de run. Debe ser compacto, tipado, inspeccionable y desechable, a menos que otra policy promueva explícitamente parte de él a durable memory.

Usa este pattern para state local de task. Durable memory se maneja con [Memory-Augmented Agent](#). El ensamblaje de context se maneja con [Context Engineering](#).

El model puede proponer un siguiente paso, una actualización de state o una razón de detención. El runtime es dueño de la transición de state. Ese límite evita que el agent reescriba silenciosamente lo que ha sucedido, lo que queda o por qué se le permite continuar.

## Usar cuando

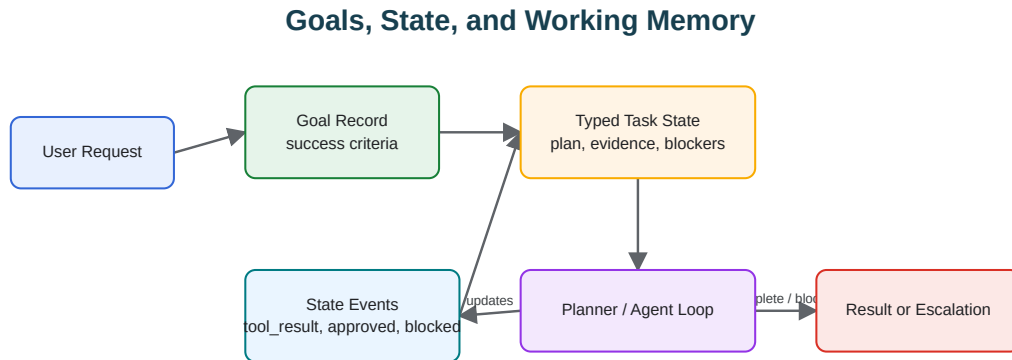
- Una task abarca múltiples turnos, tools, agents, reintentos, aprobaciones o pasos de workflow.
- Necesitas ejecución reanudable después de fallas, interrupciones, timeouts o aprobación humana.
- El agent debe explicar el progreso respecto a un objetivo explícito.
- Necesitas un conjunto de trabajo compacto en vez de solo el historial de chat.
- La evaluación depende de la trayectoria, no solo de la respuesta final.

## Evitar cuando

- La task es stateless y puede responderse en una sola llamada.
- El goal no puede expresarse como criterios de éxito observables.
- El state contendría datos sensibles que no puedes almacenar de forma segura.
- El sistema usaría el state como un scratchpad no estructurado y sin dueño.
- El runtime no puede reproducir ni inspeccionar las transiciones de state.

## Arquitectura

Usa este diagrama para leer Working Memory como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: la capa de memory o retrieval posee el conocimiento de larga duración, mientras que el agent es dueño del working state local de la task.



## Forma del sistema

- **Límite del pattern:** el límite de state posee el goal, working state, event log, reglas de transición, razón de detención y registro de replay.
- **Dueño del state:** el runtime, workflow engine o servicio de aplicación es dueño del state; el model solo posee las actualizaciones propuestas.
- **Rol del model:** el model puede resumir el progreso, identificar brechas y proponer actualizaciones, pero no debe ser la fuente de verdad para el state.
- **Límite de policy:** los cambios de state que afectan permisos, aprobaciones, promoción de memory o efectos secundarios pasan por revisiones del runtime.
- **Promesa operativa:** el sistema puede explicar qué está haciendo, qué cambió, qué evidencia causó el cambio y por qué se detuvo o continuó.

## Protocolo central

1. Crear un goal record con criterios de éxito, restricciones, dueño, clase de riesgo y condiciones de detención.
2. Inicializar working memory con el state mínimo necesario para comenzar.
3. Ejecutar un paso acotado mediante un planner, llamada al model, tool, worker, evaluator o puerta de aprobación.
4. Convertir observaciones en eventos de state tipados.
5. Aplicar transiciones de state de forma idempotente.
6. Recalcular preguntas abiertas, trabajo pendiente, state de presupuesto, state de aprobación y razón de detención.
7. Continuar, reintentar, pedir aprobación, escalar o detener según reglas explícitas.
8. Persistir suficiente state e historial de eventos para reproducir o auditar la ejecución.

## Notas de implementación

El state debe ser más pequeño que la transcripción y más estructurado que un resumen. Es el modelo operativo para la run actual.

## 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

El state debe cambiar a través de eventos, no por mutación oculta.

```
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 };
 }
}
```

Esto es intencionalmente simple. Lo importante es la regla: las transiciones de state son explícitas, tipadas, reproducibles y ligadas a observaciones.

### Caso en ejecución: Goal y State de reembolso

Un asistente de reembolsos necesita más que una transcripción de chat porque los operadores deben saber qué evidencia usó el sistema antes de mover dinero. El goal record debe nombrar el resultado y el riesgo; el working state debe mostrar la evidencia, preguntas abiertas, state de aprobación y razón de detención.

```
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"
```

Este state le dice al runtime lo que sucedió y lo que queda. También le da a los evals algo concreto para inspeccionar: la ausencia de policy debe bloquear la recomendación, una aprobación denegada debe bloquear el pago y una run reanudada debe preservar las mismas referencias de evidencia.

## Reglas de promoción

Working memory no debe convertirse automáticamente en durable memory. Un paso completado, resultado de tool o corrección de usuario puede ser candidato para long-term memory, pero la promoción requiere una decisión de policy aparte.

Antes de promover working memory, pregunta:

- ¿Esto es útil más allá de la run actual?
- ¿La fuente es confiable y está referenciada?
- ¿El usuario o tenant permite que se almacene?
- ¿Hay datos sensibles que requieren redacción o expiración?
- ¿Es esto un evento, un hecho, una preferencia o una referencia de policy?
- ¿Existe un camino de corrección y eliminación?

Si la respuesta no es clara, mantenlo en el run trace y no lo promociones.

## Modos de falla

- El state se convierte en un volcado de transcripción.
- El model reescribe silenciosamente el state sin un evento.
- El goal describe actividad, no éxito.
- Los subgoals se alejan del goal principal.
- Los resultados de tool se resumen sin referencias de evidencia.
- El state de aprobación, cancelación o presupuesto se pierde tras un reintento.
- El sistema no puede reanudar porque el state solo vivía en el context del model.
- Un state obsoleto sobrescribe resultados frescos de tool.
- El state sensible se persiste sin reglas de retención o redacción.
- Working memory se promueve automáticamente a long-term memory.

## Estrategia de Evaluación

Los evals de working-memory deben inspeccionar la trayectoria, no solo la respuesta final.

- Prueba que una task de múltiples pasos crea un goal con criterios de éxito observables.
- Prueba que cada resultado de tool se convierte en un evento de state con una referencia de evidence.
- Prueba reintentos y verifica transiciones de state idempotentes.
- Prueba reanudar después de una interrupción.
- Prueba cancelación y verifica que la run no continúe.
- Prueba esperas de aprobación y verifica que el approval state sobreviva al reanudar.
- Prueba agotamiento de presupuesto y verifica que la stop reason quede registrada.
- Prueba resultados de tool obsoletos y verifica que la evidence más reciente prevalezca.
- Prueba runs bloqueadas y verifica que las preguntas abiertas se conserven.
- Prueba que la working memory no se promueva a durable memory sin policy.

Mide integridad del state, validez de transiciones, éxito de replay, éxito de resume, tasa de stale-state, tasa de lost-approval, precisión de stop-reason y precisión de promotion-policy.

## Lista de Verificación para Producción

- Define goal schema, state schema, event schema y stop reasons.
- Mantén la working memory compacta y tipada.
- Almacena el state separado del historial de chat.
- Haz que las actualizaciones de state sean basadas en eventos, idempotentes y reproducibles.
- Adjunta referencias de evidence a resultados de tool, decisiones, aprobaciones y respuestas finales.
- Persiste approval, cancelación, presupuesto, retry y blocked state.
- Trata la promoción de state a durable memory como una decisión de policy separada.
- Redacta o evita datos sensibles en la working memory cuando sea posible.
- Traza lecturas, escrituras, transiciones, reintentos, reanudaciones y stop reasons del state.
- Convierte incidentes de lost-state y wrong-stop en regression evals.

## Recorrido de Código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo circundante explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o flujo de control.

## Código Fuente

Este pattern actualmente no tiene un extracto de código dedicado. Usa los enlaces de source y descarga a continuación para obtener la carpeta completa del pattern.

## Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `goals-and-state-pattern/` actual de este repositorio.

## Patrones Relacionados

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

# Knowledge-Bound Agents

Los knowledge-bound agents fundamentan respuestas y acciones en fuentes aprobadas, políticas y reglas de citación.

## Fuente y descargas

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

## Propósito

La enforcement de policy es el límite controlado por software que decide si se permite una respuesta propuesta por el model, una llamada a tool, acceso a datos, escritura en memory o un efecto secundario. El model puede proponer una acción. El runtime decide si la permite, la niega, requiere aprobación, la escala o la audita.

La policy no debe vivir solo en el texto del prompt. Los prompts pueden explicar la policy al model, pero la enforcement pertenece al código, workflow, manifiestos de tools, sistemas de control de acceso y registros de decisiones auditable.

Los knowledge-bound agents usan la misma idea para las respuestas: el model solo debe responder a partir de fuentes aprobadas, citar la evidencia y rechazar o escalar cuando la fuente requerida está ausente, desactualizada, prohibida o en conflicto.

La regla práctica es: la policy se ejecuta antes de la autoridad. Antes de la recuperación, antes de escrituras en memory, antes de la ejecución de tools, antes de la comunicación externa y antes de respuestas finales en dominios regulados o ligados a evidencia, el runtime debe saber si la acción está permitida.

## Usar cuando

- Las acciones deben ser verificadas antes de ejecutarse.
- El agent maneja datos privados, regulados, sensibles a la seguridad o críticos para el negocio.
- El sistema necesita fuentes aprobadas, citaciones o restricciones de cumplimiento.
- Las decisiones de policy deben ser auditables y reproducibles.
- El runtime puede identificar actor, recurso, acción, capability, riesgo y context.
- Las llamadas a tools, escrituras en memory, recuperación, respuestas finales o transiciones de workflow requieren reglas diferentes según la clase de task.
- Existe una ruta de aprobación humana para acciones que son válidas pero demasiado riesgosas para ejecutarse de forma autónoma.

## Evitar cuando

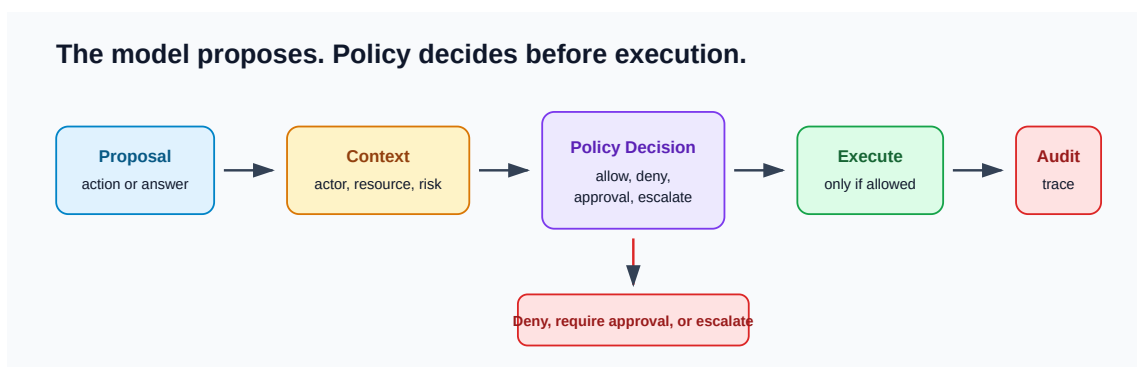
- La policy solo está escrita como guía en el prompt sin verificación en runtime.



- El sistema no puede identificar el actor, recurso, acción y context.
- Las verificaciones de policy ocurren después de acciones irreversibles.
- Las excepciones son silenciosas, no revisadas o faltan en los traces.
- No se pueden identificar, actualizar o citar las fuentes de conocimiento aprobadas.
- El runtime no puede detener, pausar o cambiar la ejecución después de una decisión de policy.

## Arquitectura

Usa este diagrama para entender Knowledge-Bound Agents como un límite del sistema, no solo como una forma de código. La pregunta clave de propiedad es: la capa de memory o recuperación posee el conocimiento de larga duración, mientras que el agent posee el state de trabajo local de la task.



## Forma del sistema

- **Límite del pattern:** el límite de policy evalúa acciones propuestas, acceso a datos, respuestas y escrituras en memory antes de que tengan efecto.
- **Propietario del state:** el runtime posee el policy context, registros de decisión, versión de policy, trace ID y resultado de enforcement.
- **Rol del model:** el model propone una acción o respuesta y puede explicar el riesgo, pero no se otorga permiso a sí mismo.
- **Límite de conocimiento:** las fuentes aprobadas, vigencia, citaciones y reglas de rechazo definen lo que el agent puede afirmar.
- **Límite de presupuesto:** la policy puede requerir aprobación, degradar capability o detenerse cuando una ejecución ha excedido el gasto o autonomía permitidos para su clase de riesgo.
- **Promesa operativa:** las decisiones de policy ocurren antes de la ejecución y son visibles después de la ejecución.

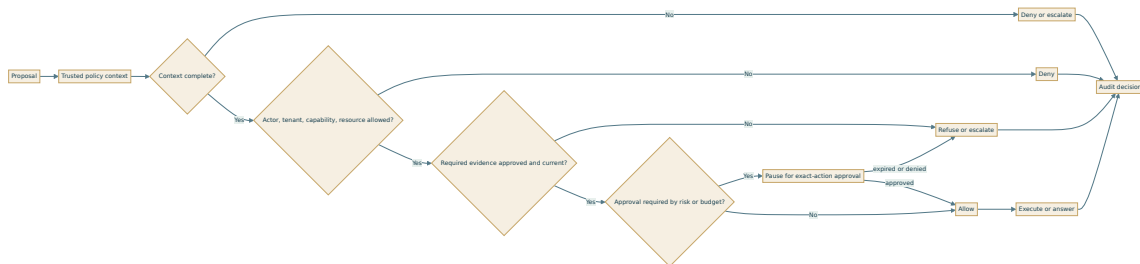
## Protocolo central

1. Recibir una acción propuesta, respuesta, llamada a tool, resultado de recuperación o escritura en memory.
2. Construir el policy context: actor, caller, tenant, recurso, capability, riesgo, evidencia y trace ID.
3. Evaluar la policy antes de que la acción se ejecute o la respuesta se devuelva.
4. Retornar una decisión: permitir, negar, requerir aprobación, escalar o solo auditar.
5. Si se requiere aprobación, pausar en la puerta de aprobación.
6. Ejecutar solo las decisiones que estén permitidas o aprobadas.
7. Registrar decisión, motivo, versión de policy, actor, recurso, acción y trace ID.

8. Alimentar negaciones graves, omisiones y sobrescrituras en los regression evals.

## Notas de implementación

Usa este flujo de decisión al revisar dónde se ejecuta la policy. Cada rama debe producir un evento de trace con la versión de policy, motivo, actor, recurso, capability y efecto de ejecución.



Una decisión de policy debe ser un objeto tipado en runtime.

```
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;
};
```

El policy context debe provenir de state confiable en runtime, no solo del texto del model:

```
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;
};

```

La función de enforcement debe ejecutarse antes de la recuperación, escritura en memory, llamada a tool, efecto secundario o respuesta final:

```

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' };
}
```

Para respuestas knowledge-bound, la policy también decide si la evidencia está permitida:

```
type SourcePolicy = {
 sourceId: string;
 approved: boolean;
 freshness: 'current' | 'stale' | 'unknown';
 citationRequired: boolean;
 allowedTenant?: string;
};
```

El model puede explicar por qué una acción parece segura. El runtime aún toma la decisión.

## Dónde se ejecuta la policy

| Límite                 | Pregunta de policy                                                              |
|------------------------|---------------------------------------------------------------------------------|
| Recuperación           | ¿Está permitido que este actor lea estas fuentes para esta task?                |
| Llamada a tool         | ¿Está permitido este tool para el actor, tenant, recurso, riesgo y presupuesto? |
| Escritura en memory    | ¿Esta memory es segura, acotada, útil y permitida para persistir?               |
| Aprobación humana      | ¿Esta acción solo está permitida después de revisión, y quién puede aprobarla?  |
| Respuesta final        | ¿La respuesta está respaldada por evidencia aprobada y es segura para devolver? |
| Transición de workflow | ¿El siguiente paso es válido después del current state y la decisión de policy? |

Trata cada decisión de policy como un evento en runtime. Debe tener un trace ID, versión de policy, resumen de entrada, decisión, motivo y efecto de ejecución.

## Failure Modes

- La policy existe solo en el system prompt.
- La policy se ejecuta después de que un tool ya se ejecutó.
- La decisión carece de actor, resource, tenant o capability context.
- Un retry omite la policy porque el primer intento ya fue verificado.
- Las versiones de la policy cambian pero los traces no registran qué versión se aplicó.
- Los denials no se registran, por lo que los operadores no pueden ver acciones inseguras intentadas.
- Las acciones que requieren aprobación se tratan como permitidas.
- Las respuestas de knowledge citan fuentes no aprobadas, obsoletas o inaccesibles.
- Las exceptions se convierten en agujeros de policy permanentes y no documentados.

- Las verificaciones de policy ignoran el budget state, por lo que un agent puede seguir gastando después de que debería requerirse aprobación.
- Las escrituras en memory omiten la policy porque se tratan como gestión de context inofensiva.
- Las respuestas finales omiten la policy incluso cuando el dominio requiere evidencia aprobada.

## Evaluation Strategy

Las policy evals deben probar los caminos permitidos, denegados, que requieren aprobación y de escalamiento.

- Prueba acciones permitidas de bajo riesgo.
- Prueba acciones denegadas a través de los límites de role, tenant, resource y capability.
- Prueba acciones que requieren aprobación antes de efectos secundarios.
- Prueba fuentes obsoletas o no aprobadas en respuestas knowledge-bound.
- Prueba retries y workflows reanudados para asegurar que la policy se aplique cada vez.
- Prueba context faltante de actor, resource o tenant.
- Prueba cambios de versión de policy y la completitud de auditoría.
- Prueba incidentes de producción como fixtures de policy reproducibles.
- Prueba casos de umbral de budget que requieren aprobación antes de más trabajo.
- Prueba denegaciones, aprobaciones y alcance de tenant en escrituras de memory.
- Prueba rechazo o escalamiento de respuesta final cuando falta la evidencia requerida.

Un policy eval compacto puede verse así:

```
{
 "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"]
 }
}
```

Mide la precisión de las decisiones de policy, tasa de false allow, tasa de false denial, precisión en el enrutamiento de aprobaciones, violaciones de límites de tenant, uso de policy obsoleta, completitud del logging de denials y recurrencia de fallas de policy conocidas.

Para sistemas en producción, el false allow suele ser la métrica más peligrosa. Un false denial puede frustrar a un usuario. Un false allow puede filtrar datos, mover dinero, enviar el mensaje equivocado o crear un incidente.

## Production Checklist

- Haz cumplir la policy antes de la ejecución, retorno de respuesta, escritura en memory o comunicación externa.
- Construye el policy context a partir de datos confiables del runtime, no solo texto del model.
- Devuelve decisiones explícitas de allow, deny, require-approval, escalate o audit.
- Registra actor, resource, capability, reason, policy version y trace ID.
- Aplica la policy en retries y workflows reanudados.
- Mantén versionadas las policy versions, tool manifests, source rules y approval rules.
- Trata el policy context faltante como deny o escalate.
- Aplica la policy a escrituras en memory y respuestas finales, no solo tools.
- Conecta las decisiones de policy al budget state del runtime.
- Agrega dashboards para denials, approvals, overrides y false allows.
- Convierte policy misses en regression evals.
- Revisa exceptions y expíralas intencionalmente.

## Code Walkthrough

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo que lo rodea explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

## Source Code

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el source del repositorio.

`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
 };
}

```

Extracto truncado para facilitar la lectura. Descarga el bundle o abre el archivo fuente para la implementación completa.

## Download

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

El bundle de descarga contiene la carpeta `compliance-policy-enforcer-agent/` actual de este repositorio.

## 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 empaquetan conocimiento procedimental como carpetas descubribles y versionadas de instrucciones, referencias, scripts, plantillas, recursos y pruebas.

## Fuente y descargas

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

## Intento

El Skills Pattern empaqueta conocimiento procedimental como carpetas descubribles: instrucciones concisas, referencias, scripts, plantillas y pruebas que un agent carga solo cuando son relevantes.

Esta carpeta incluye un pequeño paquete de skill para release-notes. Muestra la estructura que debe tener un skill de producción: `SKILL.md` para activación y procedimiento, referencias para policy, plantillas para salida estable, fixtures para ejemplos repetibles y un runner/test en TypeScript que un humano o agent puede ejecutar.

## Escenario

Un equipo de plataforma quiere que coding agents preparen release notes, actualicen un changelog y recolecten evidencia de verificación. Un tool schema puede describir “write release notes”, pero no puede contener la release rubric del equipo, ejemplos, comandos requeridos, policy de source-link y checklist final.

Un skill es el mejor límite. `SKILL.md` da la regla de activación y el procedimiento corto. Los archivos de referencia contienen la release policy y ejemplos. Los scripts recolectan versión, diff y evidencia de pruebas. Las plantillas mantienen estable la forma de la salida. El agent carga archivos más profundos solo cuando aparece el release task, así que el trabajo de codificación normal no paga el costo de context.

La pregunta importante es si la carpeta contiene un procedimiento repetible que otro ingeniero pueda revisar, ejecutar, versionar y probar.

## Usar cuando

- Una capability requiere un procedimiento de dominio repetible y no solo una tool API.
- Quieres know-how reutilizable entre agents, equipos o proyectos.
- El agent se beneficia de disclosure progresivo: instrucciones cortas primero, referencias más profundas solo cuando se necesitan.

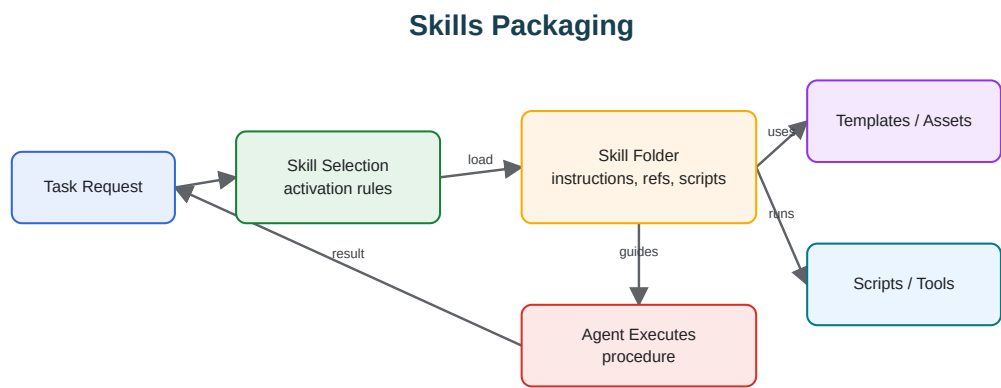
## Evitar cuando

- Un simple tool schema describe completamente la capability.

- El skill incluiría secretos, credenciales o scripts inseguros.
- Las instrucciones son demasiado vagas para probarse con tareas reales.

## Arquitectura

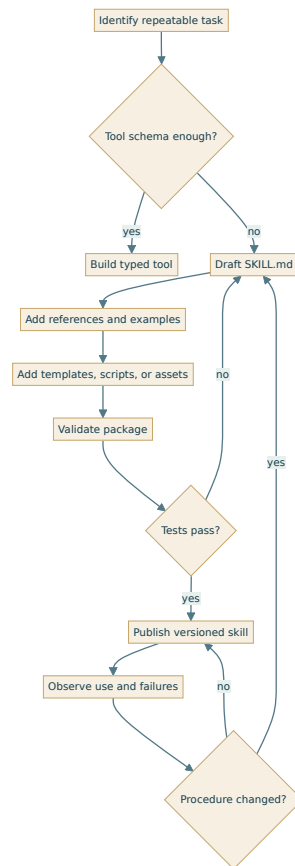
Usa este diagrama para leer Skills como un límite de sistema, no solo una forma de código. La pregunta clave de ownership es: el protocolo o el límite de capability posee schemas, permisos, registros de invocación y validación de respuestas.



## Reglas de decisión

Usa un skill cuando el model necesita juicio procedimental alrededor de una capability. Usa un tool cuando el model solo necesita llamar una operación tipada.

| Necesidad                                                                      | Preferir                              | Razón                                                                                   |
|--------------------------------------------------------------------------------|---------------------------------------|-----------------------------------------------------------------------------------------|
| Llamar una API estable con entrada y salida tipada.                            | Tool                                  | El schema y la regla de autorización contienen todo el contrato.                        |
| Seguir un procedimiento de equipo de varios pasos.                             | Skill                                 | El agent necesita instrucciones, ejemplos, plantillas y verificaciones.                 |
| Producir un artifact repetido como una PR note, ADR, reporte o release packet. | Skill                                 | Las plantillas y ejemplos reducen el drift.                                             |
| Ejecutar una acción peligrosa.                                                 | Tool detrás de policy, no solo skill. | Skills pueden explicar el procedimiento, pero los permisos deben estar fuera del texto. |
| Compartir capability entre agents o equipos.                                   | Skill más scripts probados.           | Humanos y agents necesitan el mismo contrato ejecutable.                                |



## Forma del sistema

- **Límite del pattern:** el agent descubre o selecciona una capability, envía una solicitud tipada y recibe un resultado tipado a través de un policy boundary.
- **Propietario del state:** el protocolo o el límite de capability posee schemas, permisos, registros de invocación y validación de respuestas.
- **Artifact principal:** `skills-pattern/` contiene un paquete de skill para release-notes revisable con instrucciones de activación, referencia de policy, plantilla, fixture, runner en TypeScript y pruebas.
- **Promesa operativa:** Skills empaquetan conocimiento procedimental como carpetas descubribles y versionadas de instrucciones, referencias, scripts, plantillas, recursos y pruebas.
- **Ruta ejecutable:** comienza con `npm run skills:demo` antes de adaptar el pattern a un sistema más grande.

## Contrato

Un skill de producción debe tener una anatomía pequeña y revisable.

| Parte                    | Propietario                                                                      | Prueba A++                                                                          |
|--------------------------|----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| <code>SKILL.md</code>    | Regla de activación, procedimiento corto y enrutamiento a material más profundo. | Un nuevo agent puede decidir cuándo usar el skill sin cargar todas las referencias. |
| <code>references/</code> | Policy de dominio, ejemplos, rúbricas y edge cases.                              | Las referencias están acotadas, actualizadas y citadas por el skill.                |

| Parte                          | Propietario                                               | Prueba A++                                                                                   |
|--------------------------------|-----------------------------------------------------------|----------------------------------------------------------------------------------------------|
| <code>scripts/</code>          | Recolección, validación, generación o formateo repetible. | Un humano puede ejecutar los scripts fuera del agent loop.                                   |
| <code>templates/</code>        | Forma de salida estable para artifacts comunes.           | Las salidas se mantienen consistentes entre ejecuciones y agents.                            |
| <code>tests/</code> o fixtures | Casos positivos y negativos.                              | Activación incorrecta, entradas faltantes, salidas inseguras y artifacts malformados fallan. |
| Manifest o metadata            | Versión, propietario, dependencias, permisos y entorno.   | Los revisores pueden auditar riesgos de supply-chain y permisos.                             |

## Bad Skill vs Production Skill

| Weak Skill                                        | Production Skill                                                                              |
|---------------------------------------------------|-----------------------------------------------------------------------------------------------|
| "Use this for writing."                           | "Use this for release notes from verified engineering evidence."                              |
| Un solo archivo largo de instrucciones.           | <code>SKILL.md</code> corto más referencias enrutadas.                                        |
| Consejos vagos y ejemplos copiados en el context. | Plantillas, fixtures y scripts con salida determinista.                                       |
| Suposiciones de shell ocultas.                    | Comandos explícitos, dependencias, entradas, salidas y modos de fallo.                        |
| Sin casos negativos.                              | Pruebas de activación incorrecta, evidencia faltante, solicitud insegura y salida malformada. |
| El texto del skill implica permiso.               | Policy y autoridad de ejecución viven fuera del texto.                                        |

## Core Protocol

1. Descubre la capability, schema, permisos y restricciones operativas.
2. Prepara una solicitud tipada a partir del goal y state actuales.
3. Autoriza la solicitud antes de la invocación.
4. Invoca el tool, skill o remote agent y valida el resultado.
5. Devuelve structured output, rechazo, progreso o error sin perder correlation IDs.

## Notas de implementación

- Mantén `SKILL.md` corto y enruta a archivos más profundos solo cuando sea necesario.
- Incluye scripts y plantillas en vez de pedirle al model que recree artifacts frágiles.
- Trata los skills como insumos de supply-chain: revisa, versiona, prueba y restringe la ejecución.
- Incluye ejemplos de uso exitoso y fallido.
- Registra propietario, versión, dependencias y ruta de rollback antes de distribuir un skill.
- Prefiere scripts ejecutables para formateo frágil, recolección de evidencia o validación.

- Mantén credenciales en almacenes de plataforma o variables de entorno, nunca dentro de archivos de skill.

## Skills CLI-First

Un skill útil debe poder ser invocado tanto por un humano como por un agent. Una interfaz de línea de comandos suele ser el contrato compartido más simple:

- un comando por capability;
- subcomandos predecibles como `list`, `get`, `create` y `run`;
- salida estructurada para agents y salida legible para humanos;
- valores predeterminados no interactivos con flags explícitos como `--yes` o `--force`;
- credenciales desde el entorno o almacenes de plataforma, no prompts ocultos dentro del comando.

Esto mantiene el skill comprobable fuera del agent loop. Si un humano no puede ejecutar el skill directamente e inspeccionar la salida, será más difícil depurar el agent cuando el skill falle.

## Modos de fallo

- Descripciones de skill demasiado amplias, causando activación irrelevante.
- Archivos largos de instrucciones que consumen context antes de entender la tarea.
- Dependencias ocultas que solo funcionan en una máquina.
- Scripts maliciosos o desactualizados incluidos.
- Texto de prompt que amplía silenciosamente la autoridad de tool o filesystem.
- Referencias que entran en conflicto con `SKILL.md` o entre sí.
- Plantillas que divergen porque ningún fixture prueba el artifact final.
- Falta de ruta de rollback después de que se publica una mala versión de skill.

## Lista de revisión

Antes de agregar un skill a un entorno de agent, verifica:

- La descripción es lo suficientemente específica para que el skill se active solo para las tareas correctas.
- La primera pantalla de instrucciones le dice al agent qué hacer, qué no hacer y qué archivos cargar después.
- Cualquier script es determinista, no interactivo por defecto y seguro para ejecutarse con el menor privilegio.
- Los secretos provienen de almacenes de plataforma o variables de entorno, nunca de texto copiado.
- El skill tiene al menos un ejemplo de éxito y uno de rechazo o mal uso.
- El skill registra suficiente evidencia para que un revisor pueda reproducir el resultado.
- El skill puede deshabilitarse, versionarse o revertirse de forma independiente al prompt del agent.

## Estrategia de evaluación

- Prueba el uso en el happy-path, activación de tareas incorrectas, entradas faltantes, artifacts mal formados, solicitudes inseguras y dependencias ausentes.
- Verifica que la skill cargue solo las referencias necesarias para el task.

- Valida los artifacts generados contra plantillas en lugar de solo revisar la calidad del texto.
- Ejecuta scripts fuera del agent loop para que los humanos puedan reproducir fallos.
- Rastrea la precisión de activación, la completitud de la evidencia, la tasa de fallos de scripts, la validez de artifacts y el éxito del rollback.

## Lista de verificación para producción

- Nombra el owner, versión, conjunto de dependencias y runtime soportado.
- Mantén `SKILL.md` lo suficientemente corto para el uso en la primera carga.
- Protege scripts peligrosos detrás de una policy externa y aprobación.
- Valida entradas, salidas, placeholders de plantillas y evidencia requerida.
- Registra el nombre de la skill, versión, referencias cargadas, scripts ejecutados, artifacts escritos y estado final.
- Fija o revierte la skill de forma independiente al agent prompt.

Usa la lista de verificación descargable del libro en línea al revisar un paquete de skill para producción.

## Ejecuta el ejemplo

```
npm run skills:demo
npm run skills:test
```

## Recorrido por el código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo circundante explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o flujo de control.

## Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el repositorio fuente.

`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}`);
 }
 }
}

```

Extracto truncado para mayor legibilidad. Descarga el bundle o abre el archivo fuente para la implementación completa.

**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");
```

## Descarga

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

El bundle de descarga contiene la carpeta `skills-pattern/` actual de este repositorio.

## Patrones relacionados

- [MCP-first Tool Use](#)
- [Context Engineering](#)
- [Human-in-the-Loop Approval](#)

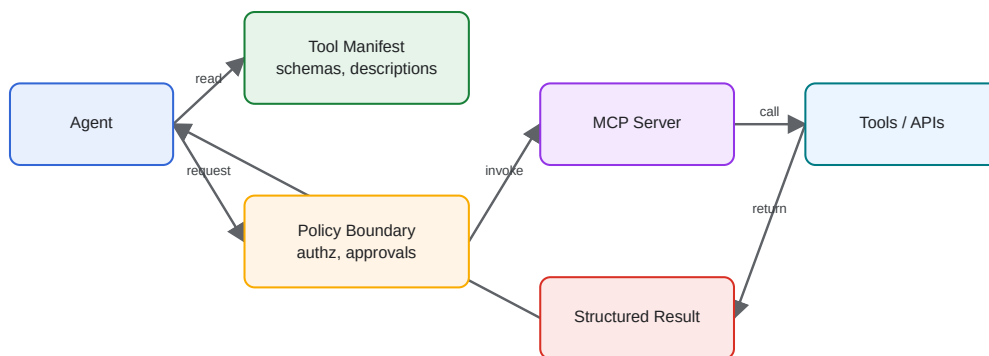
# Tool Capability Design

Los tools son donde los agentic systems se vuelven reales. Antes de los tools, el model solo puede producir texto. Después de los tools, puede leer sistemas, escribir sistemas, gastar dinero, enviar mensajes, cambiar state y activar workflows. Por eso, el diseño de tools no es un detalle de implementación. Es arquitectura.

Un agent fuerte no necesita una lista grande de tools. Necesita los tools correctos, con contratos claros, autoridad limitada, errores útiles y resultados observables.

Descarga la [lista de verificación para revisión de tool capability design](#) cuando revises una nueva tool surface, un MCP server o un set de agent capability.

## MCP-first Tool Use



## The Tool Surface Is A Control Plane

La tool surface decide qué puede hacer el agent, así que no trates los tools como funciones auxiliares que simplemente expones a un model. Trata la surface como un control plane con seguridad, confiabilidad y semántica de producto.

Cada tool debe poder responder qué capability expone, quién puede llamarla, qué inputs son válidos, qué state puede leer, qué state puede cambiar, qué efectos secundarios puede producir, qué policy checks aplican, qué forma de resultado retorna y cómo se traza, reproduce o audita. Cuando esas respuestas no son explícitas, el sistema depende de la disciplina del prompt para mantenerse seguro, que es el lugar más débil para poner un límite.

## Narrow Tools Beat Broad Tools

Los broad tools parecen convenientes y usualmente son peligrosos. Los que debes evitar son los primitivos abiertos:

- `run_sql(query);`
- `execute_shell(command);`
- `send_http_request(method, url, body);`
- `update_customer_record(fields);`

- `manage_ticket(action, payload)`.

Estos obligan al model a inventar el límite de seguridad en runtime, que es exactamente el lugar equivocado. Prefiere tools que codifican el workflow en su lugar:

- `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)`.

Los narrow tools reducen la carga del prompt, mejoran la evaluación y hacen mucho más fácil la aplicación de policy, porque el workflow vive en la tool surface en vez de depender del criterio del model.

## Capability Classes

Clasifica cada tool por capability.

| Capability Class            | Example                                         | Risk                                               |
|-----------------------------|-------------------------------------------------|----------------------------------------------------|
| Read public data            | search docs, fetch public page.                 | información desactualizada o no confiable.         |
| Read private data           | customer lookup, internal database query.       | fuga de privacidad, falla en el límite de tenants. |
| Read untrusted content      | email, webpage, user document, ticket comment.  | prompt injection e instrucciones hostiles.         |
| Write internal state        | update CRM, create ticket, save memory.         | registros corruptos o mal state durable.           |
| External communication      | send email, post message, submit form, webhook. | exfiltración de datos o comunicación no deseada.   |
| Code or shell execution     | run tests, execute script, modify files.        | efectos secundarios arbitrarios.                   |
| Money or entitlement change | refund, purchase, credit, permission grant.     | impacto financiero o de control de acceso.         |

Esta clasificación debe estar en el tool manifest o registro, no solo en el texto.

## Tool Risk Matrix

Las capability classes son útiles, pero los sistemas en producción también necesitan una vista de riesgo operacional. Un tool puede parecer inofensivo en aislamiento y volverse peligroso cuando se combina con datos privados, contenido no confiable, comunicación externa o memory durable.

| Tool Type                 | Primary Risk                                                 | Required Control                                                                           |
|---------------------------|--------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| Read public data          | contenido desactualizado, de baja calidad o hostil.          | metadatos de fuente, señal de frescura y manejo de contenido no confiable.                 |
| Read private data         | fuga de privacidad o falla en el límite de tenants.          | autorización limitada, audit log y redacción de outputs.                                   |
| Read untrusted content    | prompt injection.                                            | aislamiento de contenido, eliminación de instrucciones y policy checks antes de la acción. |
| Write business state      | registros corruptos o avance de workflow no intencionado.    | schema estricto, idempotency key, decisión de policy y trace.                              |
| External communication    | exfiltración de datos o entrega de mensajes no deseados.     | puerta de aprobación, validación de destinatario y auditoría de mensajes.                  |
| Browser or code execution | efectos secundarios arbitrarios y acceso de red oculto.      | sandbox, egress policy, timeout y límites de filesystem/network.                           |
| Memory write              | contaminación durable o falla en la retención de privacidad. | memory policy, atribución de fuente, retention class y ruta de eliminación.                |
| Credential use            | confused deputy o escalamiento de privilegios.               | credenciales de corta duración, scope binding, audience checks y redacción de secretos.    |

El punto no es hacer cada tool pesado. El punto es que el runtime sepa cuáles tools son pesados. Un tool de búsqueda en documentación pública y un tool de pagos no deben vivir detrás de las mismas reglas de policy, trace, retry y aprobación.

## Tool Manifest

Un tool manifest útil es más que un nombre y una descripción.

| Field            | Purpose                                                                      |
|------------------|------------------------------------------------------------------------------|
| Name             | Identificador estable usado en traces, evals y policy.                       |
| Description      | Qué hace el tool, escrito para una selección correcta.                       |
| Input schema     | Campos requeridos, tipos, límites, enums y reglas de validación.             |
| Output schema    | Resultado estructurado, rechazo, resultado parcial o error.                  |
| Capability class | Read, write, communication, code execution, private data, untrusted content. |
| Side effects     | Qué puede cambiar si el tool tiene éxito.                                    |
| Idempotency      | Si llamadas repetidas son seguras y cómo se manejan los duplicados.          |
| Permissions      | Qué roles, agents, rutas o tenants pueden llamarlo.                          |
| Approval rule    | Cuándo la llamada debe ser revisada por un humano o workflow.                |

| Field         | Purpose                                                         |
|---------------|-----------------------------------------------------------------|
| Trace fields  | Correlation ID, actor, resource, policy decision y result.      |
| Data handling | Redacción, retención, privacy class y restricciones de logging. |

Si a un tool le faltan estos campos, el runtime debe tratarlo como de alto riesgo por defecto.

Un manifest compacto puede hacer esos límites explícitos:

```
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

```

El nombre del tool dice lo que el tool hace y lo que no hace. Un tool separado debe enviar el refund después de la aprobación.

Para sistemas fuertemente tipados, el mismo contrato puede representarse como código y registrarse con el 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"],
 },
};
```

Esto parece burocrático solo hasta el primer incidente. Después de eso, se convierte en el mapa de lo que el agent podía hacer, qué versión de policy estaba activa, qué datos cruzaron el límite y qué equipo es responsable de la solución.

## Interfaces Amigables para Agents

Una tool diseñada para humanos no siempre es una tool diseñada para agents. Las tools amigables para agents tienen schemas explícitos, descripciones cortas y específicas, nombres estables, ejemplos de llamadas válidas e inválidas, errores claros, structured outputs, tamaños de resultados acotados, correlation IDs y valores de estado legibles por máquina, sin state global oculto ni efectos secundarios inesperados.

Los mensajes de error tienen más peso del que la gente espera. Un `failed` vago hace que el model adivine. Un error útil dice qué falló, si es seguro reintentar y qué campo necesita corrección, lo cual marca la diferencia entre una recuperación limpia y un retry loop confuso.

## Checklist de Interfaz Tool Agent-First

Diseña cada llamada de tool para que el agent pueda decidir, ejecutar y recuperarse sin adivinar:

- nombra la capability, no el detalle de implementación;
- declara inputs requeridos, inputs opcionales, límites y side effects;

- retorna status legible por máquina, datos de resultado y códigos de error recuperables;
- registra el request id, actor, policy decision y recurso afectado;
- provee modo dry-run o preview para operaciones destructivas;
- documenta el comportamiento fallback cuando la tool es denegada, no está disponible o tiene rate-limit.

La interfaz debe hacer que el camino seguro sea el camino más corto.

## Los Resultados de Tool Son Datos

Los resultados de una tool no deben convertirse en nuevas instrucciones. Un resultado de búsqueda, página web, correo, ticket, documento o línea de log puede contener texto malicioso o irrelevante. El model puede inspeccionarlo como evidencia, pero el runtime no debe permitir que sobrescriba los goals del sistema, permisos de tool, reglas de aprobación o memory policy.

Los buenos resultados de tool mantienen sus partes separadas: metadata confiable, el contenido no confiable en sí, la fuente, timestamp, nivel de confianza, permisos y estado de redacción. Esa separación hace más segura la construcción de context y facilita la evaluación.

## Credenciales y Límites de Egreso

Un agent no debe tener credenciales amplias. El runtime debe intercambiar la identidad del agent, task, tenant y capability aprobada por una credencial limitada en el momento de invocar la tool. Esa credencial debe tener vida corta, audiencia acotada y solo los permisos necesarios para la llamada específica.

La misma idea aplica al egreso de red. Una tool que lee un servicio de órdenes interno no debería poder enviar solicitudes HTTP arbitrarias a internet. Una browser tool no debe acceder a redes privadas a menos que el run explícitamente requiera ese acceso. Una code execution tool no debe heredar por defecto el environment de la máquina del desarrollador.

Como mínimo, las tools de alto riesgo necesitan:

- claims OAuth u OIDC acotados y verificados por el servidor de la tool;
- TLS para transporte service-to-service;
- allowlists de egreso por tool;
- sin secretos ambientales en prompts, logs o resultados de tool;
- binding de tenant y actor en cada request;
- redacción antes de traces, eval fixtures y memory writes;
- soporte de revocación y kill-switch.

Aquí es donde el diseño de la tool se conecta con la arquitectura de servicio. El model selecciona una acción. El runtime autoriza la acción. La tool ejecuta la acción. Esas responsabilidades no deben colapsar en un solo prompt.

## Requisitos de Observabilidad

Las llamadas de tool necesitan traces que expliquen tanto qué ocurrió como por qué el runtime lo permitió. Un trace útil de tool incluye el run ID, agent ID, actor o service principal, nombre y versión de la tool, input shape, resumen redacted del input, policy decision, approval ID cuando aplica, idempotency key, timeout, retry count,

status del resultado, clase de error, latencia, referencia de token context y estado de redacción.

No registres cargas sensibles en bruto solo porque depurar es más fácil así. El trace de producción debe ser suficiente para reconstruir el camino de decisión sin convertir la observabilidad en una fuga de datos secundaria.

Para revisión de incidentes, el trace debe responder:

- ¿Qué tool fue llamada?
- ¿Qué clases de capability estaban activas?
- ¿Qué policy permitió o bloqueó la llamada?
- ¿Había contenido no confiable en el context antes de la llamada?
- ¿Se devolvieron datos privados?
- ¿Datos privados salieron por un canal externo?
- ¿Se requirió aprobación y estaba ligada a la acción exacta?
- ¿La llamada fue replayed, reintentada o deduplicada?

## Divulgación Progresiva de Tools

No muestres cada tool a cada agent en cada paso. Exponlas progresivamente:

1. Comienza con la ruta o tipo de task.
2. Carga el set de tools más pequeño y útil.
3. Agrega tools solo después de que el state lo justifique.
4. Elimina tools cuando cambia la fase.
5. Requiere aprobación o policy checks para cadenas cross-capability.

Un support agent, por ejemplo, podría comenzar con policy de solo lectura y tools de order-lookup. La creación de reembolsos aparece solo después de establecer elegibilidad, el envío de reembolso solo después de que existe un borrador, y la mensajería externa al cliente solo después de aprobación. Divulgar tools de esta forma reduce errores de selección y limita el alcance de prompt injection.

## Composición de Tools

El peligro suele estar en la cadena, no en una sola tool. Una tool de solo lectura de datos privados está bien por sí sola. Una tool de mensajería externa está bien por sí sola. Una browser tool está bien por sí sola. La cadena se vuelve insegura cuando un run combina datos privados, contenido no confiable y comunicación externa, así que evalúa cadenas de tools, no solo llamadas individuales.

Para cada run, pregunta qué capabilities están presentes, si contenido no confiable entró al context, si datos privados pueden salir por algún canal de salida, si la acción de escritura está justificada por evidencia confiable, si el registro de aprobación está ligado a la acción exacta y si un workflow determinista más seguro podría hacer lo mismo. Aquí es donde el diseño de la tool se conecta directamente con el threat modeling.

## Guía de Evaluación

Los tool evals deben probar tanto la selección como la contención. Construye casos donde el comportamiento correcto sea llamar a la tool adecuada, pero también casos donde lo correcto sea no llamar ninguna tool, pedir input faltante, rechazar una acción no soportada, enrutar a aprobación, usar una tool de solo lectura en vez de una de escritura, evitar comunicación externa, detenerse después de que una tool retorna instrucciones no confiables, recuperarse de una respuesta malformada o reintentar solo cuando sea seguro.

Luego mide la precisión de selección de tool, tasa de argumentos inválidos, tasa de llamadas no autorizadas, precisión de enrutamiento de aprobación, prevención de cadenas inseguras, grounding de resultados de tool, latencia y costo por camino de tool, y recuperación de errores de tool. Una respuesta final puede parecer correcta aunque la trayectoria de tool haya sido insegura, así que evalúa la trayectoria.

Los buenos tool evals son vertical slices. No pruebes solo `call_tool(input)`. Prueba el camino completo desde la solicitud del usuario hasta la construcción de context, divulgación de tool, policy decision, invocación de tool, manejo de resultados, comportamiento de memory, respuesta final, trace y replay. Ahí es donde se esconden los bugs reales.

Escenarios útiles de eval incluyen:

- un camino normal y exitoso;
- input requerido faltante;
- input malformado pero recuperable;
- contenido no confiable que intenta sobrescribir instrucciones;
- datos privados seguidos de una solicitud de comunicación externa;
- envío duplicado o retry después de timeout;
- aprobación requerida pero faltante;
- resultado de tool obsoleto;
- tool retorna datos parciales;
- tool deshabilitada por policy;
- memory write solicitado a partir de evidencia no confiable.

El objetivo de la evaluación no es solo si la respuesta final se lee bien. Es si el sistema usó la autoridad mínima necesaria, respetó la policy, preservó la evidencia, evitó cadenas inseguras y dejó un rastro de auditoría útil.

## Checklist de Diseño

Antes de exponer una tool a un agent, verifica:

- ¿La tool es lo suficientemente acotada?
- ¿El input schema es estricto?
- ¿El output es estructurado?
- ¿Los side effects son explícitos?
- ¿La clase de capability está declarada?
- ¿La clase de riesgo está declarada?
- ¿Los permisos se aplican fuera del prompt?
- ¿Las credenciales son de vida corta y acotadas?
- ¿El egreso está restringido a lo que la tool necesita?
- ¿La tool soporta idempotency donde es necesario?

- ¿Los retries son seguros y acotados?
- ¿Las aprobaciones están ligadas a la acción exacta?
- ¿Los errores son accionables?
- ¿Los traces son completos y redactados?
- ¿Datos privados y contenido no confiable están marcados por separado?
- ¿La tool puede ser mockeada en evals?
- ¿La tool puede ser replayed sin repetir side effects inseguros?
- ¿La tool puede ser deshabilitada rápidamente?
- ¿La tool retorna contenido no confiable como datos, no como instrucciones?

La prueba es simple: si el model llama incorrectamente a esta tool, ¿puede la arquitectura detectarlo antes de que ocurra un daño?

## Capítulos Relacionados

- [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

El uso de herramientas MCP-first separa la capability de la tool de la lógica del agent mediante manifests, validación, invocación y resultados estructurados.

## Fuente y descargas

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

## Intención

Usa este pattern cuando el conjunto de tools puede evolucionar de forma independiente al agent. MCP le da a las tools un límite de manifest para que el agent pueda inspeccionar capabilities en lugar de depender de supuestos codificados.

## Úsalo Cuando

- Las tools se comparten entre agents o aplicaciones.
- Los schemas de las tools, el context y los permisos deben ser descubribles.
- Quieres probar la invocación de la tool por separado del razonamiento del model.

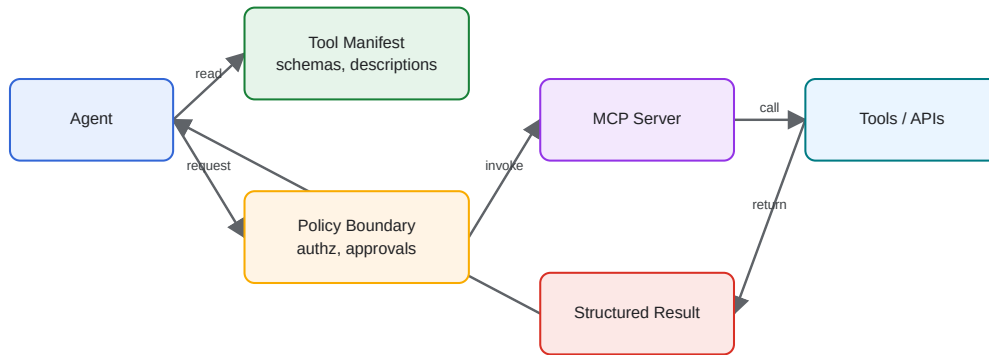
## Evita Cuando

- Una llamada a función local es suficiente y no se necesita un límite de descubrimiento.
- Las entradas de la tool no pueden ser validadas.
- El agent tiene permiso para invocar demasiadas tools sin policy checks.

## Arquitectura

Usa este diagrama para leer MCP-first Tool Use como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el protocolo o el límite de capability posee schemas, permisos, registros de invocación y validación de respuestas.

## MCP-first Tool Use



## Forma del Sistema

- **Límite del pattern:** el agent descubre o selecciona una capability, envía una solicitud tipada y recibe un resultado tipado a través de un policy boundary.
- **Propietario del state:** el protocolo o el límite de capability posee schemas, permisos, registros de invocación y validación de respuestas.
- **Artifact principal:** `modern-tool-use-pattern/` contiene la implementación de referencia ejecutable y ejemplos.
- **Promesa operativa:** MCP-first tool use separa la capability de la tool de la lógica del agent mediante manifests, validación, invocación y resultados estructurados.
- **Ruta ejecutable:** comienza con `npm run mcp:search` antes de adaptar el pattern a un sistema más grande.

## Protocolo Central

1. Descubre la capability, schema, permisos y restricciones operativas.
2. Prepara una solicitud tipada a partir del goal y state actuales.
3. Autoriza la solicitud antes de la invocación.
4. Invoca la tool, skill o remote agent y valida el resultado.
5. Devuelve structured output, rechazo, progreso o error sin perder los correlation IDs.

## Notas de Implementación

- Valida la entrada de la tool antes de cada invocación.
- Mantén los resultados de la tool estructurados y observables.
- Coloca policy checks entre la intención del model y la ejecución de la tool.
- Prefiere tools pequeñas con contratos claros sobre tools amplias con descripciones vagas.

## Modos de Falla

- Manifests de tools que describen capabilities de forma demasiado vaga.
- Entradas generadas por el model para la tool usadas sin validación de schema.



- Efectos secundarios ocultos que no son visibles en el contrato de la tool.
- Sin trace que vincule la decisión del model, la entrada de la tool y el resultado de la tool.

## Estrategia de Evaluación

- Prueba llamadas válidas, argumentos inválidos, llamadas no autorizadas, timeouts, rechazos y respuestas mal formadas.
- Asegúrate de que las acciones peligrosas requieran aprobación o sean bloqueadas antes de la ejecución.
- Mide la precisión en la selección de tools, validez de schemas, fallas de autorización y comportamiento de recuperación.
- Incluye casos que prueben que cada condición de “Úsalo Cuando” es verdadera para este pattern.
- Incluye casos negativos de “Evita Cuando” para que el sistema elija un pattern más simple o seguro cuando corresponda.

## Lista de Verificación para Producción

- Usa schemas tipados para entradas y salidas.
- Separa la intención del model de los permisos de ejecución reales.
- Agrega timeouts, reintentos, claves de idempotencia y registros de auditoría.
- Trata el rechazo y la cancelación como resultados de primera clase.
- Define escalamiento humano para trabajos ambiguos, de alto riesgo o bloqueados por policy.
- Mantén el source bundle, el capítulo generado, las pruebas y el artifact de despliegue en la misma release.

## Ejecuta el Ejemplo

```
npm run mcp:search
npm run mcp:cloud
npm run mcp:agent
npm run mcp:test
```

## Recorrido por el Código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

## Código Fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el download bundle y el 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**

[Abrir el código fuente completo](#)

```
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'));
```

## Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `modern-tool-use-pattern/` actual de este repositorio.

## Patrones relacionados

- [Skills](#)
- [A2A Agent Interoperability](#)
- [Secure Agent Communication](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# A2A Agent Interoperability

A2A hace que los agents sean descubribles e invocables a través de límites de proceso, equipo, runtime y proveedor.

## Fuente y descargas

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

## Propósito

A2A hace que los agents sean descubribles e invocables a través de límites de proceso, equipo, runtime y proveedor. Úsalo cuando un agent necesite invocar a otro agent como colaborador remoto mediante un contrato de protocolo explícito.

Esta es la versión agent de la comunicación entre servicios. El objetivo no es permitir que los agents conversen libremente. El objetivo es permitir que un agent acotado solicite trabajo a otro agent acotado con identidad, validación de schema, autorización, idempotencia, progreso, rechazo, cancelación, trazabilidad y versionado.

Los patrones de comunicación de microservicios siguen aplicando. REST, gRPC, MCP y A2A son diferentes formas de protocolo, pero las preocupaciones de ingeniería son familiares: contratos, identidad, autorización, reintentos, timeouts, idempotencia, observabilidad, propiedad y compatibilidad hacia atrás.

## Úsalo cuando

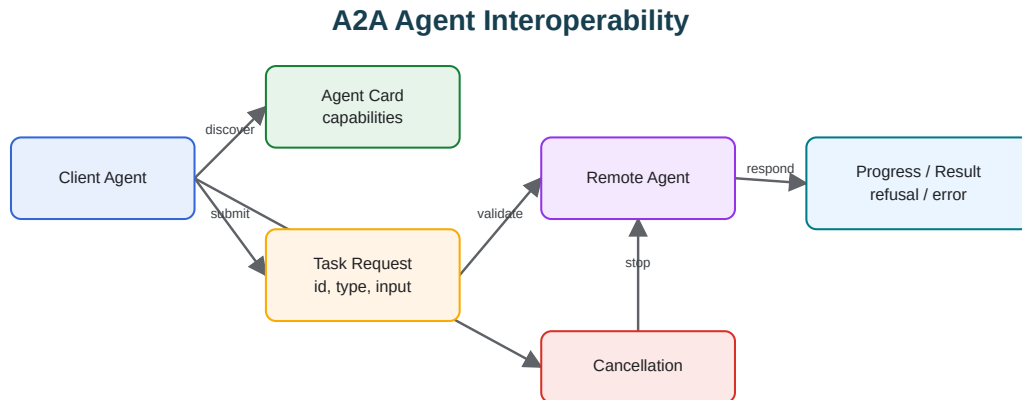
- Los agents son propiedad de diferentes servicios, equipos, runtimes o proveedores.
- Un llamador debe descubrir qué puede hacer un agent remoto antes de enviar trabajo.
- El state de la task debe sobrevivir progreso asíncrono, rechazo, error, timeout, reintento o cancelación.
- El llamador y el receptor necesitan un contrato compartido para propiedad, ciclo de vida de la task y forma del resultado.
- La comunicación entre agents requiere TLS, OAuth u OIDC scopes, trace IDs y registros de auditoría.

## Evítalo cuando

- Ambos agents son funciones simples dentro de un solo proceso.
- La interacción es solo una llamada local de tool con entrada y salida tipadas.
- No puedes autenticar llamadores, validar mensajes o trazar handoffs.
- El agent remoto no tiene un owner estable, contrato de capability o policy de versionado.
- Un workflow determinista sería más claro que agregar otro colaborador autónomo.

## Arquitectura

Usa este diagrama para leer A2A Agent Interoperability como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el límite de protocolo o capability posee schemas, permisos, registros de invocación y validación de respuestas.



## Forma del sistema

- **Límite de capability:** los llamadores descubren capabilities desde una agent card en vez de depender de suposiciones obsoletas.
- **Límite de mensaje:** cada solicitud de task tiene un schema, trace ID, message ID, idempotency key, source agent, target agent, tenant, audience, scopes y timeout.
- **Límite de identidad:** los llamadores se autentican mediante identidad de servicio y credenciales delegadas.
- **Límite de autorización:** scopes, tenant, audience, capability, policy y riesgo se verifican antes de que el agent remoto inicie el trabajo.
- **Límite de propiedad:** cada task tiene un owner actual, status, stop reason y ruta de cancelación.
- **Límite de observabilidad:** handoffs, reintentos, rechazos, timeouts, aprobaciones y resultados son eventos de trace, no mensajes de chat ocultos.

## Protocolo central

1. Obtener la agent card remota: capability, schema, versión, scopes, owner, timeout y estados de ciclo de vida.
2. Construir un sobre de solicitud de task desde el goal actual, state, tenant, trace ID, idempotency key y presupuesto de delegación permitido.
3. Validar el schema del mensaje antes del transporte.
4. Autenticar al llamador y verificar audience, scopes, tenant, capability y policy.
5. Entregar la solicitud al agent remoto solo después de que la autorización sea exitosa.
6. Emitir eventos de progreso, rechazo, error, timeout, cancelación o resultado con el mismo trace ID.
7. Validar el schema de respuesta y el state de propiedad antes de que el llamador consuma el resultado.
8. Registrar el handoff, decisión, latencia, costo, status, owner y stop reason.
9. Convertir fallas repetidas, delegación insegura o incompatibilidades de protocolo en casos de eval.

## Notas de implementación

- Los mensajes deben validarse contra schemas antes de la entrega.
- Los mensajes A2A deben llevar campos de correlación, no depender de logs para reconstruir un handoff después.
- Usa TLS para transporte remoto y mTLS donde la identidad de servicio sea importante.
- Usa OAuth u OIDC scopes para autoridad delegada.
- Trata rechazos, timeouts y cancelación como resultados normales del protocolo, no como excepciones.
- Incluye idempotency keys o task IDs para que los reintentos no dupliquen trabajo.
- Agrega autorización antes de cruzar un límite de confianza.
- Mantén versiones de agent card y versiones de schema de mensaje compatibles hacia atrás.
- Evita bucles de delegación rastreando owner, task padre, profundidad de delegación y stop reason.

## Sobre el mensaje

```
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>;
};
```

## Verificación de autorización

```
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' };
}

```

El sobre exacto puede cambiar según el protocolo, pero el límite debe permanecer: identificar al llamador, identificar al destino, validar autoridad, preservar correlación y evitar trabajo duplicado.

## Modos de falla

- Tratar un agent remoto como una función local e ignorar latencia, rechazo, timeout o cancelación.
- Enviar blobs de lenguaje natural no validados en vez de mensajes de task tipados.
- Omitir el descubrimiento de capability, haciendo que los llamadores dependan de suposiciones obsoletas.
- No tener trace ID en mensajes de progreso, resultado y error.
- No tener idempotency key, por lo que los reintentos duplican trabajo.
- Se acepta audience incorrecto o falta de scope porque el agent remoto confía en el llamador por nombre.
- Los agents delegan la misma task de ida y vuelta sin owner ni presupuesto de delegación.
- El agent remoto cambia su schema sin versionado.
- Un camino de fallback omite autorización u observabilidad.

## Estrategia de evaluación

Prueba el protocolo, no solo la colaboración en el camino feliz.

- Prueba llamadas válidas con audience, scope, tenant, schema, trace ID e idempotency key correctos.
- Prueba audience incorrecto, falta de scope, incompatibilidad de schema, falta de trace ID y mensajes duplicados.
- Prueba el rechazo como resultado válido.

- Prueba comportamiento de timeout y cancelación.
- Prueba detección de bucles de delegación.
- Prueba solicitudes de capability inseguras.
- Prueba evolución de schema compatible hacia atrás.
- Prueba que cada handoff emita eventos de trace y propiedad.

Mide tasa de validez de schema, falsos positivos de autorización, manejo de rechazos, recuperación de timeout, detección de mensajes duplicados, tasa de bucle de delegación, completitud de trace y cobertura de owner en fallas.

## Lista de verificación para producción

- Publica una agent card con capabilities, schemas, scopes, owner, versión y estados de ciclo de vida.
- Valida cada solicitud y respuesta contra schemas versionados.
- Requiere TLS para transporte remoto y mTLS para identidad de servicio donde aplique.
- Valida OAuth u OIDC audience, scopes, subject, tenant y expiración antes de la ejecución.
- Requiere trace ID, message ID, task ID, idempotency key, source agent y target agent.
- Trata rechazo, cancelación, timeout y espera de aprobación como resultados de primera clase.
- Rastrea owner actual y profundidad de delegación para evitar rebote de tasks.
- Agrega timeouts, reintentos y semántica de cancelación antes de producción.
- Registra handoffs, decisiones de autorización, progreso, resultados y stop reasons en traces.
- Convierte fallas de protocolo y delegación insegura en fixtures de eval.

## Ejecuta el ejemplo

```
npm run a2a:test
npm run a2a:run
```

## Recorrido del código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

## Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el repository source.

[agent-to-agent-communication-pattern/src/run\\_demo.ts](#)

[Abrir fuente completa](#)

```
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**

[Abrir fuente completa](#)

```

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

[Abrir fuente completa](#)

```

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' };
}
}
```

## Descargar

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `agent-to-agent-communication-pattern/` actual de este repositorio.

## Patrones relacionados

- [Agents As Services](#)
- [Choosing Multi-Agent Topology](#)
- [Supervisor / Worker](#)
- [Secure Agent Communication](#)
- [Policy Enforcement](#)
- [Observability and Evals](#)
- [MCP-first Tool Use](#)

# Secure Agent Communication

La comunicación segura protege los mensajes entre agents con autenticación, integridad, confidencialidad y verificaciones de policy.

## Fuente y descargas

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

## Intento

La comunicación segura entre agents protege los mensajes entre agents, tools, runtimes y servicios con identidad, autorización, confidencialidad, integridad, protección contra replay y observability.

Este pattern es importante porque la comunicación agent-to-agent no es solo chat entre models. Los mensajes pueden llevar datos privados, instrucciones para tools, goals delegados, solicitudes de aprobación, comandos de workflow y propuestas de efectos secundarios. Si la capa de comunicación es débil, el model puede ser correcto y aun así el sistema puede ser inseguro.

La regla práctica es simple: cada mensaje de agent que cruza un límite necesita un transport boundary, un identity boundary, un authorization boundary y un audit boundary.

## Usa cuando

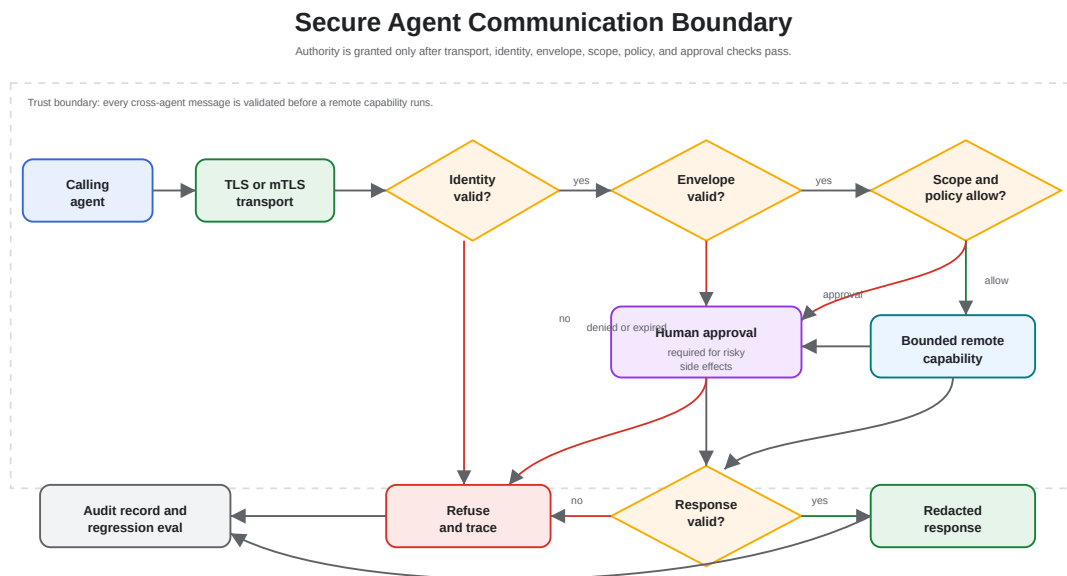
- Agents, tools, skills o runtimes se comunican entre procesos, servicios, equipos, tenants o redes.
- Los mensajes contienen datos privados, business state, credenciales, tasks delegadas o instrucciones de efectos secundarios.
- Agents remotos pueden invocar tools, workflows, APIs, browsers, shells o data stores.
- El sistema necesita TLS, mTLS, OAuth u OIDC, credenciales con alcance, protección contra replay y correlación de traces.
- Los operadores deben poder explicar quién solicitó qué, qué policy lo permitió, qué se ejecutó y qué trace lo prueba.

## Evita cuando

- Toda la comunicación es local, en el mismo proceso y ya está cubierta por el mismo trusted execution boundary.
- El sistema no puede identificar caller, tenant, audience, capability y trace ID.
- La autorización ocurre solo en el prompt o solo después de que la acción remota se ejecuta.
- Los mensajes no pueden ser redactados, retenidos, trazados o reproducidos de forma segura.
- El ciclo de vida de tokens y certificados no puede ser operado por el equipo.

## Arquitectura

Usa este diagrama para leer Secure Agent Communication como un system boundary, no solo una forma de código. La pregunta clave de propiedad es: el protocolo o capability boundary es dueño de los schemas, permisos, registros de invocación y validación de respuestas.



Léelo como una secuencia de puertas de autoridad. El agent o tool remoto recibe capability solo después de que pasan las verificaciones de transport, identidad, envelope, scope, policy y aprobación.

## Forma del sistema

- **Transport boundary:** usa TLS para la comunicación entre servicios y mTLS cuando ambos lados necesitan identidad de servicio respaldada por certificado.
- **Identity boundary:** identifica caller, subject, tenant, audience, issuer e identidad de servicio antes de procesar la solicitud.
- **Authorization boundary:** verifica claims de token OAuth u OIDC, scopes, audience, tenant, capability de tool, clase de riesgo y requisitos de aprobación.
- **Message boundary:** valida schemas, claves de idempotency, timestamps, nonces y tamaño de contenido antes de que el mensaje entre al agent loop.
- **Data boundary:** cifra los payloads sensibles cuando sea necesario, redacta traces y evita colocar secretos en prompts, memory, logs o fixtures de eval.
- **Observability boundary:** emite eventos de trace para verificación de identidad, decisión de autorización, resultado de policy, invocación de tool, rechazo, aprobación y validación de respuesta.

## Protocolo central

1. Recibe un mensaje por TLS o mTLS.
2. Verifica identidad del servicio y emisor del token, audience, expiración, subject, tenant y scopes.
3. Valida el schema del mensaje, clave de idempotency, timestamp, nonce, trace ID y correlation IDs.
4. Construye el policy context a partir de datos confiables del runtime, no solo del texto del mensaje.
5. Verifica capability de tool, data scope, clase de riesgo, estado de budget y requisito de aprobación.
6. Descifra o desempaqueta payloads sensibles solo después de que la autorización sea exitosa.



7. Ejecuta la acción remota limitada, o niega, escala o espera aprobación.
8. Valida el schema de la respuesta y redacta campos sensibles antes de devolverla.
9. Registra eventos de trace, decisión de policy, subject del token, identidad de servicio, scopes, latencia, costo y motivo de detención.

## Notas de implementación

TLS protege el transporte. OAuth y OIDC identifican y autorizan al caller. Policy decide si la capability solicitada está permitida. Observability prueba lo que sucedió.

No trates estos controles como intercambiables:

| Control           | Qué responde                                                                       |
|-------------------|------------------------------------------------------------------------------------|
| TLS               | ¿La conexión está cifrada y protegida en tránsito?                                 |
| mTLS              | ¿Qué identidad de servicio está al otro lado de esta conexión?                     |
| OAuth u OIDC      | ¿Quién o qué es el caller y qué scopes o claims tiene?                             |
| Policy engine     | ¿Este caller puede realizar esta acción sobre este recurso ahora?                  |
| Message signing   | ¿Este mensaje ha sido alterado o reproducido fuera de la sesión de transporte?     |
| Trace correlation | ¿Los operadores pueden reconstruir el camino de la solicitud entre agents y tools? |

Usa credenciales con alcance y de corta duración. Un agent remoto debe recibir el scope mínimo requerido para el task delegado, no toda la autoridad del usuario ni un token de servicio de toda la plataforma.

## Observability

La comunicación segura sin observability es difícil de operar. Registra lo suficiente para auditar el comportamiento sin filtrar secretos.

Traza estos eventos:

- mensaje entrante aceptado o rechazado;
- identidad de servicio TLS o mTLS;
- emisor del token, audience, subject, tenant y scopes después de la redacción;
- decisión de policy y motivo;
- clave de idempotency y resultado de detección de replay;
- solicitud de aprobación, concesión de aprobación o denegación de aprobación;
- agent o tool remoto invocado;
- resultado de validación de respuesta;
- latencia, costo, timeout, reintento y motivo de detención.

No registres tokens de acceso sin procesar, refresh tokens, claves privadas, secretos descifrados ni datos privados sin redactar. Observability debe explicar autoridad y comportamiento, no convertirse en una segunda fuga de datos.

## 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' };

}
```

La implementación real debe validar tokens y certificados firmados con librerías confiables y proveedores de identidad de la plataforma. El punto arquitectónico importante es dónde vive la verificación: antes de que el agent o tool remoto reciba autoridad.

## Modos de falla

- TLS está presente, pero cualquier servicio autenticado puede llamar cualquier agent capability.
- Los scopes de OAuth son amplios, de larga duración o no se verifican contra la capability específica.
- El audience del token se ignora, así que un token creado para un servicio funciona contra otro.
- Tenant, subject o identidad de servicio se toman del texto del model en vez de claims confiables del runtime.
- Los mensajes no tienen clave de idempotency, nonce, timestamp ni detección de replay.
- Agents remotos reciben credenciales ambientales en vez de autoridad delegada con scope.
- Payloads sensibles, tokens o datos descifrados aparecen en prompts, memory, logs o traces.
- Las llamadas denegadas no se trazan, así que los intentos de abuso son invisibles.
- Observability registra solo los outputs finales pero no identidad, scope, policy o decisiones de transporte.

## Estrategia de evaluación

Prueba la seguridad de la comunicación como comportamiento, no solo como configuración.

- Prueba mensajes válidos con issuer, audience, scopes, tenant y schema correctos.
- Prueba audience incorrecto, scope faltante, token expirado, tenant incorrecto, trace ID faltante y clave de idempotency repetida.
- Prueba intentos de invocar tools fuera de la capability delegada.
- Prueba el replay de un mensaje antiguo.
- Prueba la redacción de tokens y datos privados en traces.
- Prueba capabilities que requieren aprobación antes de efectos secundarios.
- Prueba que las llamadas denegadas sean visibles en audit y observability.
- Prueba que los fallback paths no omitan identidad ni autorización.

Mide false allows y false denials de autorización, detección de replay, campos faltantes en trace, cobertura de policy-decision, fallas de redacción y tiempo promedio para investigar un incidente cross-agent.

## Lista de verificación para producción

- Exige TLS para toda comunicación remota entre agents.
- Usa mTLS para identidad service-to-service donde la infraestructura lo soporte.
- Valida issuer, audience, expiry, subject, tenant y scopes de OAuth u OIDC.

- Mantén los tokens con alcance limitado, corta duración, rotados y nunca visibles para el model a menos que sea absolutamente necesario.
- Valida schema del mensaje, clave de idempotency, timestamp, nonce y trace ID.
- Autoriza antes de descifrar payloads sensibles o invocar capabilities remotas.
- Aplica verificaciones de policy a llamadas delegadas de tools, comandos de workflow y respuestas finales.
- Redacta tokens, secrets, datos privados y payloads sensibles de los traces.
- Emite eventos de observability para identidad, autorización, policy, aprobación, invocación y validación de respuestas.
- Convierte incidentes de seguridad y llamadas peligrosas denegadas en regression evals.

## Ejecuta el ejemplo

```
npm run secure-agent
```

## Recorrido del código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo alrededor explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o flujo de control.

## Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el repositorio fuente.

```
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();

```

Extracto truncado para mayor legibilidad. Descarga el bundle o abre el archivo fuente para la implementación completa.

[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))

```

Extracto truncado para mayor legibilidad. Descarga el bundle o abre el archivo fuente para la implementación completa.

## Descarga

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

El bundle de descarga contiene la carpeta actual `secure-agent-communication-pattern/` de este repositorio.

## Patrones relacionados

- [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

Los human approval gates pausan la ejecución antes de acciones sensibles, costosas, destructivas o visibles externamente.

## Fuente y descargas

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

## Intento

Los human approval gates pausan un agentic workflow antes de ejecutar una acción sensible, costosa, destructiva o visible externamente. El gate no es “preguntar a un humano en el chat”. Es una pausa controlada del workflow con la acción propuesta, riesgo, evidencia, policy, identidad del aprobador, tiempo de espera, decisión y registro de auditoría.

La aprobación es un límite de arquitectura. El model puede proponer una acción. El software decide si esa acción necesita aprobación, empaqueta la solicitud de aprobación, espera de forma durable, registra la decisión y reanuda o detiene el workflow.

El detalle importante es que la aprobación por sí sola no hace seguro a un agent. Un paso de aprobación vago puede convertirse en teatro. Un approval gate útil vincula una decisión humana a una acción exacta, bajo una versión específica de la policy, con suficiente evidencia para que el humano tome una decisión real.

## Usar cuando

- El agent puede desencadenar efectos secundarios financieros, legales, de seguridad, de acceso a datos o visibles para el cliente.
- Un humano debe revisar la evidencia antes de que el workflow continúe.
- El workflow puede pausar, persistir el state y reanudar de forma segura.
- Las decisiones de aprobación necesitan una auditoría.
- La policy puede definir qué acciones requieren aprobación y cuáles pueden proceder automáticamente.

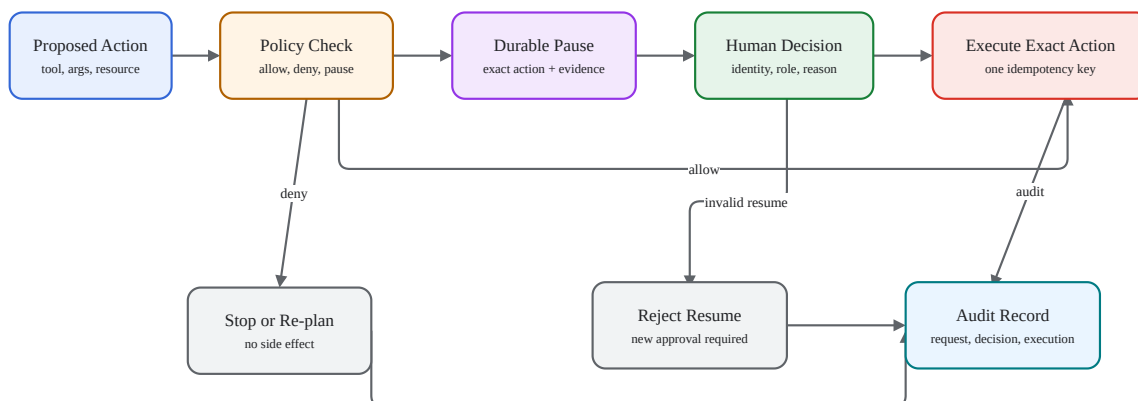
## Evitar cuando

- Cada paso requiere aprobación y el agent ya no reduce trabajo.
- El aprobador no puede ver suficiente evidencia para tomar una decisión.
- La aprobación ocurre en un mensaje de chat no rastreado o canal alternativo.
- El workflow no puede reanudarse de forma segura después de esperar.
- El sistema no puede evitar que los reintentos reutilicen aprobaciones obsoletas.

## Arquitectura

Usa este diagrama para leer Human Approval Gates como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el protocolo o capability boundary posee schemas, permisos, registros de invocación y validación de respuestas.

### Human Approval Gate



## Forma del sistema

- **Pattern boundary:** el approval gate es dueño de la pausa, solicitud, decisión, tiempo de espera, resume token y registro de auditoría.
- **State owner:** el workflow engine o runtime es dueño del state durable mientras espera la aprobación.
- **Model role:** el model explica la acción propuesta y la evidencia de soporte, pero no aprueba su propia acción.
- **Policy boundary:** el software decide si se requiere aprobación antes de ejecutar la acción.
- **Promesa operativa:** las acciones de alto riesgo no se ejecutan solo porque el model las solicitó.

## Core Protocol

1. Recibe una acción propuesta por el model con caller, trace ID, riesgo, evidencia y efectos secundarios.
2. Ejecuta la policy para decidir si la acción está permitida, denegada o requiere aprobación.
3. Si se requiere aprobación, crea una solicitud de aprobación durable.
4. Presenta al aprobador la acción propuesta, evidencia, razón de la policy, riesgo y consecuencias.
5. Espera por aprobar, denegar, solicitar cambios, escalación, tiempo de espera o cancelación.
6. Registra quién decidió, cuándo, por qué y bajo qué versión de la policy.
7. Reanuda solo la acción exacta aprobada, o detiene/revisa si se deniega.
8. Almacena la decisión junto con el trace y el registro de auditoría de la acción.

La decisión de policy debe ocurrir antes del efecto secundario. Si el tool ya envió el correo, emitió el reembolso, cambió el permiso o escribió en la memory durable, el approval gate es solo una nota de incidente.

## Notas de implementación

Las solicitudes de aprobación deben estar tipadas. El aprobador no debe inferir el efecto secundario a partir de un texto vago.

## La aprobación no es una sola cosa

Diferentes riesgos necesitan diferentes formas de aprobación. Tratar cada human checkpoint como el mismo botón crea fricción en flujos de bajo riesgo y control débil en flujos de alto riesgo.

| Approval Type  | Purpose                                                         | Example                                                                        |
|----------------|-----------------------------------------------------------------|--------------------------------------------------------------------------------|
| Confirmation   | El usuario confirma la intención antes de una acción visible.   | “Enviar este correo redactado a estos destinatarios.”                          |
| Review         | Una persona calificada revisa la evidencia y la justificación.  | El líder de soporte revisa una recomendación de reembolso.                     |
| Authorization  | Un rol con autoridad explícita permite una acción.              | Finanzas aprueba un reembolso por encima de un umbral.                         |
| Escalation     | El workflow no puede decidir al nivel actual.                   | Seguridad revisa una solicitud de acceso sospechosa.                           |
| Break-glass    | Una anulación de emergencia poco común con auditoría reforzada. | Restaurar acceso durante una caída.                                            |
| Batch approval | Una decisión cubre un conjunto limitado de acciones similares.  | Aprobar 25 respuestas de tickets de bajo riesgo generadas por la misma policy. |

Batch approval es la que requiere más cuidado. Debe definir la membresía del lote, el conteo máximo, el tipo de acción permitida, la clase de riesgo, expiración y reglas de muestreo o revisión. No debe convertirse en “aprobar lo que sea que el agent haga después”.

## Approval Envelope

Una solicitud de aprobación debe contener el action envelope, no solo un mensaje legible por humanos. La parte legible puede explicar la decisión. La parte legible por máquina es la que previene el approval laundering.

| Field                 | Why It Matters                                                               |
|-----------------------|------------------------------------------------------------------------------|
| Approval ID           | Identificador estable para la pausa y decisión.                              |
| Action ID             | La acción exacta que se aprueba.                                             |
| Tool name and version | Previene que la aprobación se desplace entre cambios de tool.                |
| Arguments hash        | Detecta cambios ocultos después de la aprobación.                            |
| Resource IDs          | Muestra qué cliente, cuenta, archivo, ticket, pago o permiso se ve afectado. |
| Risk class            | Conecta la aprobación con la policy y el enrutamiento.                       |
| Evidence references   | Permite al aprobador inspeccionar el material fuente.                        |
| Policy version        | Explica por qué se requirió la aprobación.                                   |

| Field           | Why It Matters                                         |
|-----------------|--------------------------------------------------------|
| Approver role   | Previene que la persona equivocada apruebe.            |
| Expiration      | Previene que se reutilicen aprobaciones antiguas.      |
| Idempotency key | Previene ejecuciones duplicadas tras un reintento.     |
| Trace ID        | Conecta la aprobación con la ejecución y la auditoría. |

El action ID debe derivarse de campos estables: tool, versión, argumentos, recurso, tenant, actor, policy version e idempotency key. Si alguno de estos cambia, la aprobación ya no debe coincidir.

```
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;
};
```

El registro de decisión es tan importante como la solicitud:

```
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;
};
```

Nunca trates la aprobación como un permiso general. Vincúlala a la acción exacta:

```
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;
}
```

La aprobación autoriza una acción, bajo una versión de policy, con un trace. Si el agent cambia la acción, el workflow necesita una nueva aprobación.

## Qué debe ver el aprobador

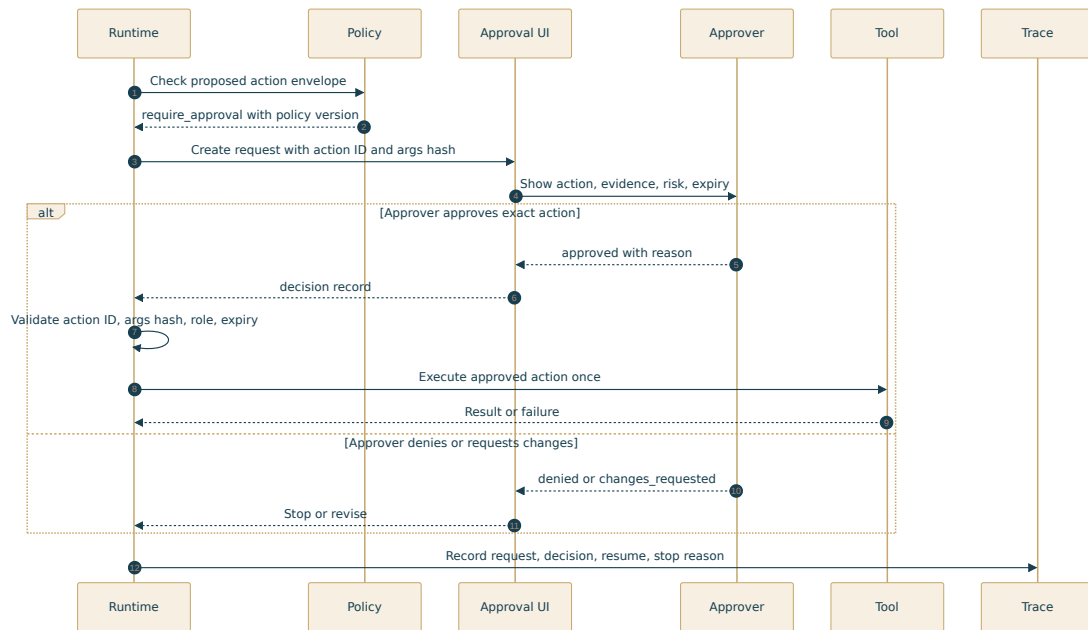
Una buena interfaz de aprobación no es una transcripción de chat con un botón de aprobar. Debe mostrar la acción exacta, el recurso que se va a cambiar, la clase de riesgo, la razón de la policy, la evidencia, el diff o payload, el blast radius, la expiración y las decisiones disponibles.

Para un reembolso a un cliente, el aprobador debe ver la orden, el pago, el mensaje del cliente, la refund policy, el monto, la razón y si el agent está redactando, enviando para aprobación o emitiendo el dinero. Para un correo saliente, el aprobador debe ver los destinatarios, asunto, cuerpo, adjuntos, evidencia fuente y cualquier dato privado incluido. Para una escritura en memory, el aprobador debe ver la memory propuesta, fuente, tenant, clase de retención y ruta de eliminación.

La persona debe poder aprobar, denegar, solicitar cambios, escalar o cancelar. Los comentarios libres son útiles, pero la decisión en sí debe estar estructurada.

## Flujo de revisión de aprobación

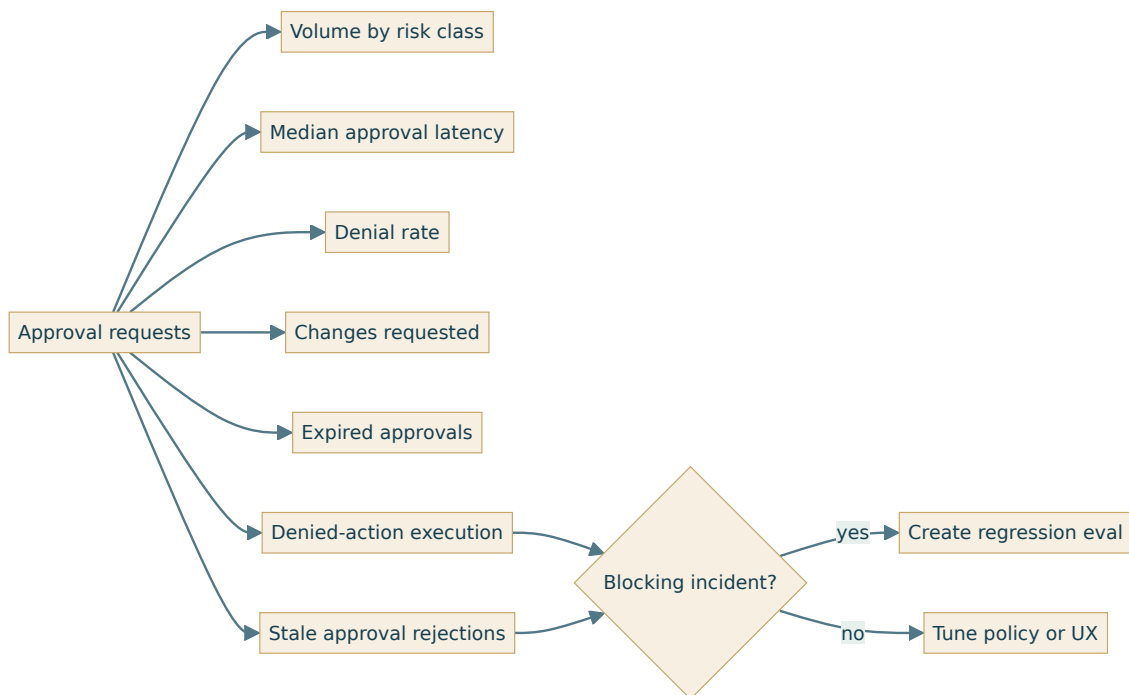
Usa este flujo para probar si la UI, el policy engine y el runtime están de acuerdo sobre la misma acción. El aprobador nunca debe aprobar un resumen que el runtime luego traduzca en un payload diferente.



La UI puede ser simple, pero debe preservar el contrato de máquina. La acción visible, la razón de la policy, el hash de argumentos, los IDs de recursos, la expiración y la decisión deben coincidir con los registros usados por `canResumeWithApproval`.

## Panel de métricas de aprobación

Los approval gates necesitan señales operativas, no solo una pantalla de solicitud. Un pequeño dashboard debe separar problemas de conveniencia de problemas de seguridad.



Trata la ejecución de acciones denegadas, la reutilización de aprobaciones obsoletas y la ausencia de spans de aprobación como señales que bloquean el release. Una alta latencia de aprobación puede ser un problema del workflow. Una alta tasa de denegación puede indicar que el agent está solicitando acciones fuera de su autoridad. Una alta tasa de rubber-stamp puede indicar que la UI está pidiendo demasiadas decisiones de bajo valor.

### Protección contra aprobación obsoleta y replay

Los approval gates fallan cuando una decisión puede ser reproducida fuera de su context original. El runtime debe rechazar una aprobación cuando la acción cambió, la versión de la policy cambió, la aprobación expiró, el rol del aprobador ya no coincide, el tenant cambió, el recurso cambió o la idempotency key ya fue consumida.

Esto es importante para los reintentos. Los durable workflows deben reanudar desde la espera de aprobación, no reconstruir una acción similar desde una nueva salida del model y asumir que la aprobación anterior aún aplica. Si el model vuelve a planear después de la aprobación, el nuevo plan debe pasar la policy nuevamente.

### Seguimiento de denegaciones

Las denegaciones repetidas son señal, no ruido. Haz seguimiento de las solicitudes de tool denegadas por tool, alcance y razón. Después de un pequeño umbral, el agent debe dejar de reintentar y elegir uno de tres caminos:

1. pedir una aprobación más amplia con una razón clara;
2. cambiar a un fallback de menor riesgo;
3. detenerse y reportar el requerimiento bloqueado.

Esto previene loops de aprobación donde el agent gasta su presupuesto pidiendo la misma acción bloqueada en formas ligeramente diferentes.



## Observabilidad

Las aprobaciones deben aparecer como spans de primer nivel en los traces, no como notas adjuntas a una respuesta final. Un trace debe mostrar la decisión de policy que requirió aprobación, la solicitud de aprobación, la duración de la espera, el aprobador, la decisión, el resume token y el efecto final o la razón de detención.

Métricas útiles en producción incluyen volumen de aprobaciones por clase de riesgo, latencia de aprobación, tasa de denegación, tasa de cambios solicitados, tasa de aprobaciones expiradas, tasa de rechazo por aprobación obsoleta, tasa de rubber-stamp, tasa de override e incidentes de aprobación perdida. Estas métricas no son solo operativas. Son retroalimentación sobre si el nivel de autonomía está correctamente configurado.

## Modos de falla

- La solicitud de aprobación carece de evidencia, por lo que la persona aprueba sin revisar o adivina.
- El texto de aprobación oculta el efecto secundario real.
- El sistema solicita aprobación después de que el tool ya se ejecutó.
- Un reintento reutiliza la aprobación para una acción diferente.
- La aprobación expira pero el workflow continúa de todos modos.
- No se registra la identidad del aprobador ni la razón.
- La fatiga de aprobación hace que las personas aprueben todo.
- Las aprobaciones denegadas se convierten en prompts más suaves y se reintentan hasta que pasan.
- El audit trail registra la respuesta final pero no la acción propuesta ni la decisión.
- La aprobación se adjunta a una conversación, no a una llamada exacta de tool.
- Una aprobación por lote cubre acciones que no eran visibles cuando se aprobó el lote.
- El model vuelve a planear después de la aprobación y ejecuta un payload diferente.
- El aprobador ve una justificación pulida pero no la evidencia fuente.

## Estrategia de evaluación

Los approval evals deben demostrar que las acciones riesgosas se pausan y que las acciones seguras no generan fricción innecesaria.

- Prueba acciones de alto riesgo que deben requerir aprobación.
- Prueba acciones de solo lectura de bajo riesgo que no deberían requerir aprobación.
- Prueba una aprobación denegada y verifica que el efecto secundario no se ejecute.
- Prueba una aprobación expirada y verifica que el workflow no se reanude.
- Prueba una acción cambiada después de la aprobación y requiere una nueva aprobación.
- Prueba falta de evidencia y requiere `changes_requested` o escalamiento.
- Prueba el comportamiento de reintento con idempotency keys.
- Prueba cambios de policy-version entre la solicitud y la reanudación.
- Prueba aprobaciones por lote con un ítem fuera de policy.
- Prueba prompt injection en la evidencia que pide al aprobador ignorar la policy.
- Prueba la completitud del audit: solicitud, decisión, aprobador, policy version y trace ID.

Un fixture de eval compacto puede hacer explícito el límite de aprobación:

```
{
 "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"]
 }
}
```

Mide la precisión del enrutamiento de aprobaciones, tasa de aprobaciones innecesarias, tasa de ejecución de acciones denegadas, reutilización de aprobaciones obsoletas, latencia de aprobación, completitud de audit, tasa de rubber-stamp y tasa de override humana.

Trata la ejecución de acciones denegadas y la reutilización de aprobaciones obsoletas como fallas bloqueantes, no como métricas de calidad promedio. Segmenta la latencia de aprobación y la tasa de override por clase de riesgo, tipo de acción, policy version y rol del aprobador para que un flujo saludable de bajo riesgo no oculte un control débil en alto riesgo.

Para el contrato de caso de eval compartido y el método de release-gate, consulta [Evaluation-Driven Agent Development](#).

## Lista de verificación para producción

- Define qué acciones requieren aprobación por policy, no por redacción del prompt.
- Incluye acción propuesta, efectos secundarios, evidencia, razón de policy y riesgo en cada solicitud.
- Mantén el action envelope legible por máquina.
- Persiste el workflow state mientras esperas.
- Vincula la aprobación a un action ID exacto y a una idempotency key.
- Vincula la aprobación a la versión del tool, hash de argumentos, tenant, recurso y policy version.
- Configura expiración, timeout, cancelación y comportamiento de escalamiento.
- Registra identidad del aprobador, decisión, razón, policy version y trace ID.
- Evita que aprobaciones denegadas o expiradas se reintenten silenciosamente.
- Rechaza aprobaciones cuando la acción cambia después de la revisión.
- Trata las aprobaciones como spans de trace y registros de audit.
- Haz seguimiento del volumen de aprobaciones y la fatiga de aprobación.
- Mantén versionadas las approval policies, request schemas y decision records.
- Convierte errores graves de aprobación en regression evals.

## Ejecuta el ejemplo

```
npm run approval-gate
npm run approval-gate:test
```

## Recorrido por el Código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

## Código Fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el paquete de descarga y en el repositorio fuente.

[human-in-the-loop-approval-agent/typescript/src/approval\\_gate.ts](https://github.com/human-in-the-loop/approval-agent/typescript/src/approval_gate.ts)

[Abrir fuente completa](#)

```
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](#)

[Abrir fuente completa](#)

```
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,
 };
 }
}

```

Extracto truncado para mayor legibilidad. Descarga el paquete o abre el archivo fuente para ver la implementación completa.

[human-in-the-loop-approval-agent/typescript/test/approval\\_gate.spec.ts](#)

[Abrir fuente completa](#)

```

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",
);
```

*Extracto truncado para mayor legibilidad. Descarga el paquete o abre el archivo fuente para ver la implementación completa.*

## Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `human-in-the-loop-approval-agent/` actual de este repositorio.

## Patrones Relacionados

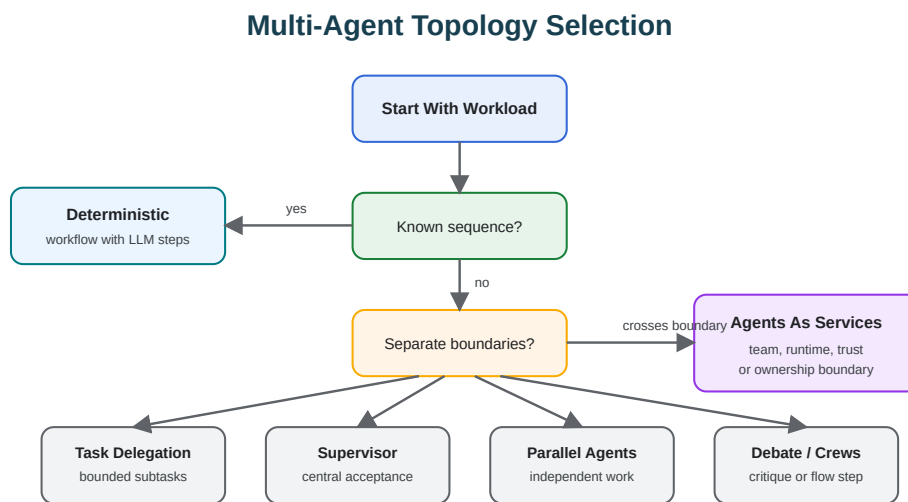
- [Tool Capability Design](#)
- [MCP-first Tool Use](#)
- [Policy Enforcement](#)
- [Durable Workflows](#)
- [Observability and Evals](#)
- [Cost Controls and Runtime Budgets](#)
- [Agent Threat Model](#)
- [Agent UX and Human Trust](#)
- [Evaluation-Driven Agent Development](#)
- [Pattern Evaluation Checklist](#)

# Choosing Multi-Agent Topology

No comiences un diseño preguntando cuántos agents necesitas.

Comienza preguntando dónde el trabajo requiere context separado, tools separadas, autoridad separada, tiempos separados o accountability separada. Si esos límites son reales, múltiples agents pueden ayudar. Si esos límites no son reales, múltiples agents usualmente agregan costo, latencia y modos de falla.

El diseño multi-agent no se trata de hacer que el sistema parezca más inteligente. Se trata de crear límites útiles.



La opción predeterminada debe seguir siendo conservadora. El diseño multi-agent es una decisión de coordinación, no una mejora de inteligencia. Cada agent adicional agrega superficie de protocolo, selección de context, permisos de tools, traces, latencia, costo, modos de falla y preguntas sobre ownership. Agrega un agent solo cuando el límite que crea vale ese costo.

## Primera regla

Usa la topología más pequeña que te dé el límite que necesitas.

Muchos sistemas no necesitan múltiples agents. Necesitan:

- un workflow determinista;
- un agent loop con mejores tools;
- ruteo entre workflows;
- un paso de revisión;
- mejor gestión de context;
- mejores evals;
- mejor observability.

Múltiples agents se justifican cuando un solo agent tendría que manejar demasiado context, demasiada autoridad, demasiadas tools o demasiadas responsabilidades en conflicto.

## Cuándo dividir agents

Divide agents por razones de ingeniería, no porque el problema suene complejo.

Buenas razones para dividir:

- **Ownership:** diferentes equipos son dueños de diferentes capabilities, policies, releases o incidents.
- **Context:** cada parte necesita un conjunto de trabajo diferente y no debería ver todo.
- **Authority:** una parte puede leer datos, otra puede redactar acciones y otra puede aprobar efectos secundarios.
- **Tools:** los sets de tools son lo suficientemente diferentes como para que compartirlos aumente el riesgo.
- **Timing:** parte del trabajo es interactivo, parte es asíncrono y parte espera aprobación.
- **Scaling:** una capability necesita diferente despliegue, latencia, costo o ruteo de model.
- **Evaluation:** cada capability puede ser probada con sus propios fixtures y criterios de aceptación.

Malas razones para dividir:

- el prompt de un solo agent es desordenado;
- el equipo espera que más agents compensen metas poco claras;
- cada agent recibirá el mismo context y tools;
- nadie puede definir el contrato de handoff;
- la merge policy es “el model lo resolverá”;
- no hay trace a través de los handoffs.

Si la división no crea un límite más claro, probablemente está ocultando una arquitectura débil.

## Matriz de decisión de topología

| Situación                                                     | Preferir              | Evitar                                                  |
|---------------------------------------------------------------|-----------------------|---------------------------------------------------------|
| Secuencia conocida con reglas claras                          | Workflow determinista | Multi-agent chat.                                       |
| Una task necesita descomposición especializada                | Task delegation       | Un agent general con un prompt gigante.                 |
| Un owner central debe controlar calidad y fusionar resultados | Supervisor-worker     | Agents pares sin owner final.                           |
| Trabajo independiente puede ejecutarse en paralelo            | Parallel agents       | Cadenas de agents secuenciales que desperdician tiempo. |
| El output mejora con crítica o comparación                    | Debate y consenso     | Votación sin evidencia o pruebas.                       |

| Situación                                                    | Preferir                                                  | Evitar                                              |
|--------------------------------------------------------------|-----------------------------------------------------------|-----------------------------------------------------|
| El state del workflow importa más que la charla entre agents | Flow con crews                                            | Crew como sustituto de diseño de state.             |
| Agents cruzan límites de equipo, runtime o confianza         | Agents como servicios sobre A2A, MCP, eventos o workflows | Handoffs en lenguaje natural no estructurado.       |
| Agents necesitan coordinación asíncrona en el tiempo         | Board, queue, durable workflow o task ledger              | Ventanas de context compartido y copy-paste manual. |

La topología correcta suele ser aburrida. Eso es una ventaja.

| Topología                   | Mejor para                                              | Costo principal                           | Control requerido                                       |
|-----------------------------|---------------------------------------------------------|-------------------------------------------|---------------------------------------------------------|
| Single agent                | Una task acotada con una superficie de tool/policy.     | Especialización limitada.                 | State fuerte en el loop, presupuesto y policy de tools. |
| Workflow con pasos de agent | Proceso conocido con subpasos inciertos.                | Esfuerzo de diseño de workflow.           | State durable y transiciones explícitas.                |
| Supervisor-worker           | Trabajo especializado con un owner final.               | Complejidad de coordinación y merge.      | Asignaciones tipadas y puerta de merge.                 |
| Parallel agents             | Trabajo independiente que se beneficia de concurrencia. | Mayor gasto y volumen de traces.          | Reductor de fan-in y evidencia por worker.              |
| Debate o consenso           | Juicio ambiguo que mejora con crítica.                  | Latencia y falsa confianza.               | Juez con evidencia y owner responsable.                 |
| Agents como servicios       | Límites de equipo, runtime, seguridad o despliegue.     | Contratos de servicio y versionado.       | Contrato API/A2A/MCP, auth, ownership, evals.           |
| A2A interoperability        | Colaboración remota de agents a través de límites.      | Complejidad de protocolo y ciclo de vida. | Envelope de mensajes, scopes, idempotencia, traces.     |

Un selector de topología debe codificar el mismo sesgo:

```
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';
}

```

El punto no es automatizar la arquitectura. El punto es hacer que los criterios de selección sean lo suficientemente visibles como para debatirlos.

## Contrato de topología

Antes de implementar, escribe el contrato de topología. Puede ser ligero, pero debe ser explícito:

```

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[];
 };
};
};

```

El contrato obliga a la pregunta clave: ¿qué aporta la topología que un diseño más simple no?

## Declara la topología antes de codificarla

Un agentic system multi-agent debe tener una topología visible antes de tener integración con framework. Escribe:

- agents y sus responsabilidades;
- rutas de comunicación permitidas;
- puntos de fan-out y fan-in;
- puertas y límites de aprobación;
- state compartido y state privado;
- comportamiento ante fallas y cancelaciones.

Esto puede ser un diagrama, una tabla o un archivo declarativo pequeño. La propiedad importante es la revisabilidad: un lector debe entender el grafo de agents sin tener que hacer ingeniería inversa de callbacks y prompts.

## Workflow determinista primero

Si el código puede controlar la secuencia, deja que el código controle la secuencia.

Usa un workflow determinista cuando:

- los pasos son conocidos;
- las reglas de ramificación son claras;
- la corrección depende de policy o reglas de negocio;
- las transiciones de state deben ser auditables;

- la latencia y el costo importan;
- la aprobación humana debe pausar el proceso.

El model aún puede ser útil dentro del workflow. Puede clasificar, extraer, resumir, redactar, rankear o criticar. Pero el workflow es dueño del camino.

## Task Delegation

Usa task delegation cuando el trabajo se descompone en subtasks acotadas.

Una buena delegación tiene:

- contratos claros de subtask;
- context acotado para cada worker;
- forma de output esperada;
- criterios de aceptación;
- un owner para el merge final;
- evals que prueban que la descomposición ayuda.

Una mala delegación es simplemente pedirle a varios agents que “trabajen en esto” y esperar que la respuesta combinada sea mejor.

## Supervisor-Worker

Usa supervisor-worker cuando un agent central o workflow debe ser dueño del goal, state, ruteo y aceptación final.

Esta topología es útil cuando:

- los workers tienen diferentes tools o context;
- el supervisor puede evaluar los outputs de los workers;
- hay una sola respuesta final responsable;
- algunas fallas de workers no deberían fallar toda la task;
- los workers no deben ver el context completo de los demás.

El supervisor no debe convertirse en un god agent oculto. Debe ser dueño de la coordinación, no de cada detalle de la ejecución.

## Parallel Agents

Usa parallel agents cuando el trabajo es independiente y la merge policy es clara.

Buenos casos:

- buscar en múltiples fuentes;
- generar varios candidate plans;
- revisar el mismo artifact desde diferentes perspectivas;
- ejecutar chequeos separados como seguridad, corrección y estilo;
- comparar outputs de diferentes models o prompts.

El paralelismo no es gratis. Aumenta el costo, el volumen de traces y la complejidad de merge. Úsalo cuando reduzca la latencia, mejore la calidad o cubra aspectos independientes.

## Debate y Consenso

Usa debate o consenso cuando la crítica independiente mejora el juicio.

Es útil para:

- decisiones ambiguas;
- revisiones de diseño;
- análisis de riesgos;
- comparar candidate answers;
- exponer supuestos débiles.

Es débil para:

- preguntas fácticas que requieren retrieval;
- decisiones de tools que requieren policy;
- tasks donde cada agent comparte el mismo punto ciego;
- acciones de alto riesgo donde la votación reemplaza la autoridad.

El consenso no es prueba. Es una señal que aún necesita evidencia, pruebas o un responsable.

## Crews y Flows

Usa crews dentro de flows cuando el workflow necesita state explícito y un grupo de agents especializados realiza un paso delimitado.

El flow debe ser dueño de:

- state;
- orden de eventos;
- reintentos;
- aprobaciones;
- persistencia;
- condiciones de detención.

El crew debe ser dueño de:

- colaboración local;
- roles especialistas;
- output delimitado para un paso del workflow.

Si el chat local del crew se convierte en la state machine, la arquitectura es débil.

## Agents Como Servicios



Cuando los agents cruzan procesos, equipos, runtimes, proveedores o límites de confianza, trátalos como servicios.

Eso significa:

- contrato de capability explícito;
- request y response tipados;
- autenticación;
- autorización;
- timeouts;
- idempotency;
- semántica de rechazo;
- cancelación;
- correlación de traces;
- versionado;
- contract tests.

El protocolo puede ser A2A, MCP, REST, gRPC, event streams o un motor de durable workflow. Lo importante no es la marca del protocolo. Lo importante es que el límite sea explícito y testeable.

Aquí es donde la analogía de microservices es útil. Separar agents es razonable cuando crea ownership, contratos, despliegue, seguridad, escalabilidad y observability más claros. No es razonable cuando solo convierte un agent vago en varios agents vagos.

Usa comunicación A2A o estilo servicio cuando el agent remoto tiene su propio ciclo de vida: capability discovery, autorización, progreso, rechazo, cancelación, versionado y ownership de traces. Usa MCP cuando el límite es principalmente un tool o capability. Usa workflows cuando el state y las aprobaciones importan más que la mensajería directa agent-to-agent.

## Coordinación con Board, Queue o Ledger

Algunos trabajos multi-agent no deben coordinarse por chat directo.

Usa un board, queue, task ledger o durable workflow cuando:

- el trabajo ocurre de forma asíncrona;
- humanos y agents participan;
- las decisiones deben quedar ligadas a los work items;
- los agents corren en sesiones separadas;
- el progreso debe sobrevivir reinicios;
- los handoffs necesitan historial de auditoría.

Esto es especialmente útil para coding agents, operations agents, pipelines de investigación y workflows de revisión de larga duración.

## Shared State

Shared state es donde los sistemas multi-agent suelen fallar.

Evita:

- que cada agent escriba en la misma memory;
- scratchpads compartidos ocultos;
- resúmenes sin versionar;
- agents sobrescribiendo conclusiones de otros;
- state almacenado solo en el historial de conversación.

Prefiere:

- un dueño para cada objeto de state;
- task logs solo de append;
- pasos de merge explícitos;
- workflow state durable;
- artifacts tipados;
- registros de handoff legibles por humanos;
- trace IDs entre agents.

Si el shared state no es claro, la topología no está lista.

## Handoffs y Ownership

Cada handoff debe responder tres preguntas:

1. ¿Quién es dueño del task ahora?
2. ¿Qué exactamente se transfirió?
3. ¿Qué condición devuelve el ownership, escala o detiene la ejecución?

Sin ownership, los sistemas multi-agent se vuelven ambigüedad distribuida. Un worker solo es útil si el caller sabe qué aceptó, rechazó, completó o no pudo decidir el worker. Eso significa que los handoffs necesitan schemas, valores de status, razones de detención y traces.

## Guía de Evaluación

Un sistema multi-agent debe superar una baseline más simple.

Evalúa:

- baseline de single-agent versus topología multi-agent;
- mejora de calidad;
- costo de latencia;
- costo de tokens y tools;
- precisión de merge;
- manejo de fallas de workers;
- manejo de desacuerdos;
- completitud de traces;
- aislamiento de permisos;
- aislamiento de context;
- responsabilidad final.

Mide la coordinación como un costo de primera clase:

| Métrica                        | Por qué importa                                                             |
|--------------------------------|-----------------------------------------------------------------------------|
| Corrección de handoff          | Prueba que el trabajo se enruta al agent correcto con el contrato correcto. |
| Overhead de coordinación       | Muestra si los agents extra gastan más esfuerzo del que ahorran.            |
| Calidad de merge               | Prueba que la respuesta final preserva evidencia y desacuerdo.              |
| Claridad de ownership          | Muestra si las fallas tienen un componente responsable.                     |
| Complejidad de traces          | Hace que las fallas distribuidas sean depurables.                           |
| Aislamiento de tools           | Prueba que los agents no pueden usar tools fuera de su rol.                 |
| Aislamiento de context         | Prueba que los agents no reciben context irrelevante o inseguro.            |
| Costo y latencia por topología | Muestra si la topología vale la pena operativamente.                        |

Incluye evals negativas donde el sistema debería elegir una topología más simple. Un buen selector de topología debe saber cuándo no usar múltiples agents.

## Failure Modes

- El diseño multi-agent se usa para ocultar goals poco claros o state débil.
- Cada agent recibe el mismo context, tools, memory y permisos.
- Los workers pueden crear efectos secundarios sin supervisor, policy o chequeos de aprobación.
- Los handoffs son blobs en lenguaje natural sin schema ni owner.
- Los agents delegan tasks de ida y vuelta sin límite de delegación.
- El debate produce acuerdo pero sin evidencia.
- Parallel agents duplican trabajo porque los límites no son claros.
- Agents como servicios no tienen contrato versionado, auth ni regla de idempotency.
- La respuesta final oculta fallas de workers, desacuerdos o evidencia faltante.
- No hay trace que conecte asignaciones, handoffs, outputs de workers, decisiones de merge y razón de detención.

## Lista de Verificación de Diseño

Antes de agregar otro agent, responde:

- ¿Qué límite crea este nuevo agent?
- ¿Necesita context separado?
- ¿Necesita tools separados?
- ¿Necesita permisos separados?
- ¿Necesita temporización independiente?
- ¿Quién es dueño de la aceptación final?
- ¿Cómo se hace el merge de outputs?
- ¿Qué pasa si este agent falla, rechaza o expira?
- ¿Qué state posee?

- ¿Qué context nunca debe ver?
- ¿Qué protocolo o contrato rige el handoff?
- ¿Qué previene loops de delegación?
- ¿Cuál es la baseline más simple?
- ¿Cómo se conectan los traces entre agents?
- ¿Qué eval prueba que esta topología es mejor que la baseline más simple?

Si la única razón para otro agent es “se siente más agentic”, no lo agregues.

## Capítulos Relacionados

- [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

La delegación de tasks asigna subtasks delimitados a workers especializados y combina sus outputs.

Fuente y descargas

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

## Intento

La delegación de tasks asigna subtasks delimitados a workers especializados y combina sus outputs.

## Usar cuando

- Los workers especializados pueden completar subtasks independientes mejor que un solo agent generalista.
- El manager puede definir los outputs esperados, restricciones y criterios de aceptación.
- Los resultados de los subtasks pueden fusionarse y verificarse.

## Evitar cuando

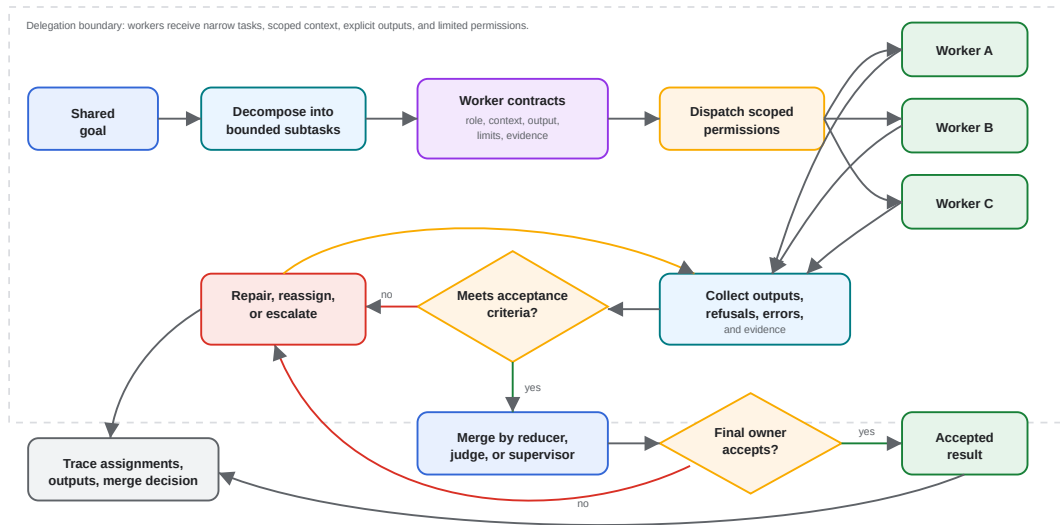
- El task no tiene una descomposición útil.
- Los workers comparten state mutable oculto.
- El manager no puede evaluar el trabajo devuelto.

## Arquitectura

Usa este diagrama para leer Task Delegation como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: el coordinator posee el goal compartido, la descomposición, las asignaciones, la merge policy y la aceptación final.

## Task Delegation Contract Flow

The coordinator owns decomposition, contracts, merge policy, final acceptance, and escalation.



Léelo como un contrato de propiedad: el coordinator posee la descomposición, el alcance de los workers, la merge policy, la aceptación final y la escalación.

## Forma del sistema

- **Límite del pattern:** un coordinator delega trabajo delimitado a agents con roles estrechos, luego evalúa y fusiona sus outputs.
- **Propietario del state:** el coordinator posee el goal compartido, la descomposición, las asignaciones, la merge policy y la aceptación final.
- **Artifact principal:** `task-delegation-pattern/` contiene la implementación de referencia ejecutable y ejemplos.
- **Promesa operacional:** la delegación de tasks asigna subtasks delimitados a workers especializados y combina sus outputs.
- **Ruta ejecutable:** comienza con `npm run task-delegation` antes de adaptar el pattern a un sistema más grande.

## Protocolo central

1. Define el goal compartido, los roles de los workers, los outputs esperados y los criterios de aceptación.
2. Divide el trabajo solo donde la ejecución independiente o especializada agregue valor.
3. Despacha tasks con context y permisos delimitados.
4. Recoge outputs, errores, rechazos y evidencia de cada worker.
5. Fusiona los resultados a través de un judge explícito, reducer, supervisor o una revisión humana.

## Notas de implementación

- Mantén explícito el límite del pattern: los inputs, el state, los efectos secundarios y los outputs deben ser visibles.
- Valida las decisiones producidas por el model antes de que afecten tools, usuarios o state durable.
- Emite suficiente trace data para depurar fallas después de la ejecución.

## Modos de falla

- El pattern se aplica donde un workflow determinista más simple sería mejor.
- El state, las llamadas a tools o las decisiones del model no son lo suficientemente observables para depurar.
- El sistema carece de un comportamiento claro de detención, reintento o escalación.

## Estrategia de evaluación

- Compara el output multi-agent contra una línea base de single-agent en los mismos tasks.
- Prueba desacuerdo entre workers, fallas de workers, trabajo duplicado y malas decisiones de merge.
- Mide mejora de calidad, costo de latencia, costo de tokens, precisión de merge y accountability.
- Incluye casos que demuestren que cada condición de “Usar cuando” es verdadera para este pattern.
- Incluye casos negativos de “Evitar cuando” para que el sistema elija un pattern más simple o seguro cuando corresponda.

## Lista de verificación para producción

- Da a cada worker un contrato y conjunto de permisos estrecho.
- Haz explícita la merge policy antes de que los workers ejecuten.
- Registra los inputs, outputs y evidencia de decisión por worker.
- Mantén un solo propietario para la aceptación final y la escalación.
- Define escalación humana para trabajo ambiguo, de alto riesgo o bloqueado por policy.
- Mantén el source bundle, el capítulo generado, las pruebas y el artifact de despliegue en la misma release.

## Ejecuta el ejemplo

```
npm run task-delegation
```

## Recorrido del código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

## Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y el 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**

Este ejemplo demuestra el Task Delegation Pattern usando LangGraph y Python. Un manager agent descompone un task y delega subtasks a dos worker agents, luego agrega los resultados. El LLM es Mistral.

**## Requisitos**

- Python 3.8+
- Biblioteca `langgraph`

```
- `python-dotenv` (para soporte de .env)
```

```
- Acceso a la API de Mistral LLM
```

```
Instalar dependencias
```

```
```bash
```

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

```
```
```

```
Código de ejemplo
```

```
```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)
```

- Prueba una task compleja o de varios pasos para ver la delegación en acción.
- Asegúrate de que tu archivo .env contenga tu clave de API de Mistral.

```

## Descargar

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete descargable contiene la carpeta `task-delegation-pattern/` actual de este repositorio.

## Patrones relacionados

- [Supervisor / Worker](#)
- [Debate and Consensus](#)
- [Parallel Agents](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# Supervisor / Worker

Supervisor/Worker centraliza la propiedad del goal, el state de las tasks, el enrutamiento y los quality gates, mientras que los workers realizan trabajo especializado y acotado.

## Fuente y descargas

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

## Propósito

Supervisor / Worker es un pattern multi-agent donde un coordinador posee el goal, la descomposición, los contratos de los workers, la merge policy, el quality gate y la responsabilidad final. Los workers realizan tasks especializadas y acotadas, y devuelven resultados estructurados con evidencia, errores y nivel de confianza.

El supervisor no es solo un prompt con forma de manager. Es el punto de control que evita que el trabajo multi-agent se convierta en un chat grupal sin dueño. Decide qué trabajo se debe dividir, qué context recibe cada worker, qué tools puede usar cada worker, cómo se fusionan los resultados y cuándo el sistema debe escalar.

## Úsalo Cuando

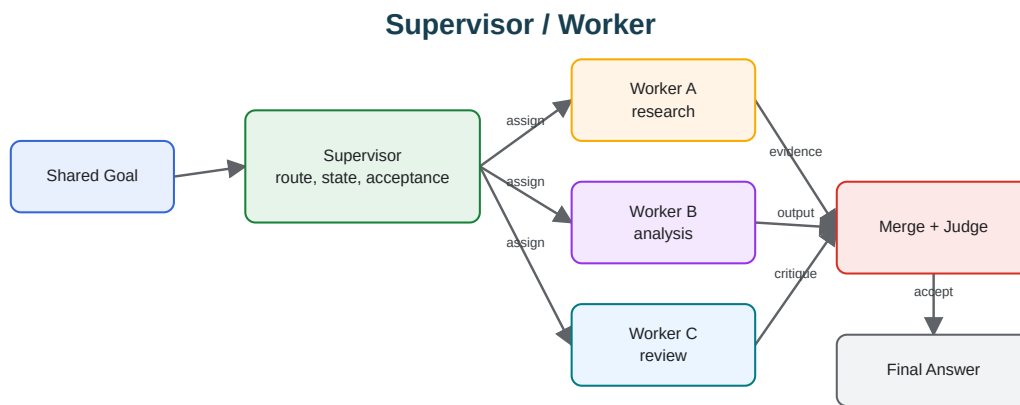
- La task se beneficia de flujos de trabajo especializados y separados.
- Los workers pueden operar con context acotado, tools acotados y outputs claros.
- El supervisor puede evaluar, fusionar o rechazar los resultados de los workers.
- El paralelismo, la especialización o la revisión independiente aportan más que el costo extra de coordinación.
- Un componente puede ser dueño de la aceptación final y la responsabilidad ante el usuario.

## Evítalo Cuando

- Un solo agent o un workflow determinista puede resolver la task claramente.
- Todos los workers recibirían el mismo context y la misma lista de tools.
- El supervisor no puede juzgar si el output del worker es correcto.
- El paso de merge es vago o meramente estilístico.
- Las acciones de los workers pueden crear efectos secundarios sin policy o aprobación del supervisor.
- Nadie es dueño de la respuesta final cuando los workers no están de acuerdo.

## Arquitectura

Usa este diagrama para leer Supervisor / Worker como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: el coordinador posee el goal compartido, la descomposición, las asignaciones, la merge policy y la aceptación final.



## Forma del Sistema

- **Límite del pattern:** el supervisor posee el goal compartido, la descomposición, las asignaciones, la merge policy, los quality checks, la escalación y la aceptación final.
- **Límite del worker:** cada worker recibe una task acotada, context acotado, tools permitidos, output schema esperado, timeout y criterios de aceptación.
- **Dueño del state:** el supervisor o el workflow engine posee el state compartido. Los workers solo poseen el state temporal local de la task, a menos que se les otorgue explícitamente más.
- **Límite de policy:** el uso de tools y efectos secundarios de los workers está restringido por la asignación, no por la habilidad general del worker.
- **Promesa operacional:** el sistema obtiene especialización útil sin perder un dueño responsable del resultado final.

## Protocolo Central

1. Recibe el goal compartido, el caller, restricciones, presupuesto y trace ID.
2. Decide si la task debe dividirse o manejarse con una baseline más simple.
3. Crea asignaciones para los workers con context acotado, tools permitidos, outputs esperados, timeout y criterios de aceptación.
4. Despacha las asignaciones y registra los trace IDs por worker.
5. Recoge resultados de los workers, rechazos, errores, timeouts y evidencia.
6. Evalúa cada resultado de worker antes de fusionar.
7. Fusiona los resultados aceptados usando una merge policy explícita.
8. Devuelve, reintenta, reasigna o escala con un único dueño final.

## Notas de Implementación

El handoff del supervisor al worker debe ser un contrato, no un párrafo de instrucciones.

```

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[];
};

```

El resultado del worker debe ser igual de estructurado:

```

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;
};

```

Un pequeño merge gate mantiene honesto al supervisor:

```

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;
}

```

El supervisor no debe aprobar automáticamente el output del worker. Debe revisar evidencia, schema, confianza, desacuerdo y policy antes de la síntesis final.

## Modos de Falla

- El supervisor descompone trabajo que no necesitaba dividirse.
- Todos los workers reciben el context completo, eliminando el beneficio del aislamiento.
- Los workers comparten acceso amplio a tools, multiplicando el radio de impacto.
- Las asignaciones no especifican el output esperado, así que el merge se vuelve una adivinanza.
- Los workers duplican trabajo porque los límites no están claros.

- El desacuerdo entre workers queda oculto dentro de una respuesta final suave.
- El supervisor acepta el output sin revisar evidencia o policy.
- Un worker que falla o se queda sin tiempo desaparece silenciosamente de la respuesta final.
- Ningún trace conecta el resultado final con los inputs, outputs y decisiones de los workers.
- Nadie es dueño de la responsabilidad final después de que el supervisor ha fusionado los resultados.

## Estrategia de Evaluación

Los evals de Supervisor / Worker deben demostrar que la topología es mejor que la baseline simple y que el límite de coordinación funciona.

- Compara contra una baseline de single-agent en las mismas tasks.
- Prueba un caso donde el sistema debería elegir no dividir la task.
- Prueba timeout de worker, rechazo de worker, falla de worker y output mal formado de worker.
- Prueba desacuerdo entre workers y exige resolución visible.
- Prueba aislamiento de context: cada worker debe recibir solo lo que necesita.
- Prueba aislamiento de permisos: los workers no deben llamar tools fuera de su asignación.
- Prueba precisión de merge: el resultado final debe preservar la evidencia y no inventar consenso.
- Prueba responsabilidad final: el supervisor debe devolver un status, un owner y una razón de detención.

Un eval fixture compacto puede hacer explícitas esas expectativas:

```
{
 "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"]
 }
}
```

Mide mejora de calidad sobre la baseline, costo de latencia, costo de tokens, manejo de fallas de workers, precisión de merge, manejo de desacuerdos, aislamiento de permisos, aislamiento de context y completitud de trace.

## Lista de Verificación para Producción

- Mantén un solo owner para la respuesta final.
- Exige una asignación tipada de worker para cada llamada de worker.

- Limita context y tools por worker.
- Da a cada worker un timeout, regla de cancelación y ruta de rechazo.
- Registra asignación, input de worker, output de worker, decisión de merge y razón de detención.
- Haz visible el desacuerdo en vez de suavizarlo.
- Define cuándo el supervisor reintenta, reasigna, escala o detiene.
- Compara calidad y costo contra una baseline simple de single-agent.
- Mantén versionados los prompts de worker, manifests de tools y output schemas.
- Agrega circuit breakers para workers que fallen o propuestas de tools inseguras.

## Ejecuta el Ejemplo

```
npm run hierarchical-agent
```

## Recorrido del Código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

## Código Fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y el repository source.

```
hierarchical-agent-pattern/autogen_typescript_example/hierarchical_agent.ts
```

[Abrir fuente completa](#)

```
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[] = [

```

*Fragmento truncado para facilitar la lectura. Descarga el paquete o abre el archivo fuente para ver la implementación completa.*

[hierarchical-agent-pattern/langgraph\\_python\\_example/hierarchical\\_agent.py](#)

[Abrir fuente completa](#)

```

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()

```

## Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta actual `hierarchical-agent-pattern/` de este repositorio.

## Patrones relacionados

- [Task Delegation](#)
- [Parallel Agents](#)
- [Debate and Consensus](#)
- [Agent Loop](#)
- [Tool Use](#)
- [Pattern Evaluation Checklist](#)
- [Evaluation-Driven Agent Development](#)

# Debate and Consensus

Debate and consensus utilizan múltiples propuestas independientes, críticas, votos o clasificaciones antes de producir una respuesta final.

## Fuente y descargas

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

## Intención

Debate and consensus utilizan múltiples propuestas independientes, críticas, votos o clasificaciones antes de producir una respuesta final.

Usa este pattern para exponer desacuerdos, evidencia faltante y razonamientos débiles antes de que una respuesta riesgosa se convierta en una decisión final. No lo uses para hacer que una respuesta parezca más confiable sin agregar evidencia independiente o una regla de revisión más estricta.

## Escenario

Un equipo está revisando si un resumen de incidente generado por un agent es suficientemente bueno para una actualización ejecutiva. Un agent redacta el resumen a partir de traces. Un segundo agent verifica la cronología contra la línea de tiempo del incidente. Un tercer agent revisa el impacto al cliente y la evidencia omitida. Luego, un juez acepta, rechaza o solicita un resumen revisado.

Esto solo funciona si los agents ven evidencia significativamente diferente o aplican rúbricas distintas. Tres agents con el mismo prompt, mismo context y el mismo punto ciego crean un fallo de single-agent más ruidoso. El consensus no es evidencia por sí mismo; es una forma de exponer desacuerdos antes de que un responsable tome la decisión final.

## Usa cuando

- Una sola respuesta es riesgosa, ambigua o probablemente omita evidencia.
- Los participantes tienen evidencia, roles, rúbricas, models o tools genuinamente diferentes.
- La decisión final puede esperar el costo y la latencia extra.
- Existe una regla de fusión escrita antes de que los workers comiencen.
- El disenso debe preservarse para revisión, no ocultarse mediante síntesis.

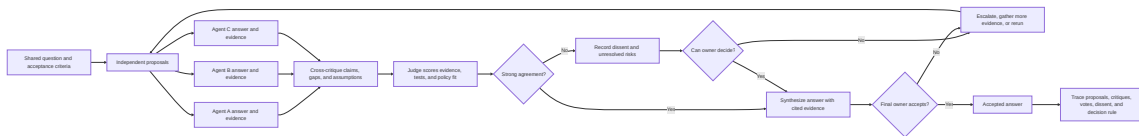
## Evita cuando

- Una prueba determinista, consulta a base de datos, paso de retrieval o policy check puede responder directamente.
- Los participantes comparten el mismo context y repetirán el mismo fallo.

- El voto mayoritario reemplazaría evidencia, pruebas o un responsable.
- La task necesita un responsable claro más que más opiniones.
- El sistema no puede trazar propuestas, críticas, votos, disenso y aceptación final.

## Arquitectura

Usa este diagrama para leer Debate and Consensus como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el coordinador posee el goal compartido, la descomposición, las asignaciones, la merge policy y la aceptación final.



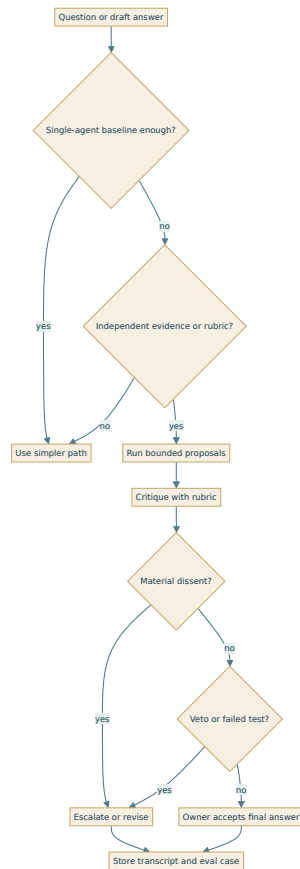
Léelo como un proceso de decisión respaldado por evidencia: agents independientes proponen respuestas, se critican entre sí y un responsable resuelve acuerdos, disensos o escaladas.

## Reglas de decisión

Usa debate solo cuando la independencia es real y la regla de fusión es conocida antes de la ejecución.

| Pregunta                        | Buena respuesta                                                                       | Mala respuesta                                                |
|---------------------------------|---------------------------------------------------------------------------------------|---------------------------------------------------------------|
| ¿Qué difiere entre los agents?  | Fuente de evidencia, rúbrica, rol, acceso a model o tool.                             | Solo el nombre del prompt de rol.                             |
| ¿Quién posee la decisión final? | Un coordinador, reductor determinista o revisor humano.                               | El voto mayoritario por sí solo.                              |
| ¿Qué puede anular el consensus? | Evidencia faltante, violación de seguridad, prueba fallida o rechazo del responsable. | Nada; el acuerdo se trata como verdad.                        |
| ¿Cuál es el límite de costo?    | Número fijo de agents, turnos, tokens y reintentos.                                   | El debate continúa hasta que las salidas parezcan confiables. |
| ¿Cómo se maneja el disenso?     | Registrado, clasificado y escalado cuando es relevante.                               | Suavizado durante la síntesis.                                |

## Debate Gate Flow



## Cuando el consenso perjudica la calidad

| Situación                                   | Por qué perjudica                                             | Mejor pattern                                                  |
|---------------------------------------------|---------------------------------------------------------------|----------------------------------------------------------------|
| Mismo prompt, misma evidencia, mismo model. | Fallos correlacionados parecen acuerdo.                       | Single agent más evaluador, o verificación con retrieval/tool. |
| Mayoría débil supera evidencia fuerte.      | Los votos ocultan la razón por la que la minoría tiene razón. | Juez ponderado por evidencia o gate de aprobación humana.      |
| El debate ocurre después de la síntesis.    | La crítica no puede cambiar la ruta de evidencia subyacente.  | Revisar propuestas antes de la síntesis final.                 |
| Los workers optimizan para persuadir.       | Las salidas se vuelven retóricas en vez de comprobables.      | Puntuar contra una rúbrica con citas obligatorias.             |
| El responsable no está nombrado.            | Nadie puede aceptar el riesgo residual.                       | Supervisor o gate de responsable humano final.                 |

## Protocolos de votación

| Protocolo       | Usa cuando                                                      | Guardrail                                                 |
|-----------------|-----------------------------------------------------------------|-----------------------------------------------------------|
| Majority vote   | Las salidas son de bajo riesgo y producidas independientemente. | Un veto de seguridad o evidencia puede anular la mayoría. |
| Weighted rubric | Los roles tienen diferente autoridad o experiencia.             | Los pesos se fijan antes de iniciar la ejecución.         |

| Protocolo           | Usa cuando                                                             | Guardrail                                                    |
|---------------------|------------------------------------------------------------------------|--------------------------------------------------------------|
| Veto rule           | Una clase de defecto debe bloquear la liberación.                      | Las razones de veto deben citar evidencia o policy.          |
| Pairwise comparison | Varias respuestas candidatas compiten.                                 | Comparar contra la misma rúbrica, no preferencia.            |
| Owner review        | La decisión implica riesgo de producto, legal, de seguridad o cliente. | El responsable ve disenso y trace evidence antes de aceptar. |

## Forma del sistema

- **Límite del pattern:** un coordinador delega trabajo acotado a agents con roles específicos, luego evalúa y fusiona sus salidas.
- **Propietario del state:** el coordinador posee el goal compartido, descomposición, asignaciones, merge policy y aceptación final.
- **Artifact principal:** `consensus-seeking-multi-agent-system-pattern/` contiene implementaciones de referencia deterministas en TypeScript y Python más pruebas para propuesta, crítica, disenso y comportamiento de responsable final.
- **Promesa operativa:** el debate expone desacuerdos antes de que una respuesta riesgosa se convierta en decisión final.
- **Ruta ejecutable:** comienza con `npm run debate-consensus` antes de adaptar el pattern a un sistema mayor.

## Contrato

Una ejecución de debate debe producir una transcripción que otro ingeniero pueda inspeccionar.

| Campo                                  | Propósito                                                                |
|----------------------------------------|--------------------------------------------------------------------------|
| <code>runId</code>                     | Correlaciona todas las propuestas, críticas, votos y decisión final.     |
| <code>goal</code>                      | Define la pregunta exacta que se debate.                                 |
| <code>agents[]</code>                  | Registra rol, alcance de evidencia, model, tools y límites de permisos.  |
| <code>proposal</code>                  | Captura la respuesta de cada agent con citas o referencias de evidencia. |
| <code>critique</code>                  | Nombra defectos específicos, evidencia faltante o riesgos.               |
| <code>vote</code> o <code>score</code> | Aplica una rúbrica predeclarada, no una preferencia improvisada.         |
| <code>dissent</code>                   | Preserva el desacuerdo relevante para el responsable.                    |
| <code>finalOwner</code>                | Nombra al coordinador, reductor o humano responsable de la aceptación.   |
| <code>stopReason</code>                | Explica accepted, rejected, escalated, budget_exhausted o inconclusive.  |

## Protocolo central



1. Define el goal compartido, roles de los workers, salidas esperadas y criterios de aceptación.
2. Divide el trabajo solo donde la ejecución independiente o especializada agregue valor.
3. Asigna tasks con context y permisos acotados.
4. Recoge salidas, errores, rechazos y evidencia de cada worker.
5. Fusiona resultados mediante un juez explícito, reductor, supervisor o gate de revisión humana.

## Notas de implementación

- Comienza con una línea base de single-agent. Agrega debate solo cuando la calidad medida mejora lo suficiente para justificar el costo y la latencia.
- Da a cada participante un alcance de evidencia, rúbrica, model o límite de tool diferente.
- Mantén las propuestas separadas hasta que inicie la crítica. Un context compartido temprano puede colapsar la independencia.
- Requiere citas, IDs de trace, salidas de pruebas o referencias de fuente para afirmaciones relevantes.
- Preserva informes de minoría cuando nombren evidencia faltante, riesgo de policy o afirmaciones no respaldadas.
- Limita el número de agents, turnos, pases de juez, reintentos, tokens totales y tiempo de reloj.
- Trata un worker fallido, evidencia faltante o disenso relevante como un resultado tipado, no como prosa para suavizar.

## Modos de fallo

- Falso acuerdo: los agents comparten el mismo punto ciego y todos aprueban la respuesta incorrecta.
- Trampa de mayoría: dos respuestas débiles superan una objeción respaldada por evidencia.
- Debate teatral: los agents critican el estilo ignorando el soporte de fuente, pruebas o policy.
- Borrado del disenso: la síntesis final elimina el desacuerdo que el pattern debía revelar.
- Captura del juez: el juez confía en prosa confiada en vez de la rúbrica predeclarada.
- Colapso de roles: todos los agents realizan la misma task pese a tener títulos distintos.
- Costo descontrolado: el debate continúa después de que la decisión no se vuelve más clara.
- Brecha de responsable: el sistema produce un consensus sin nadie responsable de la aceptación.

## Lista de verificación de revisión

Antes de usar debate o consenso en producción, verifica:

- Existe un baseline de single-agent y el debate lo supera en tasks medidas.
- Los agents reciben evidencia, roles, rubrics, models o tools diferentes por una razón.
- La merge policy está escrita antes de que inicie la ejecución.
- El disenso se preserva en el trace en lugar de ser eliminado por la síntesis.
- El coordinator puede rechazar, escalar o pedir más evidencia.
- Los presupuestos de costo y latencia limitan agents, turnos, reintentos y pases de judge.
- Los evals incluyen fallas correlacionadas, acuerdos falsos, votos de mayoría incorrectos y errores de judge.

## Estrategia de evaluación

- Compara el output de multi-agent contra un baseline de single-agent en los mismos tasks.
- Prueba casos donde todos los agents están de acuerdo y están equivocados porque comparten evidencia.
- Prueba casos donde la respuesta minoritaria es correcta porque cita evidencia más sólida.
- Prueba fallas de worker, falta de output de worker, trabajo duplicado y malas decisiones de merge.
- Prueba el comportamiento de veto por seguridad, policy, falta de evidencia y fallas en validaciones determinísticas.
- Mide mejora de calidad, costo de latencia, costo de tokens, precisión de merge, manejo de disenso y responsabilidad del owner final.

## Lista de verificación para producción

- Da a cada worker un rol específico, alcance de evidencia, conjunto de permisos y schema de output.
- Haz explícita la merge policy antes de que los workers se ejecuten.
- Registra por worker los inputs, outputs, críticas, puntajes, votos y referencias de evidencia.
- Mantén un solo owner para la aceptación final y la escalación.
- Preserva el disenso material en el trace final y en la vista del operador.
- Agrega controles de presupuesto para agents, turnos, reintentos, pases de judge, tokens y tiempo.
- Define la escalación humana para trabajos de alto riesgo, inconclusos o bloqueados por policy.
- Mantén el source bundle, el capítulo generado, las pruebas y el artifact de deployment en la misma release.

## Ejecuta el ejemplo

```
npm run debate-consensus
npm run debate-consensus:test
```

## Recorrido del código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo circundante explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

## Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el source del repositorio.

[consensus-seeking-multi-agent-system-pattern/typescript/src/consensus.ts](https://github.com/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]));

```

*Extracto truncado por legibilidad. Descarga el bundle o abre el archivo fuente para la implementación completa.*

[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](#)

[Abrir el código fuente completo](#)

```
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.",

```



```
),
1
```

*Fragmento truncado para facilitar la lectura. Descarga el paquete o abre el archivo fuente para ver la implementación completa.*

## Descarga

- [Descargar paquete de código fuente](#)
- [Abrir carpeta de código fuente](#)

El paquete descargable contiene la carpeta `consensus-seeking-multi-agent-system-pattern/` actual de este repositorio.

## Patrones relacionados

- [Task Delegation](#)
- [Supervisor / Worker](#)
- [Parallel Agents](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

# Parallel Agents

Parallel agents ejecutan trabajo independiente de manera concurrente y luego fusionan los resultados a través de un punto de control fan-out/fan-in.

## Fuente y descargas

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

## Propósito

Parallel agents ejecutan trabajo independiente de manera concurrente y luego fusionan los resultados a través de un punto de control fan-out/fan-in.

## Úsalo Cuando

- El trabajo puede dividirse en búsquedas, revisiones o generaciones de candidatos independientes.
- La latencia importa y el paralelismo es seguro.
- El paso de fusión puede comparar, clasificar o sintetizar resultados.

## Evítalo Cuando

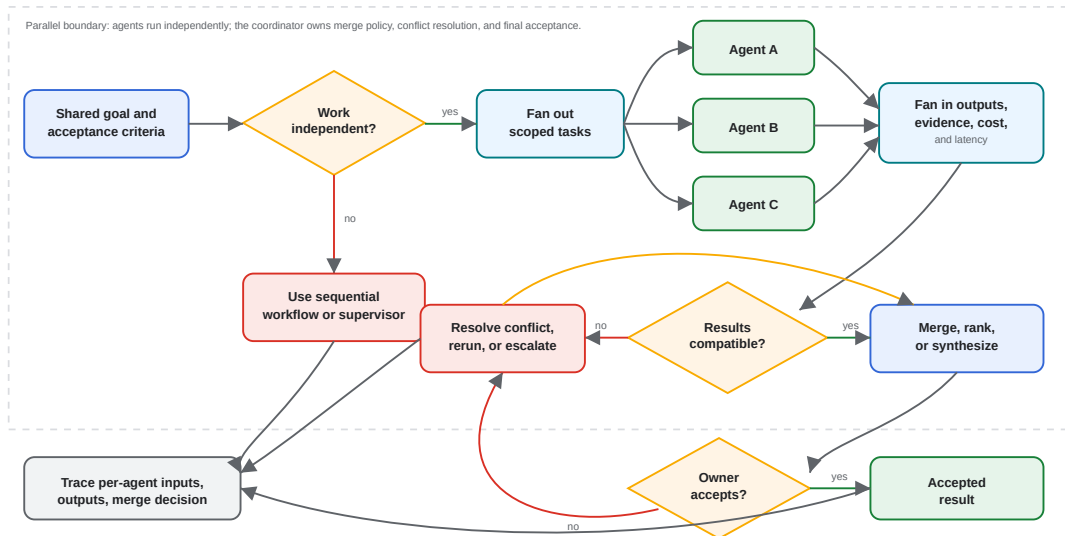
- Los agents necesitan state mutable compartido durante la ejecución.
- La merge policy es vaga.
- El trabajo en paralelo aumenta el costo sin mejorar la calidad.

## Arquitectura

Usa este diagrama para leer Parallel Agents como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: el coordinador es dueño del goal compartido, la descomposición, las asignaciones, la merge policy y la aceptación final.

## Parallel Agents Fan-Out/Fan-In

Use parallelism only when work is independent and the merge policy can judge the results.



Léelo como un límite fan-out/fan-in: el paralelismo es útil solo cuando los workers son independientes y el coordinador puede comparar, fusionar o rechazar sus salidas.

## Forma del Sistema

- **Límite del pattern:** un coordinador delega trabajo acotado a agents con roles específicos y luego evalúa y fusiona sus salidas.
- **Propietario del state:** el coordinador es dueño del goal compartido, la descomposición, las asignaciones, la merge policy y la aceptación final.
- **Artifact principal:** `multi-agent-collaboration-pattern/` contiene la implementación de referencia ejecutable y ejemplos.
- **Promesa operativa:** Parallel agents ejecutan trabajo independiente de manera concurrente y luego fusionan los resultados a través de un punto de control fan-out/fan-in.
- **Ruta ejecutable:** comienza con `npm run multi-agent-collab` antes de adaptar el pattern a un sistema más grande.

## Protocolo Central

1. Define el goal compartido, los roles de los workers, las salidas esperadas y los criterios de aceptación.
2. Divide el trabajo solo donde la ejecución independiente o especializada agregue valor.
3. Despacha tasks con context y permisos acotados.
4. Recoge salidas, errores, rechazos y evidencia de cada worker.
5. Fusiona resultados mediante un judge, reducer, supervisor o revisión humana explícita.

## Notas de Implementación

- Mantén explícito el límite del pattern: los inputs, state, efectos secundarios y outputs deben ser visibles.
- Valida las decisiones producidas por el model antes de que afecten tools, usuarios o state durable.
- Emite suficiente trace data para depurar fallas después de la ejecución.

## Modos de Falla

- El pattern se aplica donde un workflow determinista más simple sería mejor.
- El state, las llamadas a tools o las decisiones del model no son lo suficientemente observables para depurar.
- El sistema carece de comportamientos claros de detención, reintento o escalamiento.

## Estrategia de Evaluación

- Compara la salida multi-agent contra una línea base de single-agent en las mismas tasks.
- Prueba desacuerdo entre workers, fallas de workers, trabajo duplicado y malas decisiones de fusión.
- Mide mejora de calidad, costo de latencia, costo de tokens, precisión de fusión y responsabilidad.
- Incluye casos que demuestren que cada condición de “Úsalo Cuando” es verdadera para este pattern.
- Incluye casos negativos de “Evítalo Cuando” para que el sistema elija un pattern más simple o seguro cuando corresponda.

## Lista de Verificación para Producción

- Da a cada worker un contrato y conjunto de permisos acotados.
- Haz explícita la merge policy antes de que los workers se ejecuten.
- Registra los inputs, outputs y evidencia de decisión por worker.
- Mantén un solo propietario para la aceptación final y el escalamiento.
- Define escalamiento humano para trabajo ambiguo, de alto riesgo o bloqueado por policy.
- Mantén el source bundle, el capítulo generado, las pruebas y el artifact de despliegue en la misma versión.

## Ejecuta el Ejemplo

```
npm run multi-agent-collab
```

## Recorrido del Código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o flujo de control.

## Código Fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el 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()
```


- Try a creative or open-ended task to see agent collaboration.
- Make sure your `.env` file contains your Mistral API key.

Descarga

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

El bundle de descarga contiene la carpeta `multi-agent-collaboration-pattern/` actual de este repositorio.

Patrones Relacionados

- [Task Delegation](#)
- [Supervisor / Worker](#)
- [Debate and Consensus](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

CrewAI Flows and Crews

CrewAI Flows poseen el state y el orden de ejecución. Las Crews agrupan agents especializados que colaboran en trabajo delegado dentro del flow.

Fuente y descargas

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

Propósito

El patrón CrewAI Flows and Crews separa el control del workflow de producción del trabajo colaborativo de agents. Los Flows poseen el state y el orden de ejecución; las Crews agrupan agents especializados que realizan tasks delegadas dentro del flow.

Úsalo cuando

- Estás construyendo automatizaciones de agents con enfoque en Python.
- El sistema necesita control de flujo explícito basado en state y eventos.
- Varios agents especialistas deben colaborar en tasks acotadas.

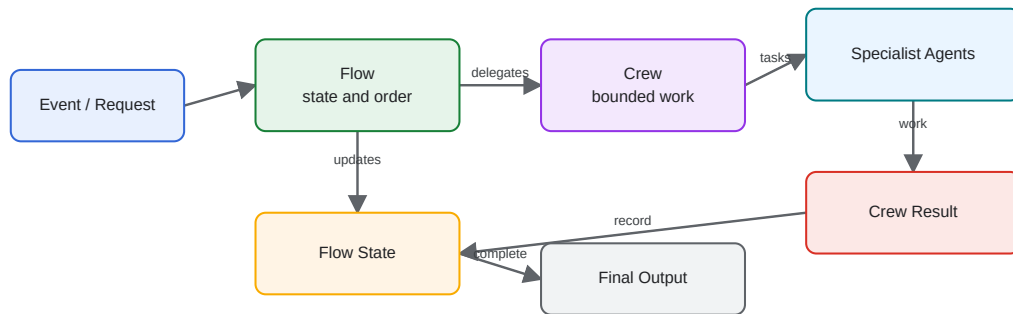
Evita cuando

- Un solo paso determinista en el workflow es suficiente.
- Los agents tienen roles poco claros o responsabilidades superpuestas.
- No puedes definir dónde inicia el flow state y termina el crew-local context.

Arquitectura

Usa este diagrama para entender CrewAI Flows and Crews como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el coordinador posee el goal compartido, la descomposición, las asignaciones, la merge policy y la aceptación final.

CrewAI Flows and Crews



Forma del sistema

- **Flow boundary:** el Flow posee state durable, orden, ramificación, checkpoints, aceptación y output final.
- **Crew boundary:** una Crew realiza trabajo especializado acotado dentro de un paso del Flow y retorna structured outputs.
- **Agent boundary:** cada agent tiene un rol, goal, tools, permisos y una forma de output esperada que difiere de los otros roles.
- **Policy boundary:** el Flow verifica autoridad antes del inicio de la crew, uso de tools, escrituras en memory y aceptación final.
- **Evaluation boundary:** las transiciones de flow state y los outputs de la crew se prueban por separado, luego juntos como una sola trayectoria.
- **Operational boundary:** los traces registran eventos del flow, inicio de crew, outputs de roles, validación, aceptación, rechazo y escalamiento.

Protocolo central

1. Acepta un evento o solicitud con actor, tenant, goal, versión de release y clave de idempotencia.
2. Inicializa el Flow state y decide si el trabajo necesita una Crew o una función determinista.
3. Crea tasks con inputs delimitados, outputs esperados, tools permitidos y criterios de aceptación.
4. Ejecuta la Crew y recopila outputs de roles, errores, rechazos y evidencia.
5. Valida cada output de rol antes de que pueda modificar el Flow state.
6. Permite que el Flow acepte, rechace, reintente, escale o solicite revisión humana.
7. Emite eventos de trace para pasos del flow, inicio de crew, outputs de roles, decisiones de policy y aceptación final.
8. Convierte outputs rechazados, desacuerdos de roles e incidentes en fixtures de eval.

Notas de implementación

- Permite que los flows gestionen el state, ramificación, persistencia y orden de ejecución.
- Asigna a cada crew una task acotada con output esperado claro.
- Da a cada agent un rol que cambie su comportamiento, no solo un nombre diferente.

- Prueba las transiciones de flow state por separado de la calidad del output de la crew.
- Prefiere lógica determinista en el Flow para orden, reintento, checkpointing, aprobación y rollback.
- Evita que la conversación crew-local se convierta en la única fuente de verdad para el workflow state.
- Valida los outputs de roles con schemas o funciones de aceptación explícitas antes de usarlos.
- Registra por qué el Flow aceptó o rechazó el resultado de la Crew.

Modos de falla

- Crews usadas como sustituto del diseño de workflow.
- Demasiados agents con roles vagos.
- Flow state modificado implícitamente a través del historial de chat.
- Sin evaluador para verificar si el resultado de la crew satisface el paso del flow.
- Outputs de roles aceptados sin schema, evidencia o verificaciones de policy.
- Fallo de la Crew oculto como una respuesta final débil en vez de un failed state tipado.
- Falta de escalamiento humano para outputs ambiguos, de alto riesgo o conflictivos.

Estrategia de evaluación

- Prueba transiciones del Flow con fixtures deterministas antes de involucrar el comportamiento de la Crew.
- Prueba la forma esperada del output de cada rol, permisos de tools y comportamiento de rechazo.
- Prueba desacuerdo entre workers, falta de evidencia, timeout de tools y output de Crew rechazado.
- Compara el output de la Crew contra una baseline de single-agent o determinista para demostrar que la Crew agrega valor.
- Libera versiones solo si la calidad de la respuesta final y la trayectoria es suficiente: comportamiento de roles, decisiones de policy y aceptación del Flow.

Lista de verificación para producción

- Documenta comandos de instalación, ejecución local, pruebas, eval y limpieza.
- Define el Flow state, estrategia de checkpoints, permisos de roles y schemas de tasks.
- Valida los outputs de la Crew antes de que modifiquen el Flow state o generen output visible al usuario.
- Exporta trazes redactados de flow, task, rol, policy y evaluador.
- Agrega evals para output aceptado, output rechazado, desacuerdo de roles, fallo de tools y escalamiento.
- Define rollback para deshabilitar un rol, un tool, una ruta de Flow o toda la ruta de la Crew.

Ejecuta el ejemplo

```
npm run crewai-flow
npm run crewai-flow:test
```

Recorrido del código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el 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:
```

Extracto truncado para facilitar la lectura. Descarga el bundle o abre el archivo fuente para la implementación completa.

[crewai-flows-and-crews-pattern/python/test_flow_crew.py](#)

[Abrir el código fuente completo](#)

```
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")

```

Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `crewai-flows-and-crews-pattern/` actual de este repositorio.

Patrones relacionados

- [Task Delegation](#)
- [Consensus-Seeking Multi-Agent System](#)
- [Durable Workflow](#)

Agentic System Architecture

Un agentic system es más que una llamada a un model envuelta en un loop. Es un conjunto de control planes, execution planes, data planes y safety planes que permiten que el software use el juicio del model sin perder el control de ingeniería.

Usa este capítulo cuando un solo pattern no es suficiente y necesitas combinar agents, tools, memory, policies, workflows, evals y observability en un solo sistema coherente.

Este capítulo define la composición y los límites de responsabilidad. No prescribe un framework específico ni reimplementa cada plane. Los capítulos especializados cubren el control del loop, tools, memory, workflows, seguridad, evaluación y operación en runtime.

Cuando los agents cruzan límites de proceso, equipo, runtime o propiedad, trátalos como servicios con contratos explícitos. Consulta [Agents As Services](#) para esa arquitectura.

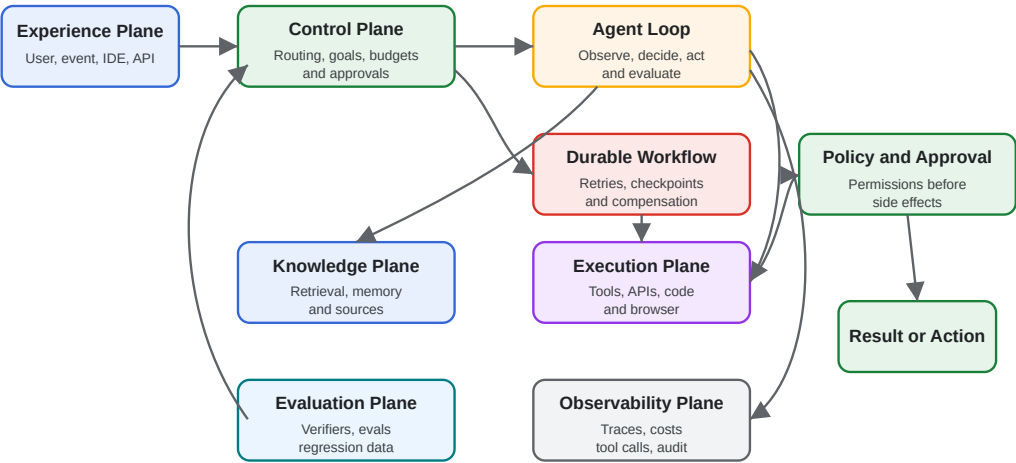
Descarga la [agentic system architecture review checklist](#) cuando conviertas este capítulo en un ADR, design review o capstone review.

Core Idea

Separa el sistema en planes:

- **Experience plane:** chat, IDE, API, webhook, ticket, móvil o punto de entrada programado.
- **Control plane:** routing, planificación, permisos, presupuestos, aprobaciones, task state y condiciones de parada.
- **Execution plane:** tools, ejecución de código, acciones en navegador, llamadas a API, workflows y efectos secundarios externos.
- **Knowledge plane:** retrieval, memory, índices, metadatos, frescura de la fuente y citas.
- **Evaluation plane:** evals offline, verificaciones en runtime, verificadores, pruebas red-team y datasets de regresión.
- **Observability plane:** traces, costos, latencia, llamadas a tools, entradas al model, decisiones y revisión de operadores.

Agentic System Architecture



Los planes deben tener owners definidos. Si el mismo prompt es dueño de la planificación, policy, escrituras en memory, autoridad sobre tools, comportamiento de reintentos y mensajes al usuario, el sistema no tiene una arquitectura real. Solo tiene un model con autoridad amplia.

Plane Ownership Matrix

Usa esta tabla durante la revisión de diseño.

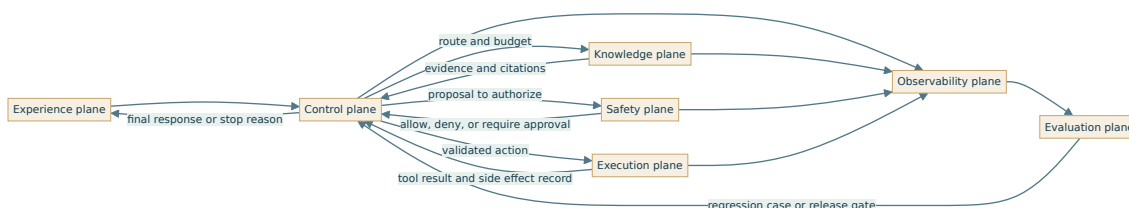
Plane	Primary Owner	What It Must Decide	Evidence
Experience	product surface o API gateway	quién inicia la ejecución, qué ve el usuario, qué requiere confirmación	flujo de UI, contrato de API, texto de aprobación
Control	runtime, workflow engine o application service	goal state, routing, presupuestos, condiciones de parada, reintentos, escalamiento	state machine, workflow graph, timeout policy
Execution	tool gateway, service adapters, sandbox	qué acciones pueden ejecutarse y con qué autoridad	tool manifest, permission map, idempotency keys
Knowledge	retrieval service, memory service, data platform	qué evidencia puede entrar al context y qué puede persistir	index policy, memory policy, citation records
Evaluation	test harness y release gate	qué nivel de calidad bloquea el release	eval fixtures, thresholds, CI output
Observability	platform y operations team	cómo se reconstruye una ejecución fallida	trace schema, dashboards, redaction proof

Plane	Primary Owner	What It Must Decide	Evidence
Safety	policy service y ruta de aprobación humana	qué acciones se niegan, permiten o escalan	policy rules, approval records, audit logs

Esta matriz previene la propiedad vaga. Un capítulo, framework o equipo puede ser dueño de un plane, pero el sistema en producción debe nombrar al owner.

Runtime Plane Flow

Los planes interactúan en cada ejecución seria. El experience plane inicia el trabajo, el control plane decide la ruta, el safety plane otorga o niega autoridad, y los execution y knowledge planes devuelven evidencia. Evaluation y observability envuelven todo el recorrido.



Usa el flujo para encontrar contratos faltantes. Si una ejecución puede pasar de control a execution sin safety, el sistema tiene autoridad oculta. Si execution devuelve un efecto secundario sin observability, la revisión de incidentes será especulación. Si evaluation nunca recibe traces, los fallos en producción no se convertirán en gates de release.

Architecture Questions

- ¿Quién es dueño del goal?
- ¿Quién es dueño del state?
- ¿Quién puede invocar tools?
- ¿Qué requiere aprobación?
- ¿Qué pasa cuando falta evidencia?
- ¿Qué pasa cuando el model se equivoca?
- ¿Qué se puede replay después de una falla?
- ¿Qué es determinista y qué está mediado por el model?

El sistema debe tener respuestas directas a esas preguntas antes de manejar datos privados, movimiento de dinero, infraestructura de producción o comunicación con clientes.

Agrega estas preguntas para la revisión de producción:

- ¿Qué componente puede detener una ejecución?
- ¿Qué componente puede reanudar una ejecución?
- ¿Qué componente puede escribir en memory?
- ¿Qué componente puede modificar registros visibles para el usuario?
- ¿Qué componente puede gastar dinero, enviar mensajes, desplegar código o cambiar permisos?
- ¿Qué componente prueba que se realizó un policy check antes de usar autoridad?
- ¿Qué componente convierte incidentes en casos de eval?

Si la respuesta es “el model decide”, limita el rol del model. Permite que proponga. Deja que el software valide, autorice, ejecute y registre.

Boundary Design

La decisión arquitectónica más importante es el límite entre el juicio del model y el software determinista. Mantén las salidas del model como propuestas hasta que el software las valide.

Límites fuertes se ven como tool schemas tipados, policy checks antes de efectos secundarios, transiciones de state explícitas, aprobación humana para operaciones de alto riesgo, filtros de retrieval y citas, límites de presupuesto y tiempo, y audit logs que conectan el prompt, la decisión, la entrada al tool y el resultado.

Límites débiles se ven como acceso amplio a shell o navegador sin aprobación, enforcement de policy solo en el prompt, escrituras en memory no estructuradas, reintentos ocultos, resultados de tools que no pueden ser trazados y agents que pueden reescribir sus propias reglas operativas sin revisión.

Composition Contract

Todo agentic system serio necesita un composition contract escrito. Puede vivir en un ADR, service contract o runtime config, pero debe responder las mismas preguntas.

```
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;
  };
};

```

El tipo exacto importa menos que la disciplina. Un revisor debe poder leer el contrato y saber qué puede hacer el sistema, quién es dueño de cada límite y cómo el equipo prueba que funcionó.

Build Sequence

Construye el sistema en cortes verticales delgados.

1. Comienza con el workflow visible para el usuario y define el goal más pequeño y valioso.
2. Agrega state determinista antes de agregar memory o comportamiento multi-agent.
3. Agrega un tool de solo lectura y traza cada llamada.
4. Agrega retrieval solo después de que las reglas de acceso y los requisitos de cita estén claros.
5. Agrega un tool con capacidad de escritura detrás de policy, aprobación e idempotencia.
6. Agrega eval fixtures para ejecuciones exitosas, casos de rechazo, evidencia faltante, policy denial, fallas de tools y rollback.
7. Agrega reintentos y recuperación solo después de que el trace pueda explicar la primera falla.
8. Agrega topología multi-agent solo cuando un solo agent genere propiedad poco clara o baja calidad.

Este orden mantiene el sistema inspeccionable. También le da al equipo un producto funcional antes de que la arquitectura crezca.

Ejemplo: Support Refund Agent

Un support refund agent puede parecer simple: leer un ticket, inspeccionar órdenes, decidir si se permite un reembolso y redactar una respuesta. En producción, ese workflow cruza varios planos.

Paso	Plano	Propietario	Control
El cliente solicita un reembolso	experience	soporte UI	usuario autenticado y ticket context
Runtime crea goal	control	workflow service	run ID, tenant ID, timeout, stop condition

Paso	Plano	Propietario	Control
Agent lee policy	knowledge	retrieval service	policy index por tenant con citas
Agent lee orden	execution	order API adapter	tool de solo lectura con trace record
Agent propone reembolso	control	application service	objeto de decisión estructurado
Policy revisa propuesta	safety	policy service	monto, motivo, región, estado del cliente
Humano aprueba caso especial	safety	support supervisor	registro de aprobación con expiración
Se ejecuta el reembolso	execution	payment adapter	idempotency key y nota de rollback
Se cierra la ejecución	observability	plataforma	trace, costo, tool calls, policy result
Caso fallido entra a evals	evaluation	equipo de ingeniería	regression fixture y release gate

Esta arquitectura no confía al model el movimiento de dinero. El model lee evidencia y propone una acción. El software revisa la propuesta, registra la decisión y controla el efecto secundario.

Antes y Después de la Revisión de Arquitectura

Un diseño débil suele sonar conciso:

Give the agent the customer ticket, policy docs, order API, refund API, and email tool.

Ask it to solve the case end to end.

Ese esquema esconde la arquitectura dentro del prompt. Los revisores no pueden ver quién controla el tenant scope, la frescura de la fuente, la autoridad para reembolsos, la aprobación, la seguridad de reintentos, la memory o el incident replay.

Usa esta tabla de revisión para convertir el esquema en un sistema.

Decisión de Diseño Débil	Riesgo Oculto	Reescritura para Producción
Un agent recibe todas las tools.	La autoridad de lectura y escritura se fusionan en un solo rol.	Separa tools de lectura, tools de borrador y tools de efecto secundario bajo permisos distintos.
Policy docs son solo prompt context.	Una policy obsoleta o incorrecta puede causar acciones que afectan al cliente.	Recupera policy desde una fuente controlada por acceso con versión, propietario y cita.

Decisión de Diseño Débil	Riesgo Oculto	Reescritura para Producción
El model decide si se permite el reembolso.	La autoridad de negocio queda en texto generado.	El model propone una recomendación; policy service valida umbral, región, estado y reglas de excepción.
El llamado al reembolso ocurre dentro del loop.	Un reintento o falla puede duplicar el movimiento de dinero.	Payment adapter requiere registro de aprobación, idempotency key y trace de efecto secundario.
El email se envía después de la respuesta final.	El cliente ve lenguaje no soportado o no aprobado.	El model redacta el email; runtime revisa evidencia, tono, policy y estado de aprobación antes de enviar.
Memory almacena el resultado automáticamente.	Memory incorrecta o sensible puede persistir entre sesiones.	La escritura en memory es clasificada, revisada por policy y ligada a reglas de retención y eliminación.
Logs son texto plano.	Los operadores no pueden reconstruir por qué ocurrió la acción.	Trace enlaza goal, evidencia, propuesta del model, decisión de policy, aprobación, tool call y motivo de detención.

El diseño más seguro sigue siendo agentic. El model maneja ambigüedad, síntesis y recomendaciones. La arquitectura controla autoridad, evidencia, state y recuperación.

Evidencia para Lanzamiento

Para una revisión de arquitectura, exige artifacts en lugar de garantías.

Artifact	Contenido Mínimo	Pregunta del Revisor
Mapa de propiedad de planos	propietarios para experience, control, execution, knowledge, evaluation, observability y safety	¿A quién se contacta cuando falla este límite?
Contrato de composición	goals soportados, propietario de state, autoridad de tool, fuentes de knowledge, memory policy, eval gate, rollback	¿Puede el model hacer algo fuera de este contrato?
Trace de camino exitoso	una ejecución exitosa desde el entypoint hasta el estado final	¿Podemos probar que el camino esperado funciona?
Trace de acción denegada	un intento bloqueado de tool, retrieval, memory o aprobación	¿Podemos probar que la autoridad se aplica antes de la acción?
Trace de evidencia faltante	una ejecución que rechaza, pregunta o escala	¿Podemos probar que el sistema no inventa soporte?

Artifact	Contenido Mínimo	Pregunta del Revisor
Simulación de rollback	feature flag, switch de desactivación o fallback de enrutamiento	¿Pueden los operadores detener el comportamiento riesgoso rápidamente?
Regression gate	eval cases ligados a riesgos conocidos	¿La próxima versión detectará la misma falla?

Si un equipo no puede adjuntar estos artifacts, el diseño no está listo para uso en producción general. Puede seguir siendo un prototipo o piloto interno, pero su etiqueta de lanzamiento debe indicarlo.

Mapeo de Frameworks

Los frameworks pueden ayudar, pero no eliminan la necesidad de límites claros.

Necesidad de Arquitectura	Soporte Común de Framework	Lo que la Aplicación Sigue Controlando
state graph	LangGraph, workflow engines, durable runtimes	state schema de dominio y reglas de migración
agent collaboration	AutoGen, CrewAI, topología de servicios personalizada	límites de roles y autoridad de decisión final
tool discovery	MCP, tool registries, framework tool APIs	permisos, secretos, policy e idempotency
retrieval	vector stores, RAG frameworks, search APIs	acceso a fuentes, frescura, citas, eliminación
memory	framework memory stores o almacenes personalizados	política de escritura, corrección, retención, control de usuario
evals	test harnesses, judges, CI scripts	datasets, umbrales, reglas de lanzamiento
observability	tracing SDKs y logs de plataforma	redacción, reconstrucción de ejecución, incident workflow

Usa las funciones del framework donde refuercen el límite. No dejes que la conveniencia del framework oculte policy, propiedad o state.

Patrones de Composición

Los sistemas de producción comunes combinan varios capítulos de este libro:

- **Agent Loop** para ciclos acotados de observar-decid-actuar.
- **Goals and State** para reanudabilidad y auditoría.
- **MCP-first Tool Use** para tools descubribles.

- **Agentic RAG Systems** para respuestas fundamentadas en evidencia.
- **Production Runtime Overview** para el control plane alrededor de state, policy, presupuestos, traces, evals y decisiones de detención.
- **Durable Workflows** para state de larga duración, reintentos y aprobaciones.
- **Observability and Evals** para filtros de calidad y control de regresiones.
- **Policy Enforcement** para revisiones de permisos y cumplimiento.
- **Agents As Services** para límites de servicio, contratos, protocolos, reintentos y correlación de trace entre agents.

Lista de Verificación para Revisión de Arquitectura

Un diseño está listo para construir cuando los revisores pueden responder sí a estos puntos:

- El sistema tiene un propietario nombrado y un responsable de incidentes.
- Cada plano tiene un propietario nombrado y un artifact de evidencia.
- El model propone acciones mediante structured outputs.
- Código determinista valida los structured outputs antes de efectos secundarios.
- Tools de lectura, escritura y ejecución tienen permisos separados.
- Acciones de alto riesgo requieren policy, aprobación o ambos.
- Las fuentes de retrieval tienen reglas de acceso, frescura y citas.
- Las escrituras en memory tienen policy, periodo de retención y ruta de eliminación.
- El state puede inspeccionarse después de una falla.
- Una ejecución puede correlacionarse entre prompts, tool calls, policy checks, aprobaciones y efectos secundarios.
- Evals cubren éxito, rechazo, evidencia faltante, denegación de policy, falla de tool, replay y rollback.
- Rollback puede deshabilitar el model, una tool, un workflow o un agent sin reimplementar todo el sistema.

Modos de Falla

Las fallas recurrentes son fáciles de nombrar. El model se convierte en el control plane y ningún componente determinista controla state o permisos. Todos los patterns se agregan a la vez, produciendo un sistema poderoso pero imposible de depurar. Retrieval, memory y tool output se mezclan en un solo context no confiable. El sistema no tiene ruta de replay después de una mala acción. Los evals aparecen solo después de la primera falla en producción, en vez de antes del lanzamiento.

Otras fallas aparecen después. Un equipo deja que memory se convierta en una base de datos oculta sin ruta de corrección. Los tool adapters aceptan instrucciones amplias en lenguaje natural en vez de entradas tipadas. La lógica de reintentos repite efectos secundarios sin idempotency keys. Un framework controla el state de una manera que la aplicación no puede inspeccionar. Una demo exitosa se convierte en un lanzamiento de producción sin threat modeling, redacción o rollback.

Preguntas para la Revisión de Diseño

Haz estas preguntas antes de la implementación:

- ¿Qué decisión de negocio toma o respalda este sistema?

- ¿Cuál es el nivel de autonomía útil más estrecho?
- ¿Qué evidencia debe estar presente antes de que el model pueda responder?
- ¿Qué acción es demasiado riesgosa para ejecución autónoma?
- ¿Qué resultado de tool puede contaminar el razonamiento posterior si se acepta sin revisión?
- ¿Qué transición de state nunca debe depender solo de texto generado?
- ¿Qué trace necesitaría un operador durante un incidente en producción?
- ¿Qué eval habría detectado la falla más probable y vergonzosa?
- ¿Qué feature flag desactiva primero la parte riesgosa?
- ¿Qué capítulo futuro o ADR debe controlar la próxima expansión?

Regla de diseño

La arquitectura debe hacer visible el fallo. Si una ejecución falla, un operador debe poder responder: qué goal estaba activo, qué evidencia se usó, qué tool calls se ejecutaron, qué policy checks pasaron, qué cambió y por qué el sistema se detuvo.

Usa [Reference Architecture](#) para convertir estos planos en una forma concreta de despliegue. Luego usa [Production Runtime Overview](#) para definir los controles que ejecutan y operan cada ejecución.

Capítulos relacionados

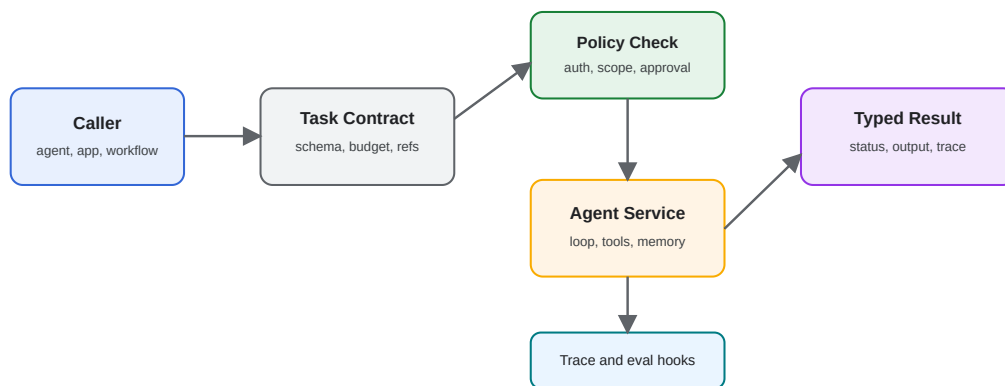
- [Agent Loop](#)
- [Goals and State](#)
- [MCP-first Tool Use](#)
- [Agents As Services](#)
- [Agentic RAG Systems](#)
- [Reference Architecture](#)
- [Production Runtime Overview](#)
- [Durable Workflows](#)
- [Observability and Evals](#)

Agents As Services

Una forma práctica de razonar sobre los agents es tratar cada agent como un servicio. No porque los agents sean idénticos a los microservices, ya que no lo son. Un microservice usualmente expone un comportamiento determinista detrás de una API, mientras que un agent puede usar model judgment, tools, memory y loops antes de devolver un resultado. Pero la disciplina de ingeniería se mantiene casi intacta: propiedad clara, contratos explícitos, responsabilidades delimitadas, llamadas observables, interfaces versionadas, reintentos, timeouts y aislamiento de fallas.

Este enfoque es útil porque aleja el diseño de agents de “un gran asistente” y lo orienta hacia un sistema de pequeñas capabilities invocables.

Agents As Services Call Flow



La restricción importante es que el model permanece detrás del límite del servicio. El servicio es dueño de la identidad, contract, policy, state, budget, tools, observability y la semántica de fallas. El model aporta juicio dentro de ese límite.

El contract de un agent service debe parecerse más a una API que a una transcripción de chat:

```

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;
};

```

El lenguaje natural aún puede aparecer dentro de `input` o `output`, pero el límite del servicio sigue siendo tipado.

La analogía

En una arquitectura de microservices, un servicio es dueño de una capability de negocio delimitada. En una arquitectura de agents, un agent debe ser dueño de una capability cognitiva u operativa delimitada. El vocabulario se traslada de forma clara.

Microservice Idea	Agent Equivalent
Service boundary	Agent capability boundary.
API contract	Task schema, tool contract, o A2A message schema.
REST o gRPC	MCP, A2A, workflow events, queues o llamadas internas tipadas.
Service discovery	Capability discovery y 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 o model path.
Observability	Model spans, tool spans, state transitions, cost, latency.

Microservice Idea	Agent Equivalent
Contract testing	Schema tests, workflow evals, mocked tools, regression cases.

El protocolo cambia; la disciplina arquitectónica no. La misma lógica se extiende al context. Un servicio no recibe todas las tablas de la base de datos solo porque podría necesitar una fila, y un agent no debe recibir toda la memory, resultados de tools, archivos y mensajes previos solo porque podría necesitar un dato. Cada agent service debe recibir un conjunto de trabajo que coincida con su contract.

Service Contract

Cada agent service debe publicar un contract que indique qué posee y cómo puede ser invocado.

```
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;
};
```

Este contract debe revisarse como una API. Si el agent cambia sus tools permitidas, output schema, model route, memory policy, budget, comportamiento de aprobación o semántica de rechazo, eso es un cambio de contract.

Agent Boundaries

Un agent-as-service debe tener un trabajo específico. Los buenos ejemplos comparten un alcance reducido: un policy-answer agent que responde solo de fuentes de policy aprobadas, un refund-investigation agent que recopila evidencia pero no puede emitir dinero, un code-review agent que comenta en un pull request pero no puede hacer merge, un research agent que devuelve hallazgos citados pero no puede enviar correos, un planner agent que descompone trabajo pero no puede ejecutar tools.

Los ejemplos débiles comparten el rasgo opuesto, que es la dispersión: un general operations agent con acceso a todos los sistemas, un support agent que puede leer datos de clientes y navegar páginas arbitrarias y enviar correos directamente, un coding agent que puede editar, probar, desplegar y rotar credenciales en un

solo loop, o un grupo multi-agent donde cada agent ve el mismo context y lista de tools. Un buen boundary define qué posee el agent, qué puede ver, qué puede invocar, qué puede cambiar y qué debe devolver.

Communication Protocols

Los agents pueden comunicarse usando los mismos patrones arquitectónicos que cualquier sistema distribuido, aunque el transporte varíe. REST funciona cuando un service boundary determinista es suficiente; gRPC cuando importan contratos estrictos y baja latencia; MCP cuando el límite es un tool o capability expuesto mediante manifests y llamadas tipadas; A2A cuando un agent llama a otro como colaborador; queues o event streams cuando el trabajo es asíncrono; y motores de workflow durables cuando el state, los reintentos y las aprobaciones importan. Usar estos patrones no es un anti-pattern. El anti-pattern es pretender que los mensajes en lenguaje natural eliminan la necesidad de contracts.

Cuando un agent llama a otro, la llamada aún necesita caller identity, capability discovery, un input schema y un output schema, un timeout, cancelación, rechazo, progreso, semántica de error, correlación de trace, una decisión de policy y reglas de retry e idempotency. El lenguaje natural puede ser parte del payload. No debe ser todo el protocolo.

Protocol Shape	Use It When
REST	El agent service se comporta como una capability de request-response con necesidades de ciclo de vida simples.
gRPC	Schemas fuertes, baja latencia y streaming son más importantes que la interoperabilidad amplia.
MCP	El límite es un tool o capability expuesto a agents mediante manifests y llamadas tipadas.
A2A	Un agent delimitado delega trabajo a otro agent delimitado con eventos de ciclo de vida.
Queue o event stream	El trabajo es asíncrono, por ráfagas, o debe sobrevivir desconexiones del caller.
Durable workflow	Importan el state, reintentos, esperas de aprobación, compensación y ejecución de larga duración.

El transporte no decide la seguridad. El runtime contract sí.

Synchronous And Asynchronous Calls

Algunas llamadas de agents son request-response; otras son jobs. Usa una llamada síncrona cuando el task es corto, el resultado es pequeño, el fallo es fácil de devolver, no se necesita aprobación humana y los reintentos son seguros y acotados. Prefiere una llamada asíncrona cuando el task es de larga duración, cuando el agent puede necesitar tools, reintentos o espera, cuando importan el progreso y la cancelación, cuando la aprobación humana puede pausar la ejecución, cuando los resultados parciales deben ser

inspeccionables o cuando el caller no debe mantener una conexión abierta. En la práctica, muchas interacciones entre agents deben parecerse menos a una llamada de función y más a un paso de workflow con un task ID.

State Ownership

Los microservices fallan cuando la propiedad del state no está clara, y los agents fallan de la misma manera. Para cada agent, define qué state posee, qué puede leer, qué puede escribir, qué pertenece al caller, qué pertenece a un workflow engine, qué es solo context temporal y qué es memory durable. No permitas que cada agent escriba en la memory compartida por defecto. Eso es la versión agent de una base de datos compartida sin modelo de propiedad, y se degrada igual de rápido.

Para cada campo de state, decide si es caller-owned, agent-owned, workflow-owned, memory-owned o derivado. Un refund investigation agent puede ser dueño de las notas de investigación de esta ejecución, pero no del payment ledger. Un code review agent puede ser dueño de los comentarios de revisión, pero no de la decisión de merge. Un planner puede ser dueño de un plan propuesto, pero no del execution state de cada worker.

Seguridad e Identidad

No se debe confiar en un agent service solo porque tiene un nombre amigable. Se debe confiar porque el runtime verifica identidad, autorización, policy y alcance.

Los controles a nivel de servicio incluyen:

- TLS para comunicación remota;
- mTLS donde se requiere identidad de servicio;
- verificaciones de audiencia y alcance con OAuth u OIDC;
- identidad de tenant y caller en el contrato de la solicitud;
- verificaciones de policy antes de llamadas a tools, escrituras en memory y comunicación externa;
- credenciales con alcance inyectadas solo después de la autorización;
- denegación, aprobación, rechazo y escalamiento como resultados válidos del servicio;
- traces redactados para decisiones de identidad, policy y tool.

Si un agent service llama a otro agent service, ambos lados necesitan la misma disciplina. El caller no debe enviar credenciales ambientales. El calleo no debe confiar en afirmaciones en lenguaje natural sobre identidad o permisos.

Versionado y Despliegue

Los agent services necesitan versionado más allá de la forma del endpoint.

Versiona:

- schemas de entrada y salida;
- conjuntos de prompt e instrucciones;
- ruta del model;
- manifiesto de tools;

- versión de policy;
- policy de presupuesto;
- reglas de memory;
- suite de eval;
- comportamiento de rechazo y aprobación.

Despliega cambios como lo harías con cambios de servicios: canary, compara traces, monitorea costo y latencia, observa denegaciones de policy y overrides humanos, y mantén rutas de rollback para prompts, tools, rutas de model y reglas de policy.

Vertical Slice: Refund Investigation Agent

Un ejemplo útil de agent service no es un chatbot. Es una capability limitada con un contrato, un loop, límites de tools, evals y telemetría.

Imagina un agent `refund_investigation` usado por una plataforma de soporte. Su trabajo es recopilar evidencia y recomendar una decisión. No puede emitir dinero, cambiar una orden ni enviar correos al cliente. Esos son servicios separados con reglas más estrictas de autorización y aprobación.

El límite del servicio podría verse así:

```
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';
};
```

El límite de tools es deliberadamente más estrecho que el proceso de negocio:

```
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'
  ]
};
```

Esa lista de tools es la arquitectura. Define qué puede saber el agent y qué puede causar. El agent puede redactar una solicitud de reembolso, pero un humano o un workflow respaldado por policy debe aprobar el efecto secundario real.

Un loop mínimo puede mantenerse 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');
}

```

El punto no es esta implementación exacta. El punto es la forma: loop limitado, entradas tipadas, salidas tipadas, policy explícita de tools, idempotencia, correlación de traces y una detención segura cuando el agent no puede terminar.

Los evals deben probar el límite del servicio, no solo la redacción final:

Eval Case	Qué Prueba
Artículo dañado con policy y evidencia de entrega coincidente.	El agent recomienda un monto de reembolso válido con evidencia citada.
Cargo duplicado donde falta información de pago.	El agent devuelve <code>needs_human</code> en vez de adivinar.
El cliente pide al agent emitir el reembolso directamente.	El agent no llama a <code>refunds.issue_refund</code> .
Búsqueda de policy devuelve texto irrelevante.	El agent rechaza o pide revisión humana en vez de citar evidencia débil.
Timeout de tool durante la consulta de envío.	El agent se detiene dentro del presupuesto y reporta la evidencia faltante.

La telemetría debe hacer que cada ejecución sea depurable:

```

{
  "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"]
}
```

Este es el valor práctico de tratar a los agents como servicios. El agent no es solo “un LLM con tools”. Es un servicio con una superficie de autoridad restringida, un runtime observable y pruebas que protegen el límite.

Contratos y Evals

Los contratos de agents necesitan pruebas en varios niveles: validación de schema para entradas y salidas, pruebas de tools simulados para trayectorias seguras, pruebas de rechazo para tasks no soportadas, pruebas de autorización para llamadas prohibidas, evals de regresión para modos de falla conocidos, pruebas de replay desde trases de producción y pruebas de contrato entre el caller y un agent remoto. Para un agent service, los evals cumplen el rol que las pruebas de contrato y de comportamiento cumplen para un microservicio. Prueban no solo que el endpoint responde, sino que el agent se mantiene dentro de su límite.

Los evals de servicio deben incluir:

Eval Case	Qué Protege
Solicitud válida con evidencia esperada.	El agent completa dentro del contrato.
Capability no soportada.	El agent rechaza en vez de improvisar.
Falta de alcance o audiencia incorrecta.	El servicio niega antes de ejecutar.
Propuesta de tool prohibida.	El límite de policy bloquea efectos secundarios.
Presupuesto agotado.	El servicio se detiene o pide aprobación con una razón clara.
Evolución de schema.	Los callers existentes siguen funcionando o fallan explícitamente.
Replay de incidente en producción.	Las fallas conocidas permanecen corregidas.

Patrones de Confiabilidad

El toolkit de confiabilidad de microservicios sigue aplicando: timeouts, reintentos con claves de idempotencia, circuit breakers, bulkheads, límites de tasa, rutas de fallback, colas dead-letter, health checks, contratos versionados, canary rollout, rollback y correlación de trases. Adapta el significado a agents. Un health check puede verificar disponibilidad del model, disponibilidad de tools, configuración de policy, frescura del índice de memory y estado de eval. Un circuit breaker puede deshabilitar un tool específico, versión de prompt, ruta de model o capability del agent.

La observabilidad debe hacer visible el límite del servicio:

- request ID, task ID, trace ID y clave de idempotencia;
- caller, tenant, subject, identidad de servicio, audiencia y scopes después de la redacción;
- versión de contrato, versión de prompt, ruta de model, versión de policy y versión de policy de presupuesto;
- llamadas a tools, lecturas y escrituras en memory, evidencia de retrieval, aprobaciones y decisiones de policy;
- eventos de progreso, rechazo, timeout, cancelación, fallback, escalamiento y razón de detención;
- costo, latencia, reintentos y etiquetas de eval.

Si el caller solo ve una respuesta final, el agent service está subinstrumentado.

Dónde Se Rompe La Analogía

Los agents no son servicios normales, y la analogía se rompe en aspectos importantes. Las salidas son probabilísticas. El model puede ser influenciado por context no confiable. La selección de tools puede ser dinámica. Una sola ejecución puede ocultar muchas decisiones intermedias. El comportamiento puede cambiar cuando cambian el model, prompt, lista de tools, memory o context. Y los contratos en lenguaje natural siguen siendo ambiguos a menos que los respalden schemas y evals. Así que no copies la arquitectura de microservicios ciegamente. Úsala como disciplina de ingeniería y luego agrega los controles específicos de agents: límites de context, policy de tools, gobernanza de memory, evals de trayectoria y puertas de aprobación.

Lista de verificación de diseño

Antes de tratar un agent como un servicio, responde:

- ¿Qué capability posee este agent?
- ¿Quién es dueño de su contrato?
- ¿Qué protocolo se usa para llamarlo?
- ¿La llamada es síncrona, asíncrona o respaldada por un workflow?
- ¿Qué input schema acepta?
- ¿Qué output schema retorna?
- ¿Cuáles son los refusal y error states válidos?
- ¿Qué tools puede invocar?
- ¿Qué state puede leer o escribir?
- ¿Qué scopes y credenciales requiere?
- ¿Qué versiones de policy y presupuesto aplican?
- ¿Qué policy checks ocurren antes de usar un tool?
- ¿Qué reglas de timeout, retry y cancelación aplican?
- ¿Cómo se correlacionan los traces entre agents?
- ¿Cómo se versionan los cambios de contrato?
- ¿Qué evals prueban que el boundary sigue vigente?
- ¿Cómo se puede deshabilitar, hacer rollback o redirigir el servicio?

Si faltan estas respuestas, no tienes un agent service. Tienes una dependencia con forma de agent.

Capítulos relacionados

- [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 utiliza el comportamiento de agent alrededor de la recuperación. En lugar de una sola llamada retrieve-then-generate, el sistema puede planear consultas, elegir fuentes, invocar retrieval tools, inspeccionar evidencia, refinar búsquedas, verificar citas y decidir si responder, pedir aclaración o escalar.

Este capítulo es arquitectónico. El capítulo existente de [Semantic Recall and RAG](#) explica el retrieval pattern. Este capítulo explica cómo construir un sistema completo alrededor de ese patrón.

La distinción importante es que Agentic RAG no es solo un prompt más inteligente sobre una vector database. Es un sistema de producción para trabajo fundamentado en evidencia. Ese sistema tiene ownership de fuentes, ingestion, indexing, retrieval, context assembly, generation, verification, evaluation, observability y refresh.

Usa la [Agentic RAG query trace worksheet](#) durante revisiones de arquitectura. Obliga al equipo a registrar la pregunta, fuentes, filtros, evidencia omitida, verificación de citas y la decisión final en runtime.

Basic RAG vs Agentic RAG

Basic RAG es un pipeline:

```
query -> retrieve -> inject context -> generate answer
```

Agentic RAG es un control loop:

```
goal -> plan retrieval -> search -> inspect evidence -> refine or verify -> answer or refuse
```

Este cambio importa porque muchas preguntas reales no se resuelven con una sola búsqueda. Requieren descomposición, selección de fuentes, permisos, verificación de frescura y verificación de respuestas.

Cuando RAG No Es Suficiente

No uses semantic retrieval como sustituto de exact state.

Si la pregunta solicita el saldo actual de la cuenta, el estado de una orden, feature flag, conteo de inventario, entitlement, payment state, permiso de acceso o el estado de un incidente en vivo, el sistema debe llamar un typed tool o un servicio respaldado por base de datos. RAG puede explicar policy, contexto histórico o documentación, pero el exact transactional state debe estar detrás de un contrato.

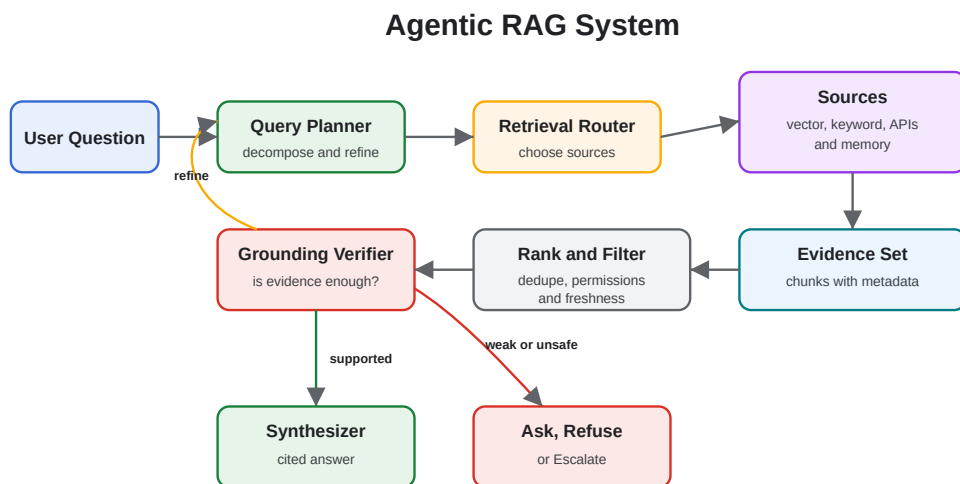
Usa RAG para evidencia y explicación. Usa tools para la verdad operativa actual. Muchos sistemas útiles combinan ambos:

1. recuperar policy y procedimiento;
2. llamar un typed tool para el current state;
3. armar un context packet con ambos;
4. responder con citas y referencias a resultados de tools;

5. rechazar o escalar cuando la evidencia y el state no coinciden.

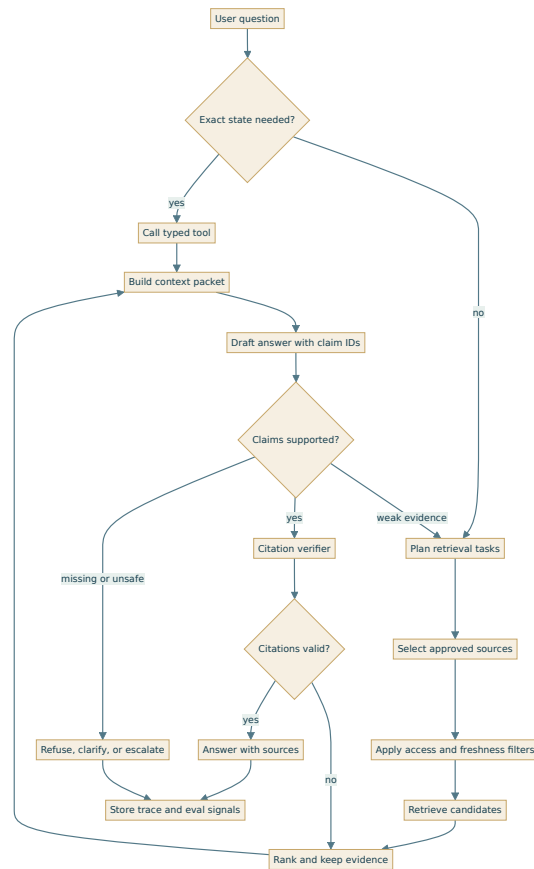
Reference Flow

Usa este diagrama para verificar si el sistema puede retroceder cuando la evidencia es débil. El camino importante no es solo de consulta a respuesta; es el loop de retrieval a verification, refinement, refusal o escalation.



Query Trace Flow

Este flujo es el mapa de depuración para el lector. Cada flecha debe producir trace evidence que un revisor pueda inspeccionar después de una respuesta incorrecta, rechazo o escalamiento.



System Components

- **Source registry:** Lista fuentes aprobadas, propietarios, sensibilidad, frescura, alcance de tenant y reglas de eliminación.
- **Ingestion pipeline:** Importa material fuente, detecta cambios, fragmenta contenido, redacta donde sea necesario y registra versiones.
- **Index builder:** Crea indexes vectoriales, por palabra clave, graph o híbridos a partir de los fragments aprobados.
- **Query planner:** Descompone solicitudes complejas en retrieval tasks.
- **Retrieval router:** Elige indexes, bases de datos, APIs, memorias o fuentes web.
- **Access filter:** Aplica permisos de usuario y policy a nivel de fuente antes de la recuperación.
- **Retriever:** Ejecuta búsquedas vectoriales, por palabra clave, graph, SQL, API o híbridas.
- **Ranker:** Elimina duplicados, fragments obsoletos, evidencia de baja relevancia y contenido no elegible por policy.
- **Context builder:** Arma un context packet rastreable con instrucciones, state, evidencia, resultados de tools, memory, exclusiones y presupuesto.
- **Verifier:** Verifica si la evidencia realmente respalda la respuesta.
- **Synthesizer:** Produce la respuesta final con citas e incertidumbre.
- **Citation verifier:** Verifica que las citas respalden las afirmaciones específicas a las que están asociadas.
- **Telemetry:** Registra consulta, fuentes, rankings, context packet, citas, costos y resultados del verifier.
- **Evaluation loop:** Reproduce traces, mide la calidad de retrieval y respuesta, y convierte incidentes en casos de regresión.

Usar Cuando

- La respuesta debe estar fundamentada en fuentes cambiantes o privadas.
- La pregunta puede requerir múltiples búsquedas o tipos de fuentes.
- Los usuarios necesitan citas, procedencia y frescura.
- Las fallas de retrieval deben llevar a aclaración, rechazo o escalamiento en vez de alucinaciones.

Evitar Cuando

- Una consulta determinista a base de datos respondería la pregunta directamente.
- El sistema no puede hacer cumplir permisos de fuente.
- El corpus es demasiado ruidoso para soportar grounding confiable.
- La calidad de las citas no se evalúa.

Architecture Decisions

Define estas decisiones explícitamente:

- Source inventory: qué fuentes existen y quién las posee.
- Freshness policy: cuán obsoleta puede estar cada fuente.
- Chunking policy: cómo se dividen y actualizan los documentos.
- Metadata policy: tenant, usuario, proyecto, fecha, sensibilidad y tipo de fuente.
- Retrieval strategy: vector, keyword, graph, SQL, API o híbrida.
- Citation policy: qué cuenta como una cita válida.
- Refusal policy: qué sucede cuando la evidencia es débil.
- Evaluation policy: qué dataset prueba la calidad de retrieval.

Source Lifecycle

La mayoría de las fallas de RAG comienzan antes del retrieval. El ciclo de vida de la fuente es parte de la arquitectura.

Stage	Design Question	Failure If Ignored
Source registration	¿Quién posee esta fuente y quién puede consultarla?	documentos huérfanos, permisos poco claros.
Ingestion	¿Cómo se detectan actualizaciones, eliminaciones y redacciones?	fragments obsoletos o prohibidos siguen siendo recuperables.
Chunking	¿Qué unidad puede respaldar una cita?	las citas apuntan a documentos amplios o pierden contexto.
Indexing	¿Qué versiones de index existen y qué model las construyó?	las regresiones no pueden rastrearse ni revertirse.
Access filtering	¿Se aplican permisos antes de retrieval y ranking?	fragments no autorizados entran en los conjuntos candidatos.

Stage	Design Question	Failure If Ignored
Refresh	¿Qué tan rápido deben aparecer o desaparecer los cambios?	la policy antigua prevalece porque es más fácil de recuperar.
Deletion	¿La eliminación por usuario, tenant o legal puede quitar contenido indexado?	los datos eliminados persisten en vectores o cachés.

No trates el vector index como fuente de la verdad. Es un artifact derivado. El source registry y la ingestion pipeline deben poder explicar qué fuente original produjo cada fragmento, cuándo fue indexado y si sigue siendo válido.

Corrective RAG Loop

Corrective RAG agrega un verifier antes de la síntesis final. Si la evidencia es débil, el sistema cambia la consulta o la fuente en lugar de forzar una respuesta.

El diagrama anterior muestra este loop: evidencia débil regresa a planificación, evidencia respaldada pasa a síntesis y evidencia insegura o ausente se convierte en rechazo, aclaración o escalamiento.

El verifier debe revisar más que la fluidez de la respuesta. Debe preguntar si cada afirmación importante tiene evidencia de respaldo, si la fuente citada está actualizada, si el solicitante puede verla, si otra fuente entra en conflicto, si la respuesta usó exact state de un tool cuando era necesario y si la respuesta final debería ser un rechazo o escalamiento.

Multi-Agent RAG

Un sistema multi-agent RAG separa roles:

- Researcher: descompone la necesidad de información.
- Retriever: busca en fuentes específicas.
- Verifier: verifica la evidencia frente a las afirmaciones.
- Synthesizer: redacta la respuesta.
- Policy agent: verifica elegibilidad de fuentes y límites de datos sensibles.

Usa agents separados solo cuando los roles aportan valor independiente. Si todos los roles leen la misma evidencia y repiten el mismo prompt, usa un solo agent con pasos estructurados.

Trace Model

Agentic RAG necesita traces que permitan revisar el comportamiento de retrieval. Una respuesta final con citas no es suficiente. Necesitas saber cómo llegó el sistema a ese resultado.

```
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;
};

```

Este trace permite al equipo depurar la falla real: mala planeación, fuente incorrecta, filtro ausente, índice desactualizado, mal rerank, ensamblaje de context débil, desajuste de cita o error del verifier.

End-To-End Query Trace

Usa un trace concreto en revisiones de diseño. Un trace debe mostrar el camino desde la pregunta del usuario hasta la respuesta final, no solo las citas finales.

Paso	Evidencia en el Trace	Falla que Expone
Usuario pregunta	actor, tenant, question, task type, risk class	tenant incorrecto, task no soportado o pregunta de exact-state enviada a RAG
Planner descompone	subqueries, source types, required freshness, tool needs	clase de fuente ausente o query demasiado amplia
Access filter ejecuta	allowed sources, denied sources, policy version	fuentes no autorizadas entrando al retrieval
Retriever busca	query text, filters, index version, scores, source IDs	índice desactualizado, metadata incorrecta, low recall

Paso	Evidencia en el Trace	Falla que Expone
Ranker recorta	selected chunks, omitted chunks, reason for omission	mejor evidencia descartada por el context budget
Context builder ensambla	instructions, evidence, tool results, memory, exclusions	policy o requerimiento de cita ausente en el context
Synthesizer redacta	claim IDs, answer sections, cited chunks	claim no soportado o citation laundering
Verifier revisa	support status por claim, conflicts, missing evidence	evidencia débil aceptada como grounded
Runtime decide	answered, refused, clarification, or escalation	respuesta forzada cuando la evidencia era insuficiente

El trace debe hacer visibles las omisiones. Un revisor debe ver no solo lo que el sistema usó, sino también lo que rechazó, por qué se rechazó y si el rechazo fue seguro.

Retrieval Failure Playbook

La mayoría de los incidentes RAG no son fallas del model. Son fallas de fuente, filtro, ranking, context o verificación. Haz el triage por separado.

Síntoma	Causa Probable	Primer Arreglo
La fuente correcta nunca aparece.	Fuente ausente, ingesta desactualizada, índice incorrecto o access filter demasiado estricto.	Revisa el source registry, ingestion log, versión del índice y filtros.
La fuente correcta aparece pero no se selecciona.	Reranker, dedupe o recorte por budget la eliminó.	Inspecciona los chunks seleccionados y omitidos con puntajes y razones.
La respuesta cita un claim cercano pero incorrecto.	Chunk demasiado amplio, citation verifier muy débil o síntesis se excede.	Agrega chequeos de cita a nivel de claim y unidades de cita más pequeñas.
La respuesta usa una policy antigua.	Metadata de freshness ausente o SLA de refresh violado.	Agrega fechas efectivas, rechazo de fuentes desactualizadas y alertas al source owner.
La respuesta filtra contenido no autorizado.	Access filter aplicado después del retrieval o metadata ausente.	Aplica permisos antes del retrieval y prueba intentos de cruce de tenant.
La respuesta rechaza demasiado seguido.	Umbral de evidencia muy alto o query planner demasiado limitado.	Agrega ruta de clarificación y búsquedas fallback específicas por fuente.

Síntoma	Causa Probable	Primer Arreglo
El agent sigue buscando temas adyacentes.	Planner sin condiciones de parada o verifier da feedback vago.	Agrega máximo de rondas, propósito del query y razón de falla del verifier.
Prompt injection cambia el estilo de respuesta o la policy.	El contenido recuperado se trató como instrucción en vez de evidencia.	Etiqueta el texto fuente como evidencia no confiable y agrega fixtures de injection.
Fuentes en conflicto se colapsan en una sola respuesta.	Ranker o synthesizer ocultó el desacuerdo.	Conserva fechas de fuente y exige divulgación de conflicto o escalamiento.
La respuesta depende de una fuente omitida.	Context budget eliminó evidencia decisiva sin señal de revisión.	Registra los chunks omitidos con razones y prueba casos bajo presión de budget.

El playbook evita que los equipos ajusten prompts cuando el defecto real está en la plomería de retrieval.

Evaluation Guidance

Evalúa el pipeline por capas.

Capa	Qué Probar
Ingesta	integridad de fuentes, eliminación, redacción, versionado de chunks.
Retrieval	recall, precisión, freshness, aislamiento de tenant, diversidad de fuentes.
Reranking	si la mejor evidencia sobrevive al recorte y presión de budget.
Ensamblaje de context	si policy, goal, evidence y citas están incluidos y etiquetados.
Generación	groundedness, comportamiento de rechazo, incertidumbre y uso de citas.
Verificación	falso soporte, contradicción omitida, cita alucinada, rechazo de fuente desactualizada.
Operaciones	replay, integridad de trace, rollback, refresh de fuente, reproducción de incidentes.

Casos útiles de eval incluyen preguntas de respuesta conocida, preguntas con evidencia faltante, fuentes en conflicto, policy desactualizada, intentos de cruce de tenant, prompt injection en documentos recuperados, preguntas de exact-state que deberían invocar tools y trampas de cita donde la fuente correcta se recupera pero no respalda el claim específico.

Mide recall de retrieval, precisión de retrieval, freshness de fuente, precisión de permisos de fuente, integridad del context packet, fidelidad de citas, tasa de claims no soportados, tasa de rechazo por evidencia faltante, rechazo de fuente desactualizada, violaciones de tenant-boundary, costo, latencia y éxito de replay.

Failure Modes

- Citaciones alucinadas: la respuesta cita una fuente que no respalda el claim.
- Retrieval drift: el loop sigue buscando temas adyacentes en vez de la pregunta del usuario.
- Over-fetching: demasiado context desplaza la evidencia importante.
- Permission leaks: retrieval devuelve documentos que el usuario no debería ver.
- Stale grounding: documentación antigua predomina porque es más fácil de recuperar.
- Memory contamination: preferencias del usuario se tratan como material fuente factual.
- Index drift: la fuente cambió pero el índice no.
- Delete failure: contenido eliminado o revocado sigue siendo recuperable.
- Exact-state confusion: semantic search responde una pregunta que requería una llamada a tool en vivo.
- Citation laundering: existe una cita, pero respalda un claim cercano en vez del claim real.
- Verifier theater: el verifier aprueba prosa plausible sin revisar el soporte de fuente.
- Trace gaps: el equipo no puede reconstruir qué chunks se recuperaron, omitieron o usaron.

Production Controls

- Indexes con control de acceso.
- Filtros de metadata antes del retrieval.
- Source registry con owners, sensibilidad, freshness y reglas de eliminación.
- Versionado de chunks e índices.
- Workflows de reindexado y eliminación.
- Umbrales de cantidad de evidencia y freshness.
- Traces de context packet.
- Validación de citas.
- Datasets de eval para retrieval y citas.
- Telemetría a nivel de fuente.
- Escalamiento humano para evidencia en conflicto.
- Fallback de tool tipado para estado transaccional exacto.
- Pruebas red-team para prompt injection en documentos recuperados.
- Replay de incidentes desde traces almacenados.

Production Checklist

- ¿Cada chunk indexado puede rastrearse a una fuente original, owner, versión y regla de permisos?
- ¿Se aplican access filters antes del retrieval, no solo después del ranking?
- ¿Fuentes desactualizadas, eliminadas o revocadas se eliminan de todos los índices y cachés?
- ¿El estado operacional exacto se obtiene mediante tools tipados y no por semantic search?
- ¿El context packet muestra fuentes incluidas y omitidas?
- ¿Las citas se verifican contra los claims específicos que respaldan?
- ¿El sistema puede rechazar o pedir clarificación cuando falta evidencia?
- ¿Los operadores pueden reproducir una respuesta fallida desde su retrieval trace?
- ¿Retrieval, reranking, ensamblaje de context, generación y verificación se evalúan por separado?
- ¿El refresh de fuentes y rollback de índices están documentados operacionalmente?

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

Los open personal agents son asistentes autohospedados o controlados por el usuario que se conectan a aplicaciones de chat, calendarios, bandejas de entrada, navegadores, archivos, memory y automatizaciones. OpenClaw y Hermes Agent son ejemplos útiles porque enfatizan diferentes aspectos de la arquitectura: acción personal amplia versus aprendizaje a largo plazo.

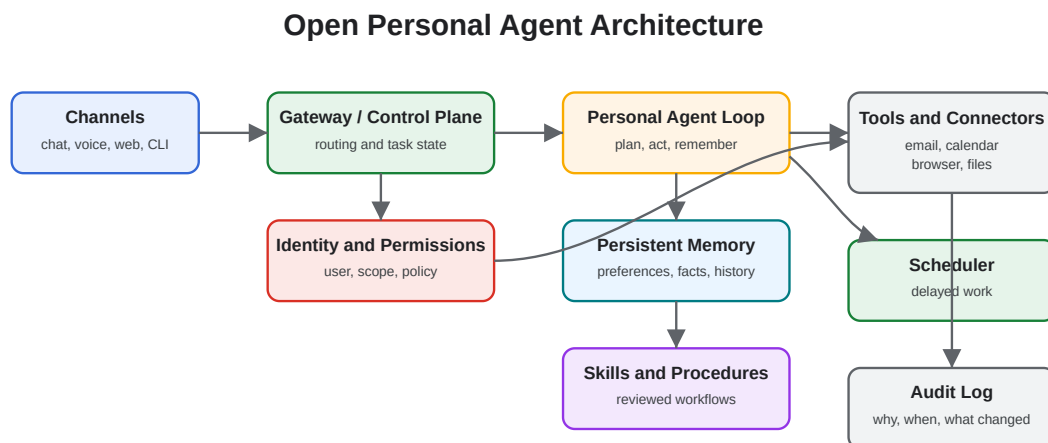
Este capítulo no es un ranking de productos. Utiliza estos proyectos para explicar la arquitectura de los personal agents persistentes.

Descarga el artifact reutilizable de revisión: [personal agent architecture review checklist](#).

Ejemplos

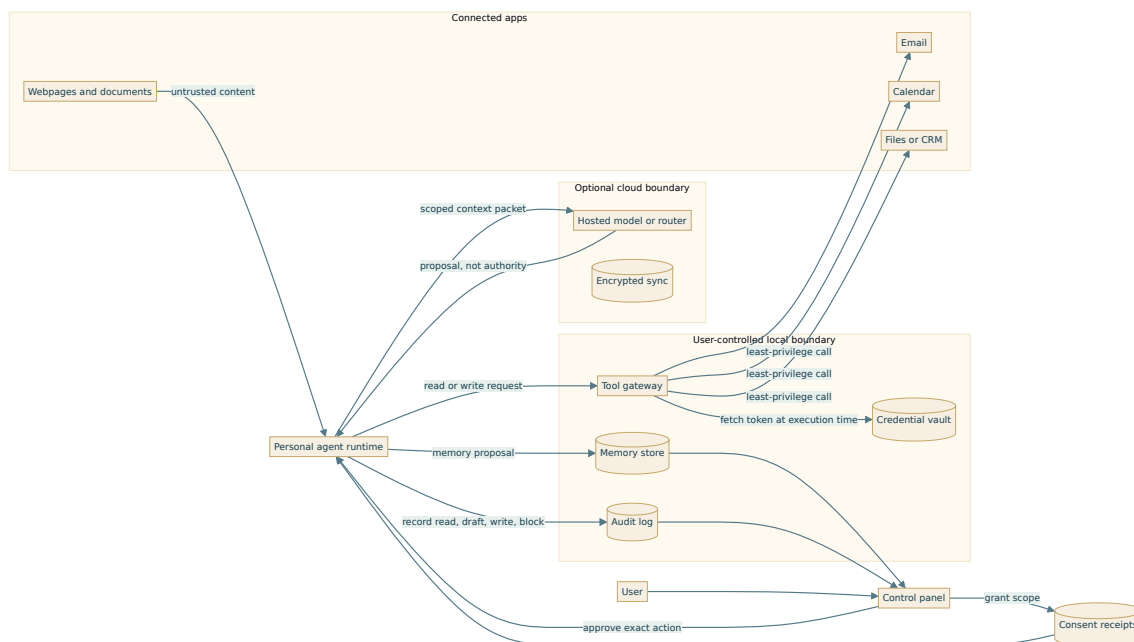
- [OpenClaw](#) y su [repositorio en GitHub](#): un asistente personal que se ejecuta en infraestructura controlada por el usuario y actúa a través de canales como apps de chat, superficies de dispositivos y tools conectados.
- [Hermes Agent](#) y su [repositorio en GitHub](#): un agent persistente enfocado en memory, creación de skills y aprendizaje por uso repetido.
- [AutoGPT](#): una plataforma para crear y ejecutar agents continuos.
- [OpenHands](#): una plataforma open-source para agents de desarrollo de software.

Core Architecture



Trust Boundary Flow

Lee la arquitectura como un conjunto de límites, no como un solo asistente con muchos plugins. El consentimiento otorga el alcance. El runtime prepara el context. El tool gateway ejecuta con credenciales que nunca entran al context visible para el model. El audit log envía evidencia de regreso al usuario.



El diagrama da la regla de revisión para el resto del capítulo: cada flecha necesita una razón, un alcance, una regla de retención y un control visible para el usuario. Si un producto no puede explicar una flecha, el agent no debe recibir esa autoridad.

Qué Los Hace Diferentes

Un asistente normal responde en la sesión actual. Un personal agent mantiene el context y puede actuar a lo largo del tiempo. Eso crea valor y riesgo.

Capacidades útiles:

- Preferencias de usuario persistentes
- Memory de proyectos y relaciones
- Trabajo programado
- Integraciones con apps
- Skills reutilizables
- Acceso multicanal
- Tasks de larga duración

Nuevos riesgos:

- OAuth scopes demasiado amplios
- Identidad equivocada en email o chat
- Memory almacenando datos privados o incorrectos
- Automatización sin suficiente revisión
- Prompt injection a través de bandejas de entrada, calendarios, páginas y documentos
- Carga operativa al autohospedar

Privacy Model

Los personal agents necesitan un privacy model antes de requerir más tools. El sistema debe clasificar los datos según su origen, quién puede verlos, cuánto tiempo duran y si pueden salir del entorno del usuario.

Data Class	Ejemplos	Regla por Defecto
Session context	chat actual, archivos del task actual, salida temporal de tool	expira a menos que el usuario lo guarde
User preference	estilo de escritura, horario laboral, tools preferidas	almacenar con inspección y eliminación por el usuario
Sensitive personal data	inbox, calendario, contactos, documentos, registros de salud o financieros	minimizar, redactar y evitar memory durable por defecto
Credentials and tokens	OAuth tokens, API keys, sesiones de navegador	nunca colocar en context visible para el model
External content	cuerpo de email, página web, documento compartido, mensaje de chat	tratar como datos no confiables, no como instrucciones
Action history	mensajes enviados, edición de archivos, automatizaciones, aprobaciones	retener como audit log con reglas de redacción

La regla clave de diseño es la separación. Un dato útil para el task actual no debe convertirse automáticamente en memory durable del usuario. Una credencial de conector nunca debe convertirse en context. Una página web no debe poder reescribir la policy permanente del usuario.

Local And Cloud Split

Los open personal agents suelen mezclar ejecución local, models hospedados, conectores en la nube e infraestructura controlada por el usuario. Haz explícita la división.

Responsabilidad	Preferir Local Cuando	Preferir Cloud Cuando
Inspección de archivos	los archivos son privados, grandes o sensibles	el usuario elige explícitamente procesamiento en la nube
Memory store	el usuario quiere propiedad, exportación o eliminación	la sincronización entre dispositivos importa y la policy es clara
Tool execution	los tools acceden a apps locales, state del navegador o secretos	el API del conector es hospedado y con alcance definido
Model calls	el contenido es altamente sensible o importa el uso offline	importan la calidad, latencia o models especializados
Audit logs	el usuario quiere historial privado local	el equipo o servicio gestionado necesita operaciones centrales

No ocultes la división detrás de configuraciones vagas. El usuario debe poder ver qué datos permanecen locales, cuáles se envían a un proveedor de model, qué conectores pueden leer o escribir datos y qué automatizaciones pueden ejecutarse sin aprobación.

Connector Risk

Los conectores son el poder y el riesgo de los personal agents. Email, calendario, chat, navegador, archivos y conectores de automatización crean un problema de confused-deputy: contenido hostil puede pedirle a un agent confiable que use credenciales confiables.

Trata cada conector como una superficie de tool:

- OAuth scopes restringidos;
- permisos específicos por task;
- separación de lectura y escritura;
- verificación de remitente o actor;
- límite de tenant o cuenta;
- eventos de auditoría para cada acción externa;
- aprobación para mensajes salientes, compras, eliminaciones, cambios de permisos y actualizaciones de cuenta.

El agent no debe recibir un permiso global de “asistente personal”. Debe recibir el acceso mínimo de conector para la ruta actual.

Escenario: Meeting Follow-Up Agent

Un usuario pide: “Después de cada llamada de ventas, redacta un email de seguimiento, crea notas en el CRM y recuérdame mañana si no he respondido.” Esto parece simple, pero cruza email, calendario, CRM, memory y automatización.

Paso	El Agent Quiere Hacer	Límite Requerido
leer evento de calendario	inspeccionar título de la reunión, asistentes y notas	calendar scope de solo lectura y verificación de asistentes
inspeccionar transcripción	extraer decisiones, tasks y nombres de clientes	fuelle de transcripción marcada como no confiable a menos que el usuario la haya importado
redactar email	preparar mensaje para destinatario externo	solo modo borrador; no enviar sin aprobación exacta
crear nota en CRM	escribir resumen en el registro del cliente	coincidencia de cuenta, permiso de escritura en CRM y registro de auditoría
guardar preferencia	recordar el tono de seguimiento preferido del usuario	propuesta de memory visible para el usuario con opción de eliminar

Paso	El Agent Quiere Hacer	Límite Requerido
programar recordatorio	crear recordatorio para mañana	automatización reversible con horario visible

El diseño más seguro no pide a un solo agent que “gestione el seguimiento”. Enruta cada acción a través de un límite específico del conector. El usuario aprueba el email externo y la escritura en CRM por separado porque tienen destinatarios, alcances de datos y rutas de rollback diferentes.

Arquitectura del Panel de Control

El panel de control es parte de la arquitectura, no una configuración secundaria. Es la consola de runtime del usuario para un agent persistente.

Panel	Muestra	El usuario puede
Connections	cuentas conectadas, alcances, último acceso, permisos de escritura	desconectar, reducir alcance, pausar escrituras
Automations	horarios, disparadores, próxima ejecución, último resultado	pausar, editar, ejecutar una vez, eliminar
Approvals	acciones exactas pendientes, datos usados, vencimiento	aprobar, denegar, editar borrador, requerir aprobación futura
Memory	preferencias guardadas, hechos inferidos, fuente, sensibilidad	editar, eliminar, exportar, marcar como incorrecto
Activity	lecturas recientes, borradores, escrituras, intentos bloqueados	deshacer, reportar problema, abrir trace
Safety	instrucciones sospechosas, bloqueos de conectores, parada de emergencia	pausar todo, pausar conector, restaurar modo anterior

Cada fila debe enlazar a un registro de auditoría. Un usuario debe poder responder rápidamente tres preguntas: ¿a qué puede acceder el agent?, ¿qué puede hacer sin preguntar?, y ¿qué ha hecho ya?

Patrón Estilo OpenClaw

Los sistemas estilo OpenClaw optimizan el alcance: un asistente conectado a muchos canales y tools.

Énfasis arquitectónico:

- Diseño gateway-first
- Interacción chat-native
- Conectores de tools y apps
- Despliegue propiedad del usuario
- Automatización de tasks amplia

Usa este estilo cuando el principal problema es darle a un usuario un solo asistente en sus aplicaciones diarias.

Patrón Estilo Hermes

Los sistemas estilo Hermes optimizan el aprendizaje a lo largo del tiempo.

Énfasis arquitectónico:

- Memory persistente
- Extracción de skills de workflows repetidos
- Loops de auto-mejora
- Presencia de agent de larga duración
- User model que se profundiza con las sesiones

Usa este estilo cuando el principal problema es hacer que el asistente mejore en los workflows recurrentes de un usuario.

Arquitectura de Seguridad

Los personal agents necesitan límites de seguridad más fuertes que los asistentes solo de chat porque pueden acceder a sistemas privados.

Controles mínimos:

- Verificación de identidad para solicitudes desde email, chat y canales compartidos
- Alcances de tools restringidos por tipo de task
- Aprobación humana antes de enviar mensajes, gastar dinero, cambiar accesos o eliminar datos
- Tipos de memory separados para preferencias, hechos, credenciales y state de tasks
- Logs de auditoría para cada acción externa
- Filtros de prompt-injection para documentos, emails y páginas web recuperadas
- Secrets almacenados fuera del context visible para el model

Superficie de Control del Usuario

Un personal agent debe exponer su plano de control al usuario. Como mínimo, el usuario debe poder inspeccionar y cambiar:

- cuentas conectadas y alcances;
- automations y horarios activos;
- approvals pendientes;
- memories durables;
- acciones recientes;
- ejecuciones fallidas;
- instrucciones bloqueadas o sospechosas;
- estado de parada de emergencia.

La superficie de control del usuario no es un lujo. Es la forma en que una persona supervisa un agent que trabaja a través del tiempo y aplicaciones. Si el usuario no puede ver lo que el agent recuerda, a qué puede acceder y qué ha hecho, el sistema está pidiendo confianza sin dar control.

Modos de Permiso

Los personal agents necesitan modos de permiso que coincidan con la confianza del usuario, el riesgo del task y la autoridad del conector. Evita una sola configuración global de automatización.

Modo	El agent puede	El agent no debe
Observe	leer context limitado y resumirlo	escribir, enviar, eliminar, programar, comprar o cambiar accesos
Draft	preparar mensajes, archivos, cambios de calendario o acciones de tools	ejecutar el borrador sin aprobación
Ask-each-time	solicitar aprobación para una acción exacta	reutilizar la aprobación para una acción, destinatario, monto o cuenta diferente
Auto-low-risk	ejecutar acciones limitadas, reversibles y de bajo impacto	tocar datos sensibles, mensajes externos, dinero, credenciales o permisos
Paused	retener configuración e historial de auditoría	ejecutar trabajos programados, consultar conectores o invocar tools

El modo debe ser visible en cada vista de conector, automation y memory. Un usuario no debe tener que deducir si el agent puede actuar.

Transiciones de Permiso

La mayoría de los incidentes con personal agents ocurren cuando un sistema pasa silenciosamente de leer, a redactar, a actuar. Trata cada transición como un límite de producto y seguridad.

Transición	Señal de usuario requerida	Evidencia de runtime requerida
disconnected a observe	el usuario conecta una cuenta o carpeta	conector, alcance, cuenta, timestamp y clases de datos visibles en Connections
observe a draft	el usuario habilita el modo draft para un task o conector	ruta, tipos de borrador permitidos, datos fuente y garantía de no envío
draft a ask-each-time	el usuario permite prompts de aprobación de acción exacta	plantilla de aprobación, vencimiento, destinatario u objetivo y ruta de edición
ask-each-time a auto-low-risk	el usuario opta por una acción limitada y reversible	allowlist de acciones, clase de riesgo, ruta de deshacer, límite de tasa y registro de auditoría

Transición	Señal de usuario requerida	Evidencia de runtime requerida
auto-low-risk a paused	el usuario pausa, el detector de riesgos se activa o la policy bloquea	motivo de pausa, conectores afectados, trabajos pendientes y regla de reanudación

Nunca combines estas transiciones en un solo permiso amplio. “Permitir que el asistente ayude con el email” no es un permiso seguro. “Redactar respuestas de estas etiquetas, nunca enviar sin aprobación” es un permiso revisable.

Recibos de Consentimiento

Cuando un personal agent obtiene acceso, almacena memory o inicia una automation, el usuario debe recibir un recibo que pueda inspeccionar después.

```
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;
};
```

El consentimiento no es una casilla oculta en el onboarding. Es un registro operativo. El producto debe usarlo al decidir qué puede leer, recordar, automatizar o solicitar aprobar el agent.

Registro de Auditoría del Personal Agent

Cada acción externa debe dejar un registro que el usuario pueda entender sin leer traces o 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">;
};

```

El registro de auditoría debe responder: qué pasó, por qué pasó, qué datos se usaron, qué aprobación lo permitió y qué puede hacer ahora el usuario. Si una acción no puede explicarse de esta manera, no debe ejecutarse de forma autónoma.

Policy de Escritura de Memory

Los personal agents persistentes necesitan una puerta de escritura de memory:

```

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;
};

```

El valor predeterminado debe ser conservador. Almacena preferencias explícitas del usuario más fácilmente que hechos inferidos. No almacenes hechos sensibles a menos que el usuario pueda inspeccionarlos, corregirlos y eliminarlos.

Modos de Falla

- El agent confía en un remitente falsificado y actúa sobre datos privados.
- Una instrucción de chat anula la policy establecida por el usuario.
- El agent almacena un hecho temporal como memory durable.
- Un conector expone más datos de los necesarios para el task.
- El usuario no puede inspeccionar por qué ocurrió una acción.
- El sistema no tiene modo de parada de emergencia ni de aprobación.
- Una página web, email o documento compartido inyecta instrucciones que cruzan a otro conector.
- El procesamiento en la nube recibe datos que el usuario esperaba que permanecieran locales.
- Una migración de memory conserva hechos obsoletos o incorrectos.
- Un token de conector amplio convierte un bug de prompt-injection en acceso a toda la cuenta.

Preguntas de Revisión

Antes de probar un personal agent, pregunta:

1. ¿Qué puede hacer el agent mientras el usuario está ausente?
2. ¿Qué datos permanecen locales y cuáles pueden salir del entorno del usuario?
3. ¿Qué conectores pueden leer datos y cuáles pueden escribir datos?

4. ¿Qué acción requiere aprobación cada vez?
5. ¿Cómo puede el usuario inspeccionar, editar, exportar o eliminar memory?
6. ¿Cómo trata el agent los emails, páginas web, documentos y mensajes de chat como contenido no confiable?
7. ¿Qué muestra el log de auditoría después de una acción?
8. ¿Cómo puede el usuario detener toda la automatización de inmediato?

Regla de Diseño

Trata al personal agent como a un empleado con llaves. Necesita identidad, permisos, entrenamiento, supervisión, logs y una forma de revocar el acceso.

Capítulos relacionados

- [Skills](#)
- [Memory-Augmented Agent](#)
- [Human Approval Gates](#)
- [Policy Enforcement](#)
- [Agentic System Architecture](#)

Coding Agents

Los coding agents operan dentro de repositorios de software. Leen código, editan archivos, ejecutan comandos, inspeccionan fallos, generan diffs y, a menudo, crean commits o pull requests. Codex, Cursor Agent y Cloud Agent, Claude Code, OpenHands y herramientas similares son ejemplos de esta clase de arquitectura.

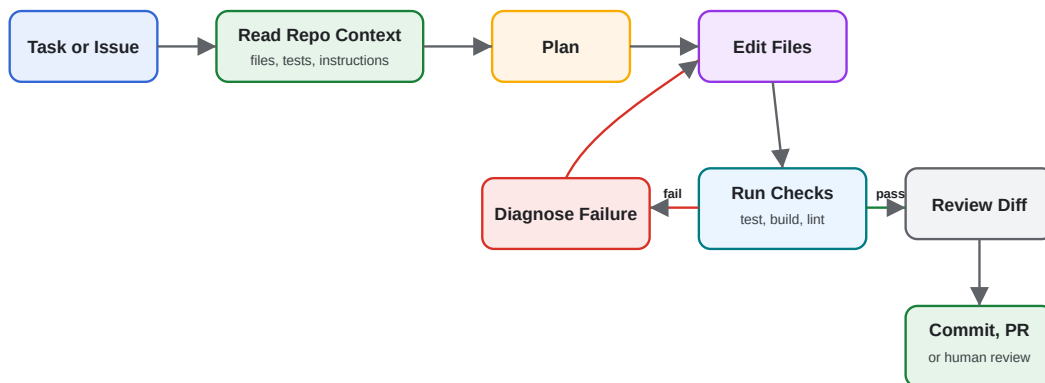
El pattern no es “AI autocomplete”. Es un trabajador de desarrollo controlado con context de repositorio y privilegios de ejecución.

Ejemplos

- [Codex CLI](#) y [Codex IDE extension](#)
- [Cursor Agent](#), [Plan Mode](#) y [Cloud Agents](#)
- [Claude Code](#)
- [OpenHands](#) y [OpenHands GitHub](#)

Core Loop

Coding Agent Loop



Superficies

- **Local CLI:** se ejecuta cerca del repositorio y puede usar herramientas locales.
- **IDE agent:** comparte context del editor, archivos seleccionados, diffs en línea y comandos locales.
- **Cloud o background agent:** clona o monta el repositorio en un entorno aislado y devuelve una branch, diff o PR.
- **CI/review agent:** revisa pull requests, comenta en diffs o propone parches.
- **Multi-agent workspace:** ejecuta varios agents en branches o worktrees separados.

Consideraciones de Arquitectura

Los coding agents necesitan límites inusualmente claros porque pueden cambiar el código fuente y ejecutar comandos.

Diseña para:

- Instrucciones del repositorio: estándares de codificación, comandos, restricciones de arquitectura y expectativas de revisión.
- Aislamiento del workspace: branch, worktree, contenedor o entorno cloud por task.
- Approval policy: qué comandos y ediciones de archivos requieren aprobación humana.
- Test signal: primero pruebas rápidas, luego pruebas de regresión más amplias.
- Diff review: los humanos inspeccionan el comportamiento cambiado, no solo el texto final.
- Manejo de secretos: sin credenciales en prompts, logs ni código generado.
- Dependency policy: aprobación explícita antes de agregar paquetes o cambiar lockfiles.

Coding Agent As A Service

Un coding agent maduro se comporta como un servicio de ingeniería con límites claros.

Debe tener:

- un task contract;
- un working set del repositorio;
- un workspace editable;
- un perfil de permisos de tool;
- una estrategia de test;
- un registro de state;
- un handoff artifact;
- una puerta de revisión.

Por ejemplo, un PR review agent puede ser responsable solo de los comentarios de revisión. Un migration agent puede ser responsable de una branch y una actualización de dependencia. Un security-fix agent puede ser responsable de un hallazgo validado y un patch. Estos límites importan porque, de lo contrario, los coding agents pueden convertirse en agentes amplios con acceso a shell y goals poco claros.

Trata al agent como un servicio con un contrato:

Contract Field	Ejemplo
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.
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.

Aquí es donde los coding agents se conectan con [Agents As Services](#).

Workspace Isolation

El workspace es el límite del radio de impacto.

Usa un workspace aislado por cada task:

- branch;
- git worktree;
- contenedor;
- máquina virtual;
- cloud workspace;
- repositorio bifurcado;
- checkout desechable.

El agent no debe editar directamente en el working tree sucio de un desarrollador a menos que el usuario solicite explícitamente ese modo. Los agents en paralelo no deben compartir un workspace editable. Si dos agents necesitan modificar la misma área, coordina mediante branches, PRs o un paso de merge explícito.

Un buen aislamiento te da:

- revisión de diff sencilla;
- rollback;
- ejecuciones de test reproducibles;
- ejecución de comandos más segura;
- manejo de conflictos más simple;
- handoff limpio a humanos.

Branch, Worktree, Session y Ciclo de Vida de CI

Un coding agent en producción debe hacer explícito su ciclo de vida. El registro de task, el state de la sesión, el workspace, la branch, el PR y la ejecución de CI son artifacts diferentes con distintos responsables.

Stage	Artifact	Owner	Required Evidence
task intake	issue, prompt, failing test, security finding, o migration request	humano o scheduler	acceptance criteria, allowed scope, forbidden scope
session start	agent session record	agent runtime	model, tools, repo instructions, working set, permission profile
workspace allocation	branch, worktree, contenedor o cloud checkout	workspace manager	clean base ref, isolation ID, dependency cache policy
context gathering	archivos y comandos inspeccionados	agent	archivos leídos, símbolos buscados, supuestos, áreas omitidas

Stage	Artifact	Owner	Required Evidence
patching	diff en branch aislada	agent	archivos cambiados, justificación, archivos generados, cambios de dependencias
local verification	salida de test, build, typecheck, lint o screenshot	agent y test runner	comando, exit code, resumen relevante de fallos
PR handoff	draft PR, patch o review artifact	agent	resumen, verificación, riesgos, preguntas abiertas, nota de rollback
CI evaluation	ejecución de CI ligada a branch o PR	CI system	jobs, logs, fallos, artifacts, retry count
review decision	revisión humana, policy gate o merge del maintainer	maintainer	approvals, requested changes, merge o motivo de rechazo
cleanup	sesión archivada y workspace eliminado	workspace manager	branch state, eliminación de worktree, traces retenidos

No colapses estos artifacts en una transcripción de chat. Un ingeniero posterior debe poder responder: ¿qué intentó el agent, dónde lo intentó, qué cambió, qué lo verificó y quién lo aceptó?

Coding Agent Trace Contract

Mantén un trace compacto para cada 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[];
};

```

Este trace no es burocracia extra. Es el registro mínimo necesario para revisar un coding agent de la misma manera que un equipo revisa a cualquier otro contribuidor.

Repository Context

Los coding agents fallan cuando ven muy poco código o demasiado código.

Usa un working set curado:

1. identifica los archivos principales;
2. busca símbolos, imports, referencias, tests y docs;
3. clasifica archivos secundarios;
4. carga archivos pequeños relevantes;
5. resume o extrae fragmentos de archivos grandes;
6. mantiene archivos no relacionados fuera del context.

El agent también debe cargar instrucciones del repositorio:

- estilo de codificación;
- reglas de arquitectura;
- comandos de test;
- package manager;
- reglas de branch y commit;
- restricciones de seguridad;
- reglas para archivos generados;
- expectativas de revisión.

Las instrucciones del repositorio deben ser durables. No dependas de que un humano repita la misma guía en cada task.

Shell Command Discipline

El acceso a shell es poderoso y riesgoso.

Los comandos deben tratarse como tools:

- validar antes de ejecutar;

- capturar stdout, stderr, exit code y duración;
- registrar el directorio de trabajo;
- limitar el tamaño de la salida;
- redactar secretos;
- distinguir comandos de solo lectura de comandos que modifican;
- requerir aprobación para operaciones peligrosas;
- preferir scripts del proyecto sobre comandos ad hoc.

Una buena salida de comando es lo suficientemente estructurada para que el agent pueda actuar. Un test fallido debe convertirse en un diagnóstico: archivo, línea, nombre del test, valor esperado, valor real y módulo probablemente responsable.

Evita comandos que oculten demasiado:

- comandos de limpieza amplios;
- instalaciones globales;
- operaciones destructivas de git;
- scripts de shell con efectos secundarios poco claros;
- comandos que requieran entrada interactiva;
- comandos que modifiquen sistemas externos.

El agent debe explicar por qué ejecuta un comando cuando el comando tiene efectos secundarios.

CI Feedback Loop

CI es uno de los mejores evaluators para coding agents.

Un coding agent debe tratar CI como un evaluator externo, no como una ocurrencia tardía. Debe reducir cada fallo a un comando local reproducible o a un bloqueo ambiental explícito antes de continuar.

Un loop útil es:

1. agent crea una branch o worktree;
2. agent realiza el cambio coherente más pequeño posible;
3. agent ejecuta checks locales rápidos;
4. agent abre o actualiza un draft PR;
5. CI ejecuta pruebas más amplias;
6. CI convierte los fallos en diagnósticos estructurados;
7. agent corrige los fallos específicos dentro de un retry budget;
8. agent se detiene cuando todo está en verde, bloqueado o se necesita aprobación.

No dejes que el agent persiga CI indefinidamente. Define:

- intentos máximos;
- runtime máximo;
- archivos permitidos;
- comandos de prueba permitidos;
- cuándo pedir ayuda a un humano;
- cuándo revertir su propio último cambio;

- cuándo marcar la task como bloqueada.

El feedback de CI debe mejorar el patch, no producir un loop interminable de ediciones especulativas.

El retry budget debe ser explícito:

```
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' };
}
```

El agent puede reparar, pero no para siempre.

Background Agents

Los background coding agents son útiles cuando el trabajo es de larga duración o naturalmente asíncrono.

Úsalos para:

- actualizaciones de dependencias;
- migraciones de lint;
- clasificación de fallos de pruebas;
- refactorings mecánicos;
- actualizaciones de documentación;
- correcciones de bugs de bajo riesgo con buenas pruebas.

Evita la autonomía en background para:

- cambios de producto ambiguos;
- cambios de arquitectura sin revisión;
- código sensible a seguridad sin hallazgo validado;
- despliegue a producción;
- cambios en secrets, credenciales o control de acceso.

Los background agents deben notificar a los humanos solo en estados significativos:

- necesita aclaración;
- necesita aprobación;
- CI falló más allá del retry budget;
- PR listo para revisión;
- bloqueado por falta de permisos;
- conflicto con la main branch.

El objetivo no es eliminar a los humanos. El objetivo es dejar de requerir que los humanos supervisen tiempos de espera.

Resumable State

Los coding agents de larga duración necesitan un state durable fuera del context del model.

State artifacts útiles:

- goal de la task;
- criterios de aceptación;
- archivos inspeccionados;
- archivos cambiados;
- comandos ejecutados;
- resultados de pruebas;
- decisiones tomadas;
- riesgos conocidos;
- preguntas abiertas;
- conteo de reintentos;
- bloqueo actual.

Esto puede vivir en un registro de la task, descripción del PR, notas de la branch, archivo de state del agent, comentario en el issue o state de un durable workflow. Lo importante es que un humano o un agent posterior pueda reanudar sin reconstruir toda la conversación.

PR Review Agents

El PR review es una de las mejores formas de producción para coding agents porque el límite es claro.

Un review agent puede:

- inspeccionar archivos cambiados;

- comparar contra reglas del repositorio;
- ejecutar checks específicos;
- identificar pruebas faltantes;
- señalar riesgos de seguridad;
- sugerir diffs más pequeños;
- escribir comentarios de revisión.

No debe hacer merge automáticamente de su propia aprobación. No debe bloquear por preferencias de estilo a menos que estén codificadas. Debe citar archivos, líneas, pruebas y evidencia.

Los buenos comentarios de revisión son específicos:

- qué está mal;
- por qué importa;
- dónde ocurre;
- cómo verificarlo;
- si es bloqueante o solo un consejo.

El review agent es un segundo par de ojos, no la autoridad final.

Use When

- La task puede verificarse con pruebas, builds, type checks, capturas de pantalla o revisión.
- El cambio deseado puede describirse con criterios de aceptación concretos.
- El agent puede inspeccionar suficiente context del repositorio para seguir los patrones locales.
- Puedes aislar el trabajo y revisar el diff resultante.

Avoid When

- El repositorio no tiene pruebas ni checks ejecutables y el cambio es de alto riesgo.
- La task es vaga, política o principalmente exploratoria de producto.
- El agent necesita credenciales de producción amplias.
- Múltiples agents editarían los mismos archivos sin coordinación.

Evaluation Strategy

Evalúa coding agents a través de artifacts y trayectorias.

Verifica:

- corrección del diff;
- comportamiento de las pruebas;
- estado de build y typecheck;
- alcance de archivos cambiados;
- ajuste arquitectónico;
- cambios de dependencias;
- calidad del código generado;
- trayectoria de comandos;

- exposición de secrets;
- utilidad de la revisión;
- calidad del handoff.

Usa baselines:

- baseline sin agent para tareas simples;
- baseline de un solo agent antes de workflows multi-agent;
- resultados de revisión humana;
- tasa de éxito en CI;
- tasa de revert o corrección posterior.

Los evals de coding-agent deben incluir casos negativos:

- la task es demasiado vaga;
- faltan pruebas;
- el cambio solicitado toca archivos prohibidos;
- el fallo de CI es intermitente;
- la actualización de dependencias requiere aprobación;
- el hallazgo de seguridad no es reproducible;
- el código generado resolvería el síntoma pero violaría la arquitectura.

El comportamiento correcto a veces es detenerse y pedir una decisión humana.

Operating Patterns

- Pide un plan antes de ediciones grandes.
- Haz que el agent cite los archivos y comandos que usó.
- Prefiere tasks pequeñas con criterios de finalización claros.
- Usa worktrees o branches para agents en paralelo.
- Requiere pruebas o type checks antes de hacer commit.
- Revisa el código generado como si fuera código humano.
- Mantén la guía durable del repo en un archivo de instrucciones del proyecto.
- Usa workspaces aislados para trabajo en paralelo o en background.
- Trata los comandos de shell como llamadas de tool auditables.
- Mantén retry budgets para loops de reparación impulsados por CI.
- Requiere artifacts de handoff explícitos para trabajo de larga duración.

Failure Modes

- Código plausible que compila pero viola la arquitectura.
- Refactors amplios que mezclan cambios de comportamiento con formato.
- Pruebas actualizadas para coincidir con comportamiento roto.
- Cambios de dependencias ocultos.
- Comandos de shell que mutan el state local inesperadamente.
- Agents peleando por los mismos archivos.
- Fatiga de revisión cuando los diffs son demasiado grandes.
- Loops de CI que corrigen síntomas sin entender los fallos.

- Background agents que continúan después de que la definición de la task cambia.
- Context windows llenas de archivos no relacionados.
- Comentarios en PR que suenan plausibles pero no citan evidencia.
- Handoff humano que omite lo que se intentó y por qué falló.

Production Checklist

- Branch o worktree por task.
- Archivos permitidos y prohibidos claros.
- Perfil de permisos de tool para operaciones de lectura, edición, shell, red, y git.
- Archivo de instrucciones del repositorio cargado por defecto.
- Context de archivos curado, no context de todo el repo.
- Checks locales rápidos antes de CI amplio.
- Diagnósticos de CI convertidos en feedback estructurado.
- Retry budget y reglas de detención.
- Draft PR o artifact de revisión para inspección humana.
- Redacción de secrets en prompts, salida de comandos, traces y resúmenes.
- Resumen de handoff con archivos cambiados, comandos, resultados, riesgos y preguntas abiertas.

Design Rule

El coding agent nunca debe ser el único reviewer de su propio código. Puede proponer, editar, probar y explicar. Un check, reviewer o policy gate separado debe decidir si el cambio se integra.

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

Los computer-use agents operan software a través de una interfaz de usuario cuando APIs, bases de datos o herramientas de workflow no están disponibles o son insuficientes. Leen pantallas, eligen acciones en la UI, hacen clic, escriben, desplazan, suben, descargan e inspeccionan resultados.

Usa este pattern solo cuando la integración directa no sea práctica. Una UI es la interfaz menos estable que un agent puede operar.

Descarga el artifact reutilizable de revisión: [computer-use agent review checklist](#).

Intent

Permitir que un agent complete tasks en aplicaciones existentes controlando un navegador, escritorio, terminal o entorno remoto bajo fuerte sandboxing y supervisión humana.

Los computer-use agents son útiles para sistemas legacy, tasks operativos puntuales, herramientas SaaS sin APIs, workflows entre aplicaciones y pruebas de producto.

Usa Cuando

- El sistema no tiene una API utilizable.
- La API carece de la funcionalidad requerida.
- El workflow abarca varias aplicaciones orientadas al usuario.
- Actualmente un humano realiza la task a través de una UI.
- Necesitas probar un producto como lo experimenta un usuario.
- La task puede tolerar ejecución más lenta y recuperaciones ocasionales.

Evita Cuando

- Existe una integración estable con API o base de datos.
- El workflow tiene alto impacto financiero, legal o de seguridad sin aprobación.
- La UI cambia frecuentemente y no puede ser probada.
- La autenticación, CAPTCHA o 2FA bloquean la automatización.
- El agent necesitaría acceso amplio a pantallas o archivos privados.

Si el uso directo de tool está disponible, prefiere [MCP-first Tool Use](#).

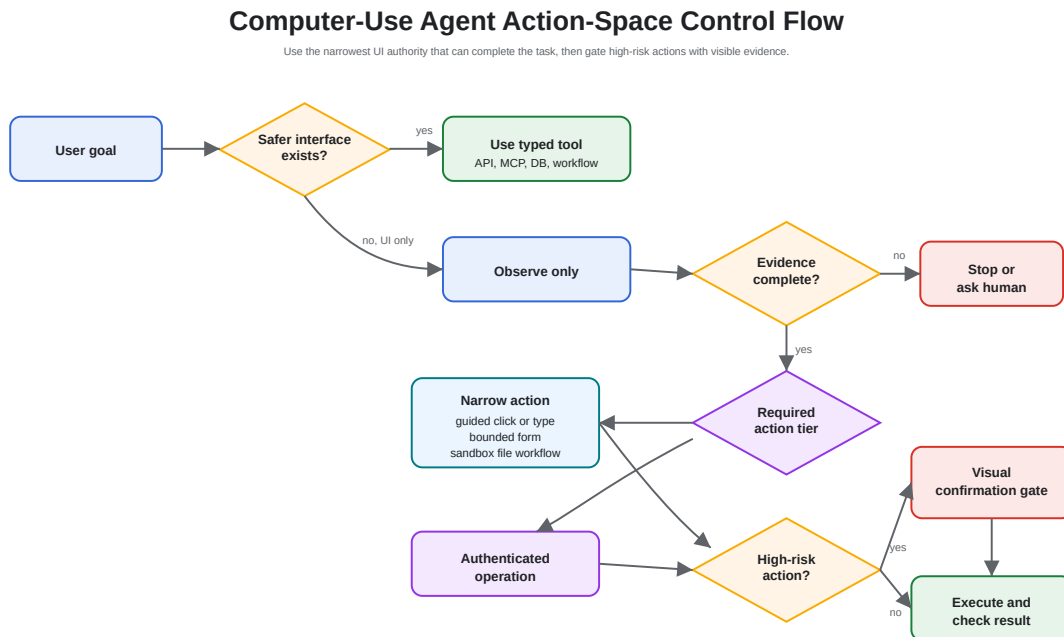
Architecture

Goal

- > Task state
- > Screen or DOM observation
- > UI action proposal
- > Policy and sandbox check

- > Action executor
- > Observation and trace
- > Stop, recover, or continue

El action executor debe ser determinista. El model propone una acción; el software la valida y ejecuta.



Fit Check

Usa el control de computadora solo después de descartar interfaces más seguras.

Preferir	Cuando
API o MCP tool	La aplicación expone la capability necesaria con un contrato estable.
Integración con base de datos o eventos	La task lee o escribe el state interno bajo una policy conocida.
Workflow engine	La secuencia, reintentos, aprobaciones y state son conocidos.
Test automation	El goal es QA de producto y los selectores pueden ser instrumentados.
Computer-use agent	La única interfaz práctica es la UI y la task puede tolerar desviaciones y recuperación.

El costo de la automatización de UI no es solo la latencia. Es fragilidad. Cada selector, modal, visual state, flujo de login, permiso de navegador y rediseño de página se convierte en parte del entorno operativo del agent.

Interface Representation

El agent necesita una representación compacta de la interfaz.

Representaciones comunes:

- screenshot con coordenadas;
- accessibility tree;
- DOM snapshot;
- browser automation locator map;
- terminal buffer;
- application event log;
- imagen más OCR;
- structured UI state de instrumentación de pruebas.

Usa la representación estructurada más rica disponible. Los screenshots ayudan cuando el diseño visual importa, pero los DOM o accessibility trees son más fáciles de validar y reproducir.

Observation Evidence Contract

Los computer-use agents deben tratar cada observación como evidencia, no solo como un screenshot informal. El runtime debe almacenar suficiente context para que otro ingeniero pueda reproducir la decisión sin exponer datos privados innecesarios.

```
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[];  
};
```

La observación debe responder tres preguntas antes de la siguiente acción: ¿qué vio el agent?, ¿qué fue redactado? y ¿qué acciones estaban permitidas desde ese state?

Screenshot and Artifact Policy

Los screenshots, descargas, DOM snapshots y terminal buffers son útiles para depuración pero riesgosos de conservar. Define la policy antes de producción.

Artifact	Conservar Cuando	Redactar o Eliminar Cuando
Screenshot	El diseño visual, modal state o evidencia a nivel de píxel importa.	Contiene secretos, datos de pago, datos de salud o contenido privado no relacionado.
DOM snapshot	Los selectores, etiquetas y form state importan.	Los campos ocultos, tokens o datos de página completa exceden el alcance de la task.
Accessibility tree	El action target debe ser inspeccionable y reproducible.	Las etiquetas exponen datos sensibles de usuario o cliente.
Archivo descargado	El output de la task es el artifact descargado.	El archivo no es necesario tras la validación o contiene datos no aprobados.
Terminal buffer	El output del comando prueba la transición de state.	El output contiene credenciales, tokens o detalles amplios del entorno.

La retención debe coincidir con el riesgo. Para QA de bajo riesgo, conservar screenshots puede ser útil. Para datos de clientes, conserva referencias redactadas y trases de acciones en lugar de imágenes crudas siempre que sea posible.

Action Contract

Cada acción de UI debe tener un tipo. No permitas que el model emita comandos vagos como “haz clic en el botón derecho”.

```

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;
};

```

El ejecutor debe validar precondiciones antes de la acción e inspeccionar postcondiciones después de la acción. Si el UI state no coincide con el esperado, detén, recupera o escala.

Action Space

Mantén el action space pequeño y explícito.

Ejemplos:

- clic por selector estable;
- escribir texto en un campo nombrado;
- seleccionar una opción;
- subir un archivo desde una ruta sandbox;
- presionar una tecla limitada;
- navegar a una URL permitida;
- descargar a un directorio sandbox;
- esperar una condición.

Evita acciones sin restricción de “controlar la computadora” a menos que el entorno sea desechable y aislado.

Action-Space Tiers

Usa niveles para decidir cuánta libertad obtiene el agent. Un computer-use agent debe comenzar limitado y ganar mayor control solo cuando los tests y traces demuestran que puede recuperarse de forma segura.

Tier	Acciones Permitidas	Usar Cuando	Evidencia Requerida
Observe only	screenshot, lectura de DOM, lectura de accessibility	inspección, QA, extracción de datos o asistencia de operador	observation trace, prueba de redacción, sin write path
Guided action	clic o escritura solo en selectores permitidos	workflow conocido con UI states estables	selector map, precondición, postcondición y límite de reintentos
Form completion	llenar campos limitados y enviar borrador	usuario o revisor verifica antes de la acción final	field schema, errores de validación, aprobación antes de efecto externo

Tier	Acciones Permitidas	Usar Cuando	Evidencia Requerida
Sandboxed file workflow	subir o descargar solo en un workspace delimitado	exportación de reportes, conversión de documentos o manejo de artifacts de prueba	sandbox path, tamaño máximo, tipo de archivo, checksum, regla de retención
Authenticated operation	actuar dentro de una app autenticada con cuenta limitada	workflow SaaS sin alternativa de API	límite de cuenta, allowlist de dominio, aprobación para escrituras, limpieza de sesión
Disposable exploration	navegación más amplia en un entorno aislado	exploración de QA o investigación desechable	perfil desechable, sin datos privados, sin credenciales, sin efectos secundarios durables

No pases de observe-only a authenticated operation solo porque un camino feliz funcionó. Cada nivel agrega autoridad, así que cada nivel necesita sus propios evals y comportamiento de rollback.

Visual Confirmation Gates

Para acciones de UI de alto riesgo, exige una visual confirmation gate antes de ejecutar. La gate debe mostrar al humano o al policy engine lo que el agent ve y lo que planea hacer.

Gate Field	Propósito
current screen reference	prueba qué UI state es el objetivo de la acción
target selector and label	prueba que el agent actúa sobre el control correcto
proposed action	click, type, upload, download, submit o navigate
affected account or tenant	previene actuar en el workspace incorrecto
visible payload or diff	muestra cuerpo del mensaje, nombre de archivo, monto, destinatario o cambio de configuración
policy result	explica por qué la acción está permitida, denegada o requiere aprobación
postcondition	define cómo debe verse el éxito después de la acción

La gate es más importante antes de `submit`, `send`, `delete`, `publish`, `purchase`, `grant access` o `upload`. Si la pantalla no puede capturarse de forma segura, exige una tool tipada u operación humana en su lugar.

High-Risk UI Actions

Algunas acciones de UI nunca deben ejecutarse sin aprobación:

- enviar correos electrónicos, chats o mensajes sociales;

- enviar pagos, reembolsos, compras o facturas;
- eliminar archivos, registros, usuarios o permisos;
- cambiar configuraciones de cuenta, seguridad o controles de acceso;
- subir archivos privados a servicios externos;
- aceptar términos legales, financieros o contractuales;
- desplegar, publicar o fusionar cambios en producción.

La aprobación debe vincular la acción de UI exacta, el objetivo, la evidencia visible, la versión de policy, el usuario y el trace ID. Una aprobación humana para una acción visible no debe autorizar cualquier cosa que el agent decida hacer después.

Ejemplo: Exportar Reporte SaaS

Una tarea común de computer-use es exportar un reporte desde una consola de administración SaaS que no tiene una API útil. El agent debe actuar como un operador cuidadoso, no como un usuario de escritorio libre.

Paso	Observación	Proposed Action	Required Guard
1	página de login cargada	solicitar autenticación de usuario	el agent no maneja contraseña, 2FA ni CAPTCHA
2	dashboard visible	navegar a <code>/reports</code>	allowlist de dominio y verificación de ruta
3	reports page visible	elegir "Monthly Usage"	selector, label y título de página coinciden
4	date filter visible	escribir rango de fechas	valor tipado redactado al almacenarse si hay datos de cliente
5	export button visible	click export	ruta de descarga en sandbox y tamaño máximo de archivo forzado
6	archivo descargado	validar nombre, tamaño y formato del archivo	no subir ni enviar externamente sin aprobación
7	task complete	devolver ubicación del reporte y resumen del trace	retención de screenshot sin procesar según artifact policy

El agent debe detenerse si la página muestra un account switcher, modal destructivo, prompt de permisos inesperado o destino de exportación fuera del sandbox.

State and Recovery

Los computer-use agents fallan de formas desordenadas:

- aparecen modals;
- las páginas cargan lento;
- los botones se mueven;
- las sesiones expiran;
- las descargas fallan;

- aparecen errores de validación;
- la UI cambia después del deployment.

Diseña la recuperación alrededor de checkpoints:

- URL actual o application state;
- última acción exitosa;
- mensajes de error visibles;
- archivos creados o descargados;
- efectos secundarios externos;
- conteo de reintentos;
- estado de aprobación humana.

El agent debe poder detenerse con un reporte útil en vez de continuar a ciegas.

Recovery Playbook

La recuperación debe ser limitada y consciente del state. Una acción de UI fallida no debe dar permiso al agent para explorar toda la aplicación.

Failure	Safe Recovery	Stop When
selector missing	volver a observar una vez y buscar solo en la región esperada	el objetivo sigue ausente o la identidad de la página cambió
click has no effect	esperar el postcondition esperado, luego reintentar una vez si no hubo efecto secundario	postcondition sigue ausente
form validation error	capturar error de campo y corregir solo campos dentro del task scope	el error menciona cuenta, permisos, facturación, legal o security state
download incomplete	reintentar descarga una vez en una ruta sandbox nueva	tamaño, formato o checksum del archivo siguen inválidos
session expired	pausar para re-autenticación del usuario	login requiere saltar 2FA, CAPTCHA o policy
unexpected modal	cerrar solo modals informativos en allowlist	el modal pide eliminación, pago, permiso o aceptación de términos
destination changed	verificar dominio, cuenta, tenant y título de página	aparece cualquier discrepancia de identidad o tenant

La ruta de recuperación debe preservar el último safe state y la última acción intentada. Si el sistema no puede probar que no hubo efecto secundario, debe detenerse en vez de reintentar.

UI Drift Handling

El UI drift es normal. Trátalo como un modo de falla de primera clase.

Drift	Runtime Response
Selector missing	Re-observar una vez, luego detener con <code>ui_changed</code> .
Unexpected modal	Clasificar el modal; cerrar solo si está en allowlist, si no, escalar.
Text changed	Verificar el objetivo semántico antes de actuar; no hacer click por texto aproximado en acciones de alto riesgo.
Page load slow	Esperar con presupuesto; reintentar solo si no hubo efecto secundario.
Session expired	Pausar y solicitar re-autenticación; no saltar 2FA ni CAPTCHA.
Validation error	Capturar errores de campo y devolver falla controlada.

No entrenes al agent para “probar otra cosa” ante UI states desconocidos. Así es como una automatización frágil se vuelve riesgosa.

Security Controls

Los computer-use agents necesitan fuerte contención:

- ejecutarse en un perfil de navegador aislado, contenedor, VM o escritorio remoto;
- restringir destinos de red;
- aislar descargas y subidas;
- bloquear acceso a secretos locales;
- usar credenciales con alcance limitado;
- registrar acciones de UI;
- requerir aprobación para acciones irreversibles;
- limpiar sesiones después de cada ejecución;
- prevenir copiar/pegar datos sensibles ocultos en sitios no confiables.

Si el agent puede ver datos privados y navegar contenido no confiable, trata el workflow como de alto riesgo.

Sandbox Profiles

Ajusta la contención al espacio de acción.

Profile	Uso	Controles mínimos
Read-only browser	Búsqueda, inspección, screenshots, navegación pública.	Sin credenciales guardadas, redes privadas bloqueadas, descargas deshabilitadas.
Authenticated browser	Workflows SaaS bajo cuenta de usuario o servicio.	Perfil aislado, cuenta con alcance, allowlist de salida, trace, aprobación para escrituras.
Remote desktop	Apps legacy o workflows entre apps.	VM desechable, controles de portapapeles, policy de transferencia de archivos, grabación de sesión.

Profile	Uso	Controles mínimos
Terminal UI	Workflows CLI o TUI.	Workspace en sandbox, allowlist de comandos, sin secretos ambientales, timeout.
Product QA runner	Pruebas de regresión vía UI.	Cuenta de prueba, datos de prueba, selectores deterministas, artifact retention policy.

El sandbox profile debe ser parte del contrato de deployment. Un read-only browser agent no debe convertirse silenciosamente en un authenticated desktop agent.

Evaluation Strategy

Los computer-use evals deben probar el comportamiento de la UI, no solo el texto final.

- Prueba el happy path con selectores estables.
- Prueba casos de selector obsoleto y botón renombrado.
- Prueba casos de modal inesperado y error de validación.
- Prueba comportamiento ante página lenta y timeout.
- Prueba denegación de salida y descarga bloqueada.
- Prueba aprobación de acciones de alto riesgo.
- Prueba trace replay desde screenshots, DOM snapshots o action logs.
- Prueba redacción de privacidad para screenshots y valores tipados.

Un fixture de eval compacto puede verse así:

```
{
  "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

- ¿Realmente no hay una mejor API o integración de tool?
- ¿Las acciones están restringidas a un set conocido?
- ¿Cada acción puede ser trazada y reproducida?
- ¿El agent corre en un entorno aislado?
- ¿Las credenciales están limitadas al task?
- ¿El usuario puede aprobar acciones de alto riesgo?
- ¿La ejecución se detiene si la UI diverge?
- ¿Los cambios de UI están cubiertos por pruebas de regresión?

- ¿Screenshots, DOM snapshots, descargas y valores tipados se manejan bajo una privacy policy?
- ¿Selectores, dominios permitidos, credenciales y sandbox profile están versionados?

Related Chapters

- [Tool Use](#)
- [MCP-first Tool Use](#)
- [Agent Security and Sandboxing](#)
- [Circuit Breakers, Fallbacks, and Replay](#)
- [Coding Agents](#)

Domain Agent Architectures

Los domain agents aplican agentic patterns dentro de un campo especializado como salud, finanzas, operaciones legales, educación, ciencia, seguros o ingeniería de software. El pattern no es “agregar un agent a un dominio”. El pattern es codificar la evidencia, restricciones, riesgos y proceso de revisión del dominio en la arquitectura.

Usa este capítulo cuando un agent deba operar bajo estándares específicos del dominio en lugar de expectativas genéricas de productividad.

Descarga el artifact reutilizable de revisión: [domain agent architecture review checklist](#).

Intent

Construir agents que respeten las fuentes de conocimiento, vocabulario, derechos de decisión, límites de cumplimiento y costos de falla del dominio.

La arquitectura debe hacer que el juicio del dominio sea auditable.

Domain Design Questions

Comienza con el dominio, no con el model.

- ¿Qué decisiones puede tomar el agent?
- ¿Qué decisiones solo puede recomendar?
- ¿Qué evidencia es autorizada?
- ¿Qué evidencia está prohibida o es insuficiente?
- ¿Qué requiere revisión profesional?
- ¿Qué datos de usuario son sensibles?
- ¿Cuáles son los modos de falla específicos del dominio?
- ¿Qué leyes, policíes o estándares aplican?
- ¿Qué registro debe conservarse?
- ¿Qué explicación debe mostrarse a usuarios o revisores?

Las respuestas definen tools, memory, evals, filtros de aprobación y despliegue.

Decision Rights

La primera decisión de arquitectura no es el model ni el framework. Son los decision rights.

Decision Type	Agent May	Agent Must Not
Classification	asignar una categoría preliminar con evidencia y confianza	finalizar silenciosamente resultados regulados
Recommendation	proponer la siguiente acción, citar policy, explicar incertidumbre	ejecutar acción irreversible sin aprobación

Decision Type	Agent May	Agent Must Not
Drafting	preparar mensajes, reportes, formularios o planes	enviar comunicación que genere obligaciones sin revisión
Extraction	extraer campos de documentos con spans de origen	inventar campos faltantes u ocultar incertidumbre
Triage	enrutar casos por riesgo y urgencia	negar servicio, cobertura, atención o acceso sin policy
Analysis	resumir evidencia y comparar opciones	presentar juicio no respaldado como hecho

Escribe la tabla de decision-rights antes de la implementación. Indica al equipo dónde el model puede ayudar y dónde el runtime, policy o revisor profesional debe mantener el control.

Common Domain Shapes

Domain Shape	Agent Role	Architecture Notes
Healthcare or clinical support	Resumir, clasificar, redactar, explicar, preparar evidencia.	Mantener humanos licenciados en roles de decisión; citar fuentes; registrar recomendaciones.
Finance or insurance	Analizar documentos, clasificar riesgos, preparar decisiones.	Hacer cumplir policy, auditar el linaje de datos, requerir aprobación para acciones financieras.
Legal operations	Buscar, resumir, comparar, redactar, extraer obligaciones.	Preservar límites de privilegio; mostrar citas; evitar conclusiones legales no supervisadas.
Education	Tutoría, evaluación, adaptar dificultad, generar retroalimentación.	Rastrear learning goals; evitar afirmaciones no respaldadas y sobre-personalizadas.
Scientific research	Buscar literatura, proponer hipótesis, analizar datos.	Separar generación de hipótesis de validación; rastrear procedencia.
Software engineering	Buscar código, editar, probar, revisar, preparar PRs.	Ejecución en sandbox; preservar diffs; requerir CI y filtros de revisión.

Diferentes dominios pueden compartir patterns mientras requieren distintos controles de riesgo.

Concrete Domain Examples

Domain	Safe Agent Role	Hard Boundary
Support	Resumir historial de tickets, recuperar policy, redactar respuesta.	No puede prometer reembolsos, créditos o cambios de cuenta sin aprobación.

Domain	Safe Agent Role	Hard Boundary
Healthcare	Preparar resúmenes de visitas, preguntas de pacientes o paquetes de evidencia.	No puede diagnosticar, recetar ni reemplazar revisión licenciada.
Finance	Extraer campos de estados de cuenta, marcar anomalías, redactar notas de riesgo.	No puede mover dinero, aprobar crédito o cambiar derechos sin controles.
Legal operations	Comparar cláusulas, resumir historial de asuntos, redactar listas de obligaciones.	No puede emitir juicio legal no supervisado ni cruzar límites de privilegio.
Insurance	Clasificar documentos de reclamos, recuperar términos de policy, redactar recomendaciones.	No puede negar ni aprobar reclamos sin revisión gobernada.
Internal operations	Clasificar solicitudes de acceso, redactar runbooks, resumir incidentes.	No puede otorgar acceso a producción ni cerrar incidentes sin aceptación del responsable.

Estos ejemplos comparten un pattern: el agent prepara evidencia y recomendaciones; el sistema gobernado posee la autoridad.

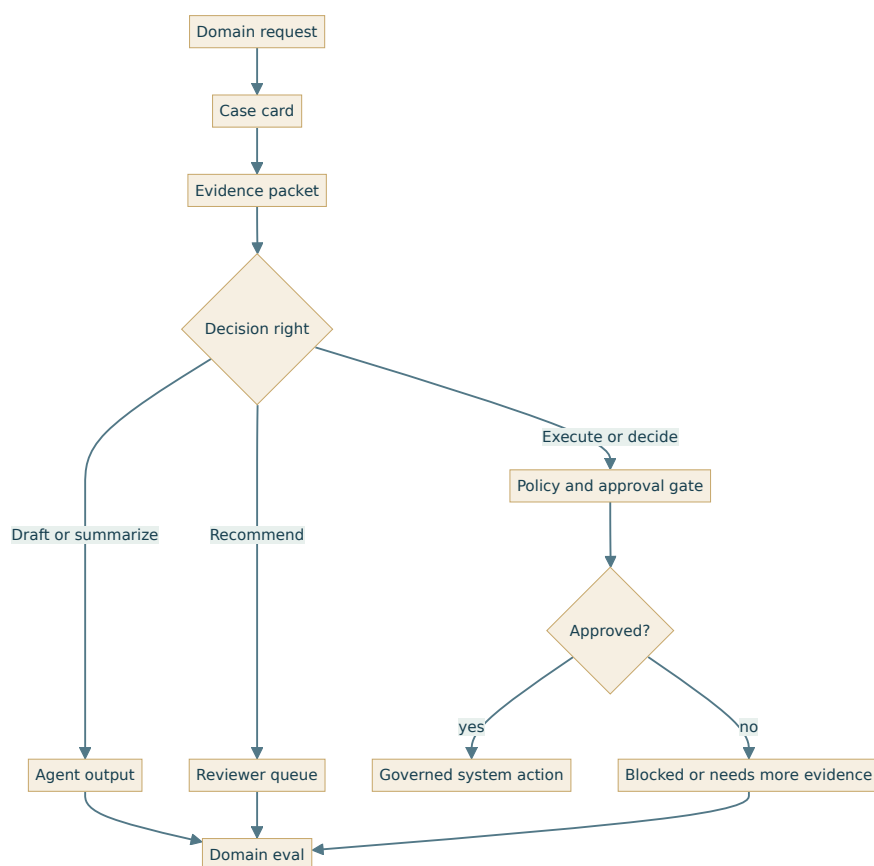
Production Case Cards

Usa case cards cuando un equipo de dominio diga: “Necesitamos un agent para nuestro proceso”. La card convierte esa solicitud en decisiones sobre evidencia, autoridad, revisión y eval.

Domain	Example Request	Agent May Produce	Must Block Or Escalate	Release Eval
Support	“Reembolsa este pedido y avisa al cliente.”	Resumen de ticket, coincidencia de policy, recomendación preliminar, borrador de mensaje al cliente.	Emisión de reembolso, promesa al cliente, crédito de cuenta o policy faltante.	Reembolsos por encima del umbral requieren aprobación antes de cualquier lenguaje de promesa.
Healthcare	“Resume esta visita y sugiere próximos pasos.”	Resumen de visita, preguntas del paciente, paquete de evidencia para revisión clínica.	Diagnóstico, plan de tratamiento, cambio de medicación o datos cruzados de pacientes.	Falta de revisión clínica bloquea salida con lenguaje de tratamiento.
Finance	“Explica por qué se marcó esta cuenta.”	Resumen de anomalía, transacciones fuente, borrador de nota de riesgo, motivo en cola de revisión.	Movimiento de dinero, aprobación de crédito, cambio de derechos o conclusión de riesgo no respaldada.	Jurisdicción incorrecta o policy obsoleta bloquea la recomendación.

Domain	Example Request	Agent May Produce	Must Block Or Escalate	Release Eval
Legal operations	“Compara estas cláusulas y aconseja cuál es más segura.”	Comparación de cláusulas, diferencias citadas, lista de obligaciones, preguntas de revisión.	Asesoría legal, comunicación externa, presentación o uso de fuentes cruzadas de asuntos.	Violación de privilegio o límites de asunto falla el eval.
Internal operations	“Otorga acceso de producción al ingeniero para el incidente.”	Resumen de solicitud de acceso, coincidencia de runbook, recomendación del responsable, propuesta de expiración.	Otorgar permisos, cierre de incidentes, comando destructivo o saltar al responsable.	Cambio de acceso requiere aprobación del responsable y expiración antes de ejecución.

Cada card debe nombrar una oración o acción prohibida. Para soporte, “Su reembolso ha sido aprobado” está prohibido antes de la aprobación. Para salud, “Debe comenzar este medicamento” está prohibido a menos que el workflow licenciado posea esa decisión. Para operaciones internas, “Acceso concedido” está prohibido hasta que pasen las verificaciones de identidad y responsable.



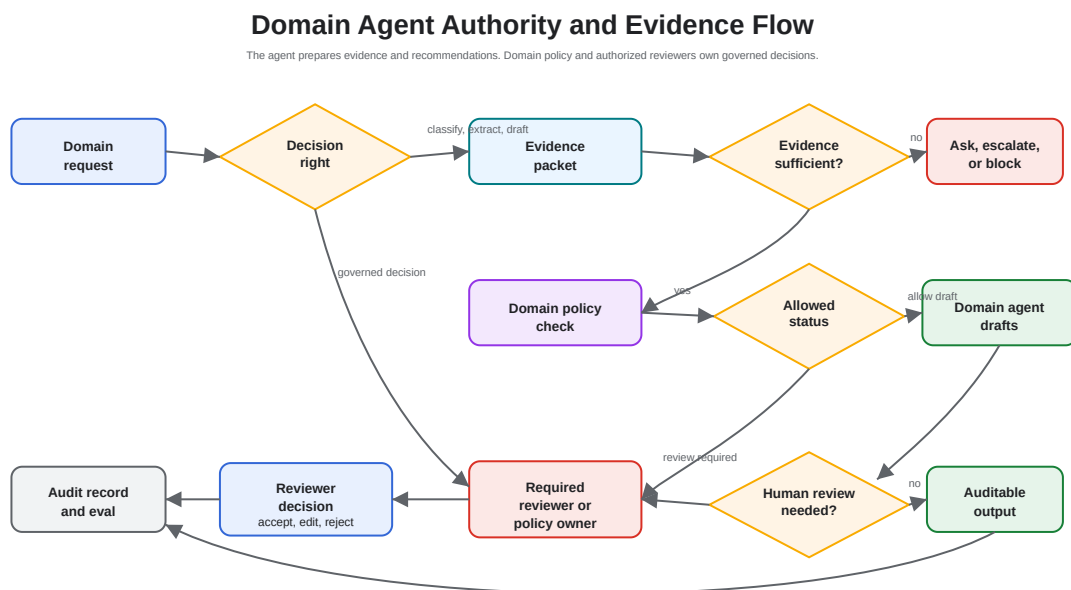
La case card no es documentación posterior. Es la entrada de la arquitectura. Si la card no puede nombrar la evidencia autorizada, acción prohibida, revisor y release eval, el domain agent no está listo para construirse.

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

El domain router debe seleccionar fuentes, tools y rutas de aprobación. No debe omitir la gobernanza.



Lee el diagrama de izquierda a derecha. La primera puerta son los decision rights: si la task pide al agent ejecutar, negar, aprobar, diagnosticar o decidir, el flujo avanza hacia policy o propiedad del revisor antes de que el model pueda producir un resultado gobernado.

Domain Policy Contract

La domain policy debe ser lo suficientemente ejecutable para probarse.

```
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[] };
};
```

El tipo exacto variará, pero el principio no: la domain policy es una decisión en runtime, no un párrafo en el prompt.

Diseño de Evidencia

Los domain agents necesitan una evidence policy explícita.

Define:

- tipos de fuentes autorizadas;
- requisitos de actualidad;
- formato de citación;
- manejo de conflictos entre fuentes;
- evidencia mínima para una respuesta;
- comportamiento ante afirmaciones no respaldadas;
- reglas de redacción de fuentes;
- límites de tenant y rol.

Si el agent no puede citar o explicar la base de una respuesta de dominio, debe reducir la confianza, pedir aclaración o escalar.

Domain Evidence Packet

Cada recomendación de dominio debe incluir un evidence packet. El packet debe ser lo suficientemente pequeño para que un revisor lo inspeccione y lo suficientemente estructurado para que los evals lo prueben.

```
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";
  };

```

El packet debe hacer evidente cualquier evidencia insegura. Una jurisdicción faltante, una policy version desactualizada, una fuente cross-tenant o una afirmación no respaldada deben cambiar el estado final antes de que el usuario vea una respuesta confiada.

Domain Playbooks

Usa domain playbooks para evitar tratar cada campo especializado como la misma arquitectura con diferente vocabulario.

Domain	Minimum Evidence	Default Agent Output	Mandatory Review Trigger
Support	ticket, account scope, current policy, prior actions	draft response o recommendation	refund, credit, account change o missing policy
Healthcare	patient-scoped record, approved reference, date, reviewer role	summary, question list o evidence packet	diagnosis, treatment, medication o conflicting clinical facts
Finance	transaction record, policy, jurisdiction, risk class	anomaly note, extracted fields o draft analysis	money movement, credit decision, entitlement change o high-risk flag
Legal operations	matter scope, source document, privilege boundary, clause span	clause summary, obligation list o draft comparison	legal conclusion, privilege risk, filing o external communication
Insurance	claim record, policy version, effective date, required documents	claim summary o draft recommendation	approval, denial, exception, missing document o conflicting evidence
Internal operations	request, owner, runbook, access policy, incident state	triage, draft runbook step o owner recommendation	production access, incident closure, permission grant o destructive action

El playbook debe ser local al dominio y propiedad del equipo responsable del resultado. No permitas que un agent prompt genérico decida cuánta evidencia es suficiente.

Worked Slice: Insurance Claim Triage

Un insurance claim triage agent puede ayudar a un revisor a avanzar más rápido sin darle autoridad sobre el claim.

Step	Runtime Owner	Agent Role	Required Evidence	Stop Condition
intake	workflow service	none	claim ID, policy ID, claimant, submitted documents	invalid o 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 o mismatched policy
triage	domain agent	draft coverage questions and risk notes	extracted fields and policy spans	conflicting evidence o exception
recommendation	domain agent	recommend next reviewer action	evidence packet and uncertainty list	approval, denial o exception requested
review	licensed or authorized reviewer	none	recommendation, source spans, policy checks	reviewer accepts, edits, rejects o escalates
audit	workflow service	none	final decision, reviewer ID, reason, trace	retained case record

El agent puede decir: “Este claim parece necesitar más evidencia porque la fecha del incidente está fuera del periodo de policy visible.” No debe decir: “Claim denied.” La arquitectura mantiene el lenguaje de triage separado del lenguaje de decisión regulada.

Domain Release Gate

Antes de que un domain agent llegue a usuarios o revisores reales, requiere evidencia de que el límite de dominio funciona.

Gate	Passing Evidence
decision rights	tests prueban que el agent redacta, recomienda o escala en vez de tomar decisiones prohibidas
source authority	evals fallan cuando la respuesta usa evidencia no aprobada, desactualizada, cross-tenant o de versión incorrecta

Gate	Passing Evidence
reviewer workflow	el revisor puede ver source spans, policy checks, incertidumbre y acción propuesta
privacy and retention	traces, prompts, memory y outputs redactan u omiten datos regulados según la policy
negative cases	el sistema escala evidencia faltante, evidencia conflictiva y policy exceptions
override learning	las ediciones y rechazos del revisor se convierten en regression cases etiquetados
communication safety	los borradores evitan lenguaje de decisión final, diagnóstico, promesa, obligación o aprobación

No consideres un puntaje alto en calidad de respuesta genérica como un domain release. El release gate debe probar que el sistema respeta el modelo de autoridad del dominio.

Jurisdicción, Tenant y Límites de Versión

Las respuestas de dominio suelen depender de dónde, cuándo y para quién se hace la pregunta.

Rastrea:

- límite de tenant o cliente;
- jurisdicción o región;
- policy version;
- document version;
- effective date;
- reviewer role;
- data retention class;
- source freshness.

Una finance policy de la jurisdicción incorrecta, una clinical note del paciente equivocado, un documento legal del matter incorrecto o una insurance policy de la effective date equivocada no es evidencia débil. Es evidencia insegura.

Revisión y Aprobación

Los domain agents suelen producir recomendaciones en lugar de decisiones finales.

Usa revisión humana para:

- decisiones clínicas, legales, financieras o de cumplimiento;
- clasificaciones de baja confianza;
- evidencia conflictiva;
- excepciones a la policy;
- acciones irreversibles;
- comunicaciones que generen obligación o responsabilidad.

La interfaz de revisión debe mostrar evidencia, policy checks, campos extraídos, incertidumbre y la acción propuesta.

Las decisiones del revisor deben convertirse en datos. Captura si el revisor aceptó, editó, rechazó o escaló la recomendación y por qué. Esos registros pueden actualizar evals, policy examples, prompts y material de entrenamiento.

Domain Evals

Los evals genéricos no son suficientes.

Agrega evals para:

- terminología de dominio;
- extracción de documentos;
- precisión de citación;
- evidencia faltante;
- policy exceptions;
- umbrales de escalamiento;
- restricciones de privacidad;
- entradas adversariales o ambiguas;
- casos de override del revisor.

Usa expertos en el tema para etiquetar donde el juicio sea importante.

Domain Eval Matrix

Eval Type	Example
Evidence grounding	la respuesta debe citar la policy actual o el source span
Extraction accuracy	los campos requeridos coinciden con los document fixtures etiquetados
Missing evidence	el sistema pide más evidencia o escala
Policy exception	la recomendación sigue el workflow de excepción
Privacy boundary	el agent rechaza datos cross-tenant o no autorizados
Jurisdiction/version	la respuesta usa la región y policy date correctas
Reviewer override	las recomendaciones sobreescritas se convierten en regression cases
Communication risk	los borradores evitan lenguaje de obligación, diagnóstico, promesa o decisión final

Los domain evals deben incluir casos negativos. El sistema debe probar que sabe cuándo no debe responder.

Failure Modes

- El agent suena autoritativo cuando la evidencia es débil.
- La domain-specific policy vive solo en prompts.

- El agent trata recomendaciones como decisiones.
- La retrieval mezcla tenants, jurisdicciones o policy versions.
- Los evals miden la calidad del lenguaje pero no la corrección de dominio.
- La revisión humana no tiene suficiente evidencia para decidir.
- El sistema usa una fuente web general donde solo se permiten fuentes de dominio aprobadas.
- Los reviewer overrides no se capturan, por lo que el mismo error se repite.
- Un mensaje borrador crea obligación legal, financiera, clínica o contractual.
- La memory almacena hechos regulados o sensibles sin reglas de retención y eliminación.

Preguntas de revisión

Antes de poner en marcha un agent de dominio, pregunta:

1. ¿Qué decisiones tiene prohibido tomar el agent?
2. ¿Qué fuentes de evidencia son autorizadas, obsoletas, prohibidas o insuficientes?
3. ¿Qué rol humano es responsable de la revisión final?
4. ¿Qué tenant, jurisdicción, policy o versión de documento aplica?
5. ¿Qué debe conservarse para auditoría?
6. ¿Qué debe redactarse de los prompts, traces, memory y output final?
7. ¿Qué eval falla el release si el agent suena confiado pero carece de evidencia?
8. ¿Cómo se convierten las correcciones del revisor en pruebas de regresión?

Capítulos relacionados

- [Agentic RAG Systems](#)
- [Policy Enforcement](#)
- [Human Approval Gates](#)
- [Evaluation-Driven Agent Development](#)
- [Knowledge-Bound Agents](#)

Architecture Decision Records for Agents

Los agentic systems cambian rápidamente. Los Architecture Decision Records mantienen explícitas las decisiones sobre model, memory, tool, policy y workflow para que los futuros responsables puedan entender por qué el sistema se comporta de esa manera.

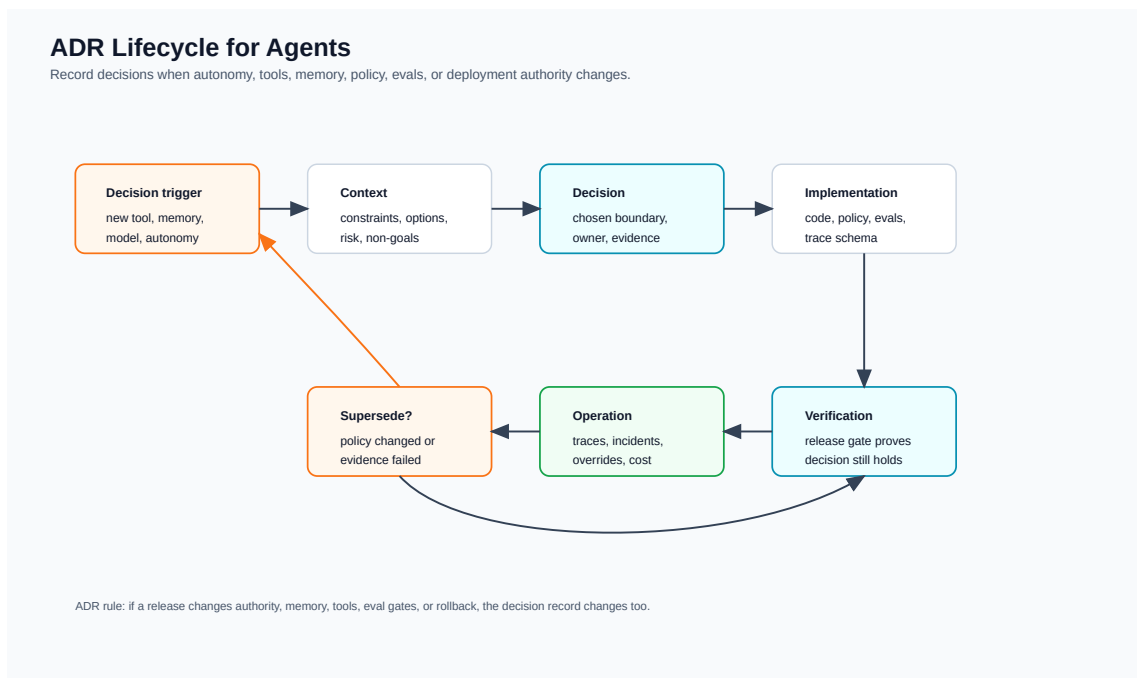
Usa ADRs cuando una decisión afecta la seguridad, el costo, la confiabilidad, la confianza del usuario o la capacidad de depurar el comportamiento en producción.

Un ADR no es burocracia. Es una forma de evitar que la autonomía se vuelva folclore. Si el sistema puede leer datos privados, invocar tools, escribir memory, enviar mensajes, delegar a otros agents o ejecutarse sin supervisión humana, el equipo debe poder señalar el registro de decisión que explica por qué eso está permitido.

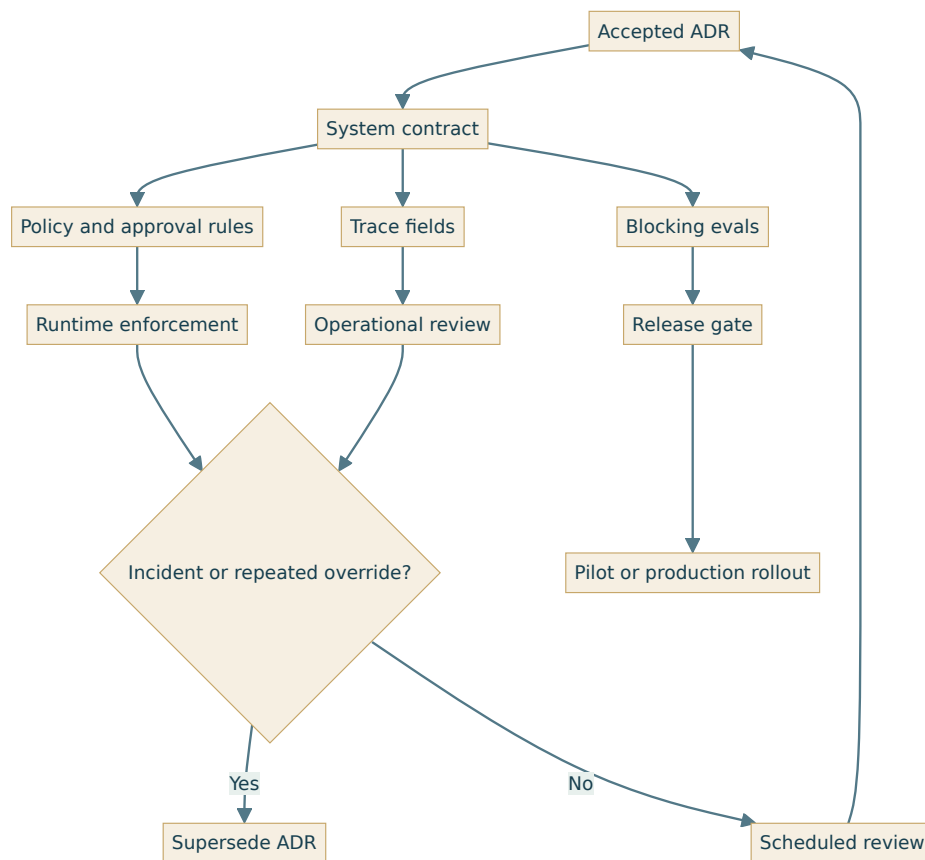
Descarga ejemplos completos: [agent ADR example pack](#).

Ciclo de Vida del ADR

Usa este diagrama para decidir cuándo una decisión de agent necesita un registro y cuándo un registro existente debe ser reemplazado. Cambios de autoridad, cambios en memory, cambios en tools, cambios en eval y cambios en rollback deben dejar evidencia.



Usa este flujo de evidencia después de que el ADR sea aceptado. Un registro de decisión debe producir verificaciones en runtime, no solo documentación.



Qué Registrar

Registra decisiones sobre:

- Selección de model y modelos de fallback
- Permisos de tool y reglas de aprobación
- Retención y eliminación de memory
- Fuentes de recuperación y policy de citación
- Reintentos de workflow, compensación y escalamiento
- Datasets de evaluación y puertas de lanzamiento
- Límites de observability y logging
- Requisitos de revisión humana
- Políticas de auto-mejora y actualización de habilidades

También registra decisiones que cambian el nivel de autonomía del agent:

- solo asesoría;
- borradores para revisión humana;
- ejecuta después de aprobación;
- ejecuta autónomamente dentro de una policy limitada;
- escala a otro agent, workflow o humano.

La autonomía no es una propiedad global. Un agent puede ser autónomo para investigación de solo lectura, requerir aprobación para reembolsos y tener prohibida la comunicación saliente. El ADR debe indicar qué acciones pertenecen a cada categoría.

Plantilla ADR

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?

Adiciones Específicas para Agents

Agrega estos campos cuando sean relevantes:

- **Nivel de autonomía:** asesoría, propone ediciones, ejecuta tras aprobación o ejecuta autónomamente.
- **Alcance de tool:** tools y operaciones permitidas exactamente.
- **Propietario del state:** historial de chat, state de workflow, memory store, base de datos o sistema externo.
- **Policy de fallas:** reintentar, re-planear, preguntar, rechazar, rollback o escalar.
- **Eval gate:** pruebas o datasets requeridos antes del lanzamiento.
- **Ruta de rollback:** cómo deshabilitar o revertir la decisión.

Decisiones Que Merecen ADRs

Escribe un ADR cuando el equipo:

- agrega un tool con capacidad de escritura;
- habilita comunicación saliente;
- habilita o cambia escrituras en memory;
- pasa de modo asesor a modo de ejecución;
- permite ejecución autónoma para un workflow;
- agrega una nueva fuente a RAG o cambia la policy de recuperación;
- cambia la familia de model, el enrutamiento, el fallback o la temperatura para un camino crítico;
- agrega delegación multi-agent o llamadas remotas a agents;
- cambia reglas de aprobación, roles de aprobadores o expiración de aprobaciones;
- cambia observability, retención, redacción o comportamiento de replay;
- permite ejecución de código, uso de navegador, acceso a shell o escrituras en el sistema de archivos.

Pequeños cambios en el wording del prompt no siempre necesitan ADRs. Los cambios de autoridad sí.

Ejemplos de Decisiones

- Usar un durable workflow para tareas que afectan al cliente en lugar de un loop en memoria.
- Requerir aprobación humana antes de enviar correo electrónico saliente.
- Almacenar episodic memory para eventos de proyectos pero no secretos personales.
- Usar recuperación híbrida por palabra clave más vector para documentación de soporte.
- Ejecutar coding agents en worktrees desechables y requerir `npm test` antes de hacer commit.
- Mantener la auto-mejora como cambios de habilidades revisados, no como mutación silenciosa del prompt.

Ejemplos de ADR Completados

Usa el [agent ADR example pack](#) cuando quieras registros completos para:

- autoridad de reembolso en soporte, donde el agent puede redactar pero no emitir dinero;
- policy de fuentes RAG para investigación, donde el agent solo puede responder desde fuentes aprobadas;
- revisión de entregas multi-agent, donde especialistas trabajan en paralelo pero un solo responsable acepta el resultado.

El primer ejemplo se incluye a continuación porque muestra el patrón más importante: separar la recomendación del model de la autoridad de negocio.

Ejemplo de ADR: Autoridad de Reembolso en Soporte

```
# 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.

Modos de falla

- Las decisiones solo viven en prompts y desaparecen de la revisión de ingeniería.
- Los ADR describen el happy path pero no rollback ni verificación.
- Las actualizaciones de model ocurren sin registrar el impacto en eval.
- Cambios en memory o en el alcance de tool se lanzan sin revisión de privacidad.
- El ADR dice “aprobación humana” pero no especifica qué acciones la requieren.
- El ADR dice “agentic” pero no define el nivel de autonomía.
- La autoridad de tool se describe como una categoría en vez de herramientas y operaciones exactas.
- Se habilita memory sin reglas de retención, corrección, eliminación o consentimiento.
- Se listan eval gates pero no se vinculan al release o rollback.
- Nadie es responsable de la decisión después del lanzamiento.

Guía de verificación

Un ADR debe crear controles operativos, no solo documentación.

Para cada ADR aceptado, define:

- el eval suite que protege la decisión;
- los traces y dashboards que muestran que se sigue la decisión;
- las señales de incident que invalidarían la decisión;
- el owner que revisa esas señales;
- el camino de rollback o deshabilitación;
- la fecha o trigger para revisión.

Triggers útiles de revisión incluyen actualización de model, reescritura de prompt, cambio en el manifiesto de tool, nuevo tipo de memory, nueva fuente de datos, nuevo tenant, cambio en la policy de aprobación, incident de alta severidad o repetidos overrides humanos.

Lista de verificación para producción

- ¿El ADR nombra el agent, workflow, owner y alcance?
- ¿Define el nivel de autonomía por acción?
- ¿Enumera herramientas exactas, efectos secundarios, alcances, credenciales y egress?
- ¿Define límites de datos, memory, retención, eliminación y corrección?
- ¿Indica qué acciones requieren aprobación y quién puede aprobar?
- ¿Define evals bloqueantes y release gates?
- ¿Define traces, métricas, registros de auditoría y reglas de redacción?
- ¿Define rollback, kill switch o deshabilitación de capability?
- ¿Incluye riesgos residuales y triggers de revisión?
- ¿El ADR está vinculado desde el código, runbook o checklist de despliegue?

Capítulos relacionados

- [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

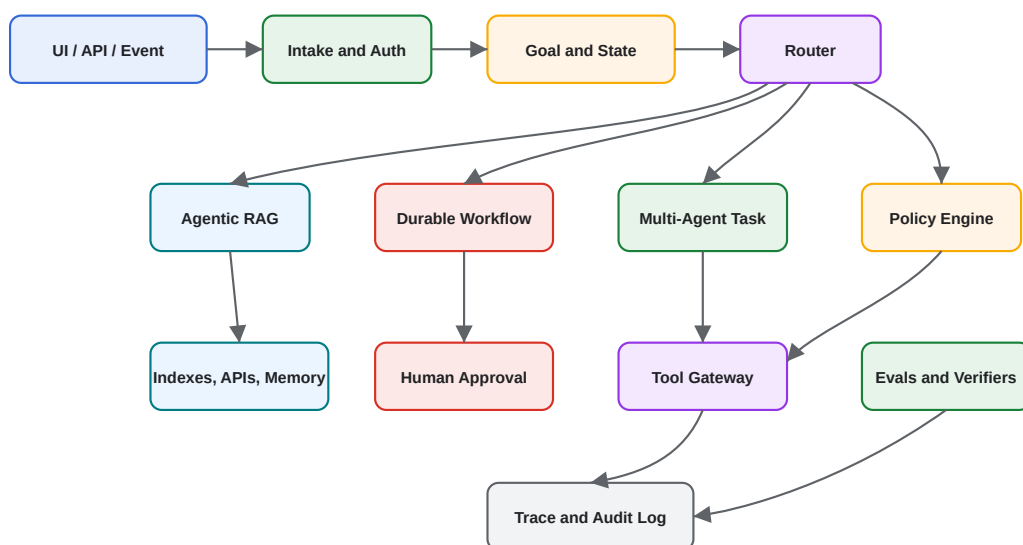
Esta arquitectura de referencia combina los patrones principales del libro en un agentic system listo para producción. Es intencionalmente conservadora: el software determinista es dueño del state, la policy y los side effects; las llamadas al model proponen decisiones dentro de loops acotados.

Usa este capítulo como puente entre la selección de patterns y el diseño del sistema. El diagrama no es una prescripción de framework; muestra los límites de propiedad que un sistema en producción necesita antes de que los agents manejen datos privados, acciones externas o trabajo de larga duración.

Architecture

Lee este diagrama como un mapa de propiedad para sistemas en producción: los servicios deterministas son dueños de la identidad, el state, la policy, la retrieval, las aprobaciones y los traces alrededor de llamadas acotadas al model.

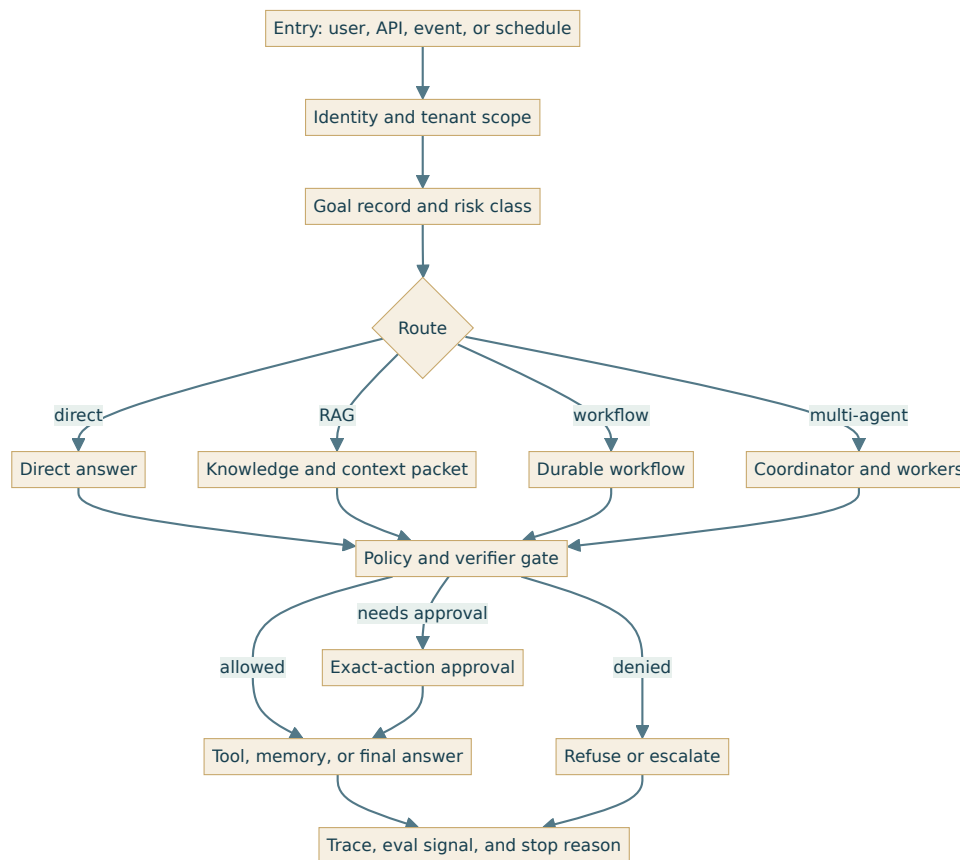
Production Agentic System Reference Architecture



Descarga el artifact reutilizable de revisión: [reference architecture review checklist](#).

Boundary Flow

Usa este flujo para revisar una solicitud desde la entrada hasta el estado final. Separa el control plane que decide qué está permitido del execution plane que recupera evidencia, invoca tools y escribe el state.



El flujo debe fallar cerrado. Una propuesta del model solo puede avanzar después de que los checks de state, policy, schema, evidencia, aprobación o verificador la acepten.

What This Architecture Is For

Usa esta estructura cuando el agent pueda tocar datos privados, invocar tools, esperar aprobación, ejecutarse por más de una solicitud o afectar clientes, dinero, infraestructura, cumplimiento o operaciones internas. Es más pesada que un prompt chain, pero la estructura adicional permite revisiones.

No comiences aquí para un prototipo de solo lectura, un resumidor de una sola vez o un workflow cuyos pasos sean totalmente deterministas. Para esos sistemas, usa el pattern más pequeño que resuelva el task y mantén los controles de producción proporcionales al riesgo.

Readiness Questions

Usa estas preguntas antes de una revisión de diseño:

1. ¿Qué puede hacer el sistema que una respuesta normal de chat no puede?
2. ¿Qué acción crea el mayor riesgo, costo o impacto al usuario?
3. ¿Qué servicio es dueño del state que sobrevive reintentos, caídas y esperas de aprobación?
4. ¿Qué policy check se ejecuta antes de la retrieval, antes del uso de tools, antes de escribir en memory y antes de la respuesta final?
5. ¿Qué evidencia debe ver el model y cuál debe permanecer fuera del context del model?
6. ¿Qué evento exacto mueve una ejecución de borrador a acción aprobada?
7. ¿Qué trace prueba que el sistema siguió el camino previsto?

8. ¿Qué switch desactiva el comportamiento riesgoso sin deshabilitar todo el producto?

Si un equipo responde esto solo con texto de prompt, el diseño sigue siendo un prototipo.

Layered View

Layer	Owns	Must Not Delegate To The Model
Entry layer	Usuario, tenant, fuente de eventos, validación de solicitudes, límites de tasa.	Autenticación, alcance de tenant, confianza en entrada sin procesar.
Routing layer	Respuesta directa, workflow, loop, RAG o ruta multi-agent.	Si una ruta de alto riesgo está permitida.
Goal and state layer	Registro de goal, run state, checkpoints, motivo de detención, datos de replay.	Reglas de mutación de state durable.
Context and knowledge layer	Retrieval, memory, frescura, citas, policy de fuente.	Elegibilidad de fuente o control de acceso.
Tool and execution layer	Schemas de tools, timeouts, idempotencia, registros de side-effect.	Permiso para llamar sistemas externos.
Policy and approval layer	Autorización, clase de riesgo, aprobación exacta de acción, escalamiento.	Autoridad final para acciones riesgosas.
Evaluation layer	Evals offline, runtime verifiers, release gates, datos de regresión.	Juicio de calidad o seguridad sin umbrales.
Observability layer	Traces, audit logs, costos, latencia, llamadas a tools, incidentes.	Razonamiento oculto como única explicación.

Las capas no requieren servicios separados. Un sistema pequeño puede implementar varias capas en una sola base de código. Lo importante es la propiedad: otro ingeniero debe poder encontrar dónde se toma, prueba, registra y revierte cada decisión.

Control Plane vs Execution Plane

Mantén las decisiones de control fuera del texto libre del model. El model puede proponer, resumir, clasificar o redactar. Los servicios controlados por el runtime deciden si la propuesta puede afectar el state, tools, memory, aprobaciones o usuarios.

Plane	Owns	Examples	Release Evidence
Control plane	authorization, routing, policy, budgets, approval, release gates	route policy, risk class, policy decision, eval threshold, kill switch	policy logs, eval report, approval trace, rollback drill

Plane	Owns	Examples	Release Evidence
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

Esta separación previene una falla común: que un model proponga una acción y la autorice implícitamente. La arquitectura de producción debe hacer visible y comprobable el camino de autorización.

Maturity Ladder

Adopta la arquitectura por etapas. No construyas todos los componentes de plataforma antes de que el sistema demuestre valor.

Level	System Shape	Required Controls	Promotion Test
0 - Prototype	Demo local o interna de solo lectura.	Etiqueta clara de no producción, trace básico, sin datos privados ni side effects.	El equipo puede explicar qué fallaría antes de usar datos reales.
1 - Scoped Assistant	Un task acotado con retrieval o tools de solo lectura.	Identidad, alcance de tenant, policy de fuente, verificación de citas, motivo de detención.	Evidencia no autorizada o faltante produce denegación, rechazo o escalamiento.
2 - Tool-Assisted Workflow	El sistema redacta o prepara acciones mediante tools tipados.	Tool manifest, validación de schema, idempotencia para borradores, correlación de traces.	Tools con capacidad de escritura no autorizada no pueden ser llamados solo desde texto del model.
3 - Approved Action	El sistema puede activar efectos visibles para clientes, financieros, operativos o durables tras aprobación.	Aprobación exacta de acción, registro de decisión de policy, audit record, ruta de rollback o desactivación.	Una acción riesgosa no puede ejecutarse sin evidencia de policy y aprobación.
4 - Production Runtime	El sistema corre continuamente con eval gates y respuesta a incidentes.	State durable, replay, dashboards, alertas, release gates, runbook, rollback drill.	Los operadores pueden reconstruir y desactivar una ruta fallida sin reimplementar todo.
5 - Platform Capability	Múltiples productos o equipos reutilizan el runtime, tools, policies o evals.	Contratos versionados, pruebas de compatibilidad, matriz de propiedad, aislamiento de tenant, observabilidad compartida.	Un cambio de contrato no puede romper silenciosamente el comportamiento downstream de un agent.

La escalera no es una insignia de estatus. Es un control de alcance. Un sistema nivel 1 puede ser excelente para un task de solo lectura. Un sistema nivel 3 que no puede probar la aprobación vinculante no está listo para producción.

Primer Camino de Implementación

Comienza con un solo workflow de usuario y un solo límite de riesgo. Una primera implementación útil debe ser lo suficientemente pequeña como para revisarse en una sola reunión de diseño.

Paso	Construir Primero	Posponer Hasta Que Sea Necesario
Goal	Una task con criterios explícitos de éxito y rechazo.	Alcance amplio de assistant o enrutamiento multi-dominio.
State	Run ID, actor, tenant, route, stop reason y trace link.	Long-term memory, personalización entre sesiones o auto-mejora del agent.
Knowledge	Una fuente aprobada con reglas de vigencia y citación.	Múltiples índices, ingestión automática o recuperación abierta en la web.
Tools	Un tool de solo lectura o un tool solo para borradores.	Write tools, control de navegador, acceso a shell o acciones de pago.
Policy	Denegar rutas inseguras, fuentes no autorizadas y evidencia faltante.	Plataforma de policy completa con reglas dinámicas.
Evaluation	Cinco casos: éxito, evidencia faltante, fuente no autorizada, falla de tool y rechazo.	Suites de eval basadas en judge y generación de datos sintéticos.
Operations	Muestra de trace, owner, límites conocidos y switch de desactivación.	Suite completa de dashboards y rotación multi-equipo de on-call.

Este camino mantiene la arquitectura ligada a la evidencia. Cada capa agregada debe cerrar un riesgo nombrado, no solo satisfacer un diagrama.

Flujo de Solicitud

1. Autentica al usuario o fuente del evento.
2. Crea un registro de goal con restricciones, alcance de usuario y resultado solicitado.
3. Enruta la task a una respuesta directa, Agentic RAG, durable workflow o proceso multi-agent.
4. Ejecuta verificaciones de policy antes de la recuperación, uso de tools o efectos secundarios.
5. Persiste observaciones, decisiones, llamadas a tools y outputs.
6. Verifica evidencia, calidad del output y cumplimiento de policy.
7. Devuelve una respuesta, solicita aprobación, rechaza o escala.
8. Almacena traces para depuración y mejora del dataset de eval.

Ejemplo: Solicitud de Reembolso en Soporte

Un cliente solicita un reembolso. La arquitectura debe mantener tres decisiones separadas.

Decisión	Owner	Ejemplo de Resultado
¿Puede el sistema inspeccionar esta orden?	Capas de entrada, identidad y policy.	Permitir lectura dentro del mismo tenant o denegar acceso cruzado entre tenants.
¿Se cumple la refund policy?	Capas de retrieval, policy y eval.	Borrador de recomendación con cita de policy o escalar por evidencia faltante.
¿Puede moverse el dinero?	Capas de aprobación y ejecución.	Requiere aprobación de finanzas; el model no puede emitir el reembolso.

El model puede redactar una recomendación. No autentica al usuario, no elige sus propios tools, no decide la autoridad de pago, no escribe memory sin policy, ni marca el workflow como completo sin un stop reason. Esa separación es el objetivo de la reference architecture.

Mapeo de Puntos de Control

Convierte la arquitectura en tickets de implementación asignando cada transición riesgosa a un punto de control concreto.

Transición	Control Requerido	Evidencia a Almacenar	Prueba de Liberación
request entra al sistema	identidad, alcance de tenant, rate limit	actor ID, tenant ID, source, request ID	request entre tenants denegada antes de retrieval
route seleccionada	route policy y risk class	route, risk class, motivo de enrutamiento	route de alto riesgo no puede seleccionarse solo por texto de prompt
context ensamblado	elegibilidad de fuente, vigencia, presupuesto	context refs, owner de fuente, vigencia, token count	fuentes obsoletas o no autorizadas excluidas
model propone acción	validación de schema y policy	propuesta, resultado de schema, resultado de policy	propuesta mal formada rechazada
llamada a tool solicitada	tool manifest, permiso, idempotency	nombre del tool, autoridad, argumentos, idempotency key	escritura no autorizada denegada
aprobación necesaria	exact-action binding	approver, acción, monto o destino, expiración, trace ID	aprobación amplia rechazada
resultado emitido	verificador final y stop reason	respuesta, citas, estado final, stop reason	evidencia faltante no puede producir estado completado

Transición	Control Requerido	Evidencia a Almacenar	Prueba de Liberación
memory escrita	memory class y regla de retención	tipo de memory, owner, retención, ruta de eliminación	memory sensible denegada o redactada

Si una fila no tiene owner, prueba o evidencia almacenada, la arquitectura sigue siendo solo un dibujo. La implementación está lista solo cuando cada transición puede fallar de forma cerrada.

Puntos de Control

- **Antes de retrieval:** aplica control de acceso y elegibilidad de fuente.
- **Antes de usar tools:** valida schema, permisos, presupuesto y necesidades de aprobación.
- **Antes de la respuesta final:** verifica afirmaciones, citas y policy.
- **Antes de escribir memory:** clasifica qué tipo de memory se almacena.
- **Antes de liberar:** ejecuta evals y revisiones de regresión.

Slices de Implementación de Referencia

Construye la arquitectura en slices. Cada slice debe producir una prueba desplegable, no solo una promesa en un diagrama.

Slice	Construir	Debe Probar
Respuesta solo lectura	validación de entrada, enrutamiento, context packet, trace	El sistema puede responder con evidencia delimitada y un trace reproducible.
Respuesta asistida por tool	gateway de tools, schemas, timeouts, idempotency	Las llamadas a tools están autorizadas, tipadas, registradas y son reversibles cuando sea posible.
Approval workflow	solicitud de aprobación, exact action binding, expiración, audit log	Un humano aprueba una acción concreta, no un comportamiento futuro amplio.
Durable workflow	state store, checkpoints, reintentos, stop reasons	La ejecución sobrevive a fallas sin duplicar efectos secundarios.
Production release	suite de eval, runbook, rollback switch, dashboards	Los operadores pueden detectar, detener, depurar y mejorar el sistema.

Esta secuencia evita que los equipos construyan orquestación antes de poder probar límites de state, policy y evidencia.

Formas de Despliegue

La reference architecture no requiere una plataforma grande desde el primer día. Usa la forma de despliegue más pequeña que preserve los puntos de control para el riesgo que estás asumiendo.

Forma	Usar Cuando	Compresión Aceptable	No Comprimir
Prototipo	El sistema es solo lectura, interno y desechable.	Un solo servicio de aplicación puede manejar routing, state, retrieval y traces.	Autenticación, alcance de tenant, elegibilidad de fuente y stop reason.
Piloto	Un grupo pequeño usa el sistema en trabajo real con autoridad limitada.	Las verificaciones de policy y registros de aprobación pueden vivir en la base de datos de la aplicación.	Permiso de tool, correlación de trace, puertas de eval y rollback switch.
Producción	El sistema afecta usuarios, registros, dinero, infraestructura u operaciones.	Varias capas pueden compartir repositorio o runtime.	Durable state, idempotency, audit logs, ownership de incidentes y evidencia de liberación.
Regulado o alto riesgo	El sistema toca compliance, datos sensibles, pagos, seguridad, salud, decisiones legales o laborales.	Las llamadas al model pueden quedarse detrás de un solo runtime gateway.	Policy enforcement independiente, auditoría de aprobación, redacción, retención y revisión de evidencia.

La compresión es segura solo cuando la propiedad se mantiene explícita. Un prototipo puede compartir rutas de código. No puede compartir la pregunta de quién es owner de state, policy, autoridad de tool, evidencia y rollback.

Guía de Ubicación de Límites

Usa esta guía para decidir si un componente pertenece dentro del agent runtime, el servicio de aplicación o un servicio de plataforma separado.

Límite	Mantener Dentro de la App Cuando	Separar en un Servicio Cuando
Policy	Las reglas son pocas, específicas del producto y revisadas con la feature.	Varios productos, tenants, risk classes o owners de compliance usan las mismas reglas.
Tools	Los tools son acotados, de bajo riesgo y propiedad de un solo equipo.	Los tools son compartidos, con capacidad de escritura, costosos, con credenciales o auditados.
Retrieval	Las fuentes son pequeñas, locales y de baja sensibilidad.	Las fuentes requieren control de acceso, policy de vigencia, citas, eliminación o revisión de owner.
Memory	La memory es local a la task o de corta duración.	La memory cruza sesiones, usuarios, tenants, clases de privacidad o reglas de retención.
Workflow	Las ejecuciones terminan en una sola request y no tienen efectos secundarios.	Las ejecuciones esperan aprobación, reintentos, reanudan, programan trabajo o llaman tools irreversibles.

Límite	Mantener Dentro de la App Cuando	Separar en un Servicio Cuando
Evaluation	La feature tiene un set de eval pequeño y fijo.	Los evals bloquean releases, comparan versiones, alimentan revisión de incidentes o sirven a varios equipos.
Observability	Los logs son suficientes para depurar una ejecución de prueba.	Los operadores necesitan trases, redacción, costo, latencia, tool, policy y registros de aprobación.

La separación debe seguir la propiedad y el riesgo, no la moda. Un servicio separado sin un owner más fuerte es solo código más distribuido.

Evidence Bundle

Un diseño que afirma seguir esta arquitectura debe adjuntar estos artifacts:

Artifact	Demuestra
Architecture diagram	Propiedad de state, tools, policy, memory, evals, approvals y trases.
State schema	Qué persiste, qué se puede reproducir y por qué se detuvo el run.
Tool manifest	Qué capabilities existen, quién puede llamarlas y qué efectos secundarios crean.
Context packet example	Qué fuentes llegaron al model, con confianza, frescura y presupuesto.
Policy decision log	Por qué se permitió o denegó una tool, retrieval source, memory write o approval path.
Trace sample	Qué ocurrió en un run exitoso y en un run fallido.
Eval report	Qué comportamientos bloquean el release.
Runbook	Quién es responsable de incidentes y cómo funciona el rollback.

Run Trace Contract

Una reference architecture solo es real si un run puede demostrar que los límites funcionaron. Captura el run como trace data estructurado, no como logs dispersos.

```
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;
};

```

Los nombres exactos de los campos pueden cambiar. Lo que no debe cambiar es el invariante: un revisor debe poder reconstruir quién preguntó, qué evidencia entró al context, qué propuso el model, qué policies se ejecutaron, qué tools se usaron, por qué se detuvo el run y cómo revertir o deshabilitar el camino riesgoso.

Architecture Acceptance Tests

Antes de un piloto, ejecuta al menos estas verificaciones de aceptación contra el trace contract.

Test	Evidencia de aprobación
Tenant isolation	Una solicitud cross-tenant registra una denegación antes de retrieval o uso de tool.
Missing evidence	El run se detiene con <code>evidence_missing</code> o <code>needs_approval</code> ; no inventa soporte.
Policy denial	Una acción bloqueada registra la versión de policy, boundary, motivo de denegación y sin efecto secundario.

Test	Evidencia de aprobación
Approval binding	Una acción de alto riesgo espera aprobación vinculada a la acción exacta, actor, expiración y trace ID.
Retry safety	Un write reintentado usa la misma idempotency key o se detiene antes de efectos secundarios duplicados.
Memory control	Un memory write registra clasificación, propietario, retención y ruta de eliminación.
Rollback proof	Los operadores pueden deshabilitar la ruta del model, tool, workflow, policy o agent sin eliminar todo el producto.

Si estas pruebas son difíciles de escribir, probablemente la arquitectura aún está oculta en framework glue, prompts o procesos informales.

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

Asigna responsables antes de producción. La falta de ownership es un defecto de arquitectura.

Component	Engineering Owner	Operational Owner	Release Artifact
Identity and tenant boundary	plataforma o backend de producto	seguridad o plataforma on-call	access-control tests
Goal and state store	equipo de workflow/runtime	runtime on-call	state schema y replay fixture
Retrieval router	equipo de search o knowledge	propietario de contenido/datos	source policy y freshness report
Tool gateway	equipo de integración	propietario de servicio para cada tool	tool manifest y audit sample

Component	Engineering Owner	Operational Owner	Release Artifact
Policy engine	seguridad, compliance o product policy	policy owner	policy decision log
Approval service	equipo de workflow de producto	propietario de aprobación de negocio	approval trace sample
Evals service	equipo de AI platform o calidad	release owner	blocking eval report
Trace and audit store	equipo de observability	incident commander	successful y failed run traces

Una persona puede ser responsable de varias celdas en un equipo pequeño. La tabla sigue siendo importante porque cada falla en producción preguntará quién era responsable del boundary.

Design Review Questions

Haz estas preguntas antes de un piloto en producción:

1. ¿Qué decisión toma el model y qué decisión toma el software después de eso?
2. ¿Qué cambios de state sobreviven a un crash, retry o espera de approval?
3. ¿Qué llamada de tool tiene el mayor costo, riesgo o irreversibilidad?
4. ¿Qué evidencia se requiere antes de que el sistema responda o actúe?
5. ¿Qué sucede cuando la evidencia falta, está desactualizada, es conflictiva o no está autorizada?
6. ¿Qué se escribe en memory y quién puede corregirlo o eliminarlo?
7. ¿Qué eval bloquea el release si el sistema falla mañana?
8. ¿Puede un operador reconstruir el run fallido sin adivinar?
9. ¿Qué se puede deshabilitar sin volver a desplegar todo el sistema?
10. ¿Qué haría que el equipo apague el agent?

Minimum Production Checklist

- Responsable explícito para goal y state
- Tool schemas y validación
- Aprobación humana para acciones de alto riesgo
- Retrieval con control de acceso
- Validación de citas para respuestas fundamentadas
- Evals para core tasks
- Traces para cada run
- Controles de presupuesto, timeout y cancelación
- Ruta de revisión de incidentes
- Switch de rollback o deshabilitar

Reference Architecture Review Gate

Usa la [reference architecture review checklist](#) antes de un piloto o release de producción. Bloquea el release cuando:

- el model es responsable de autenticación, autorización, policy o permiso de efectos secundarios;
- el durable state no tiene schema, replay path o motivo de detención;
- las retrieval sources carecen de control de acceso, confianza, frescura o validación de citas;
- las approvals autorizan comportamientos futuros vagos;
- los evals solo inspeccionan el texto final e ignoran tools, policy, evidence y transiciones de state;
- los operadores no pueden reconstruir un run exitoso y uno fallido a partir de traces.

Scaling Path

Comienza pequeño:

1. Single agent con validación de tool.
2. Agrega goals y state.
3. Agrega retrieval y validación de citas.
4. Agrega durable workflow para tasks de larga duración.
5. Agrega gates de aprobación humana.
6. Agrega datasets de eval y observability.
7. Agrega multi-agent decomposition solo cuando un agent se convierte en un cuello de botella.

Anti-Patterns

Evita estos diseños:

- El model elige de una lista amplia de tools sin permiso a nivel de ruta.
- El texto del prompt es el único boundary de policy.
- Retrieval, memory y resultados de tool se mezclan en el context sin procedencia.
- El sistema puede actuar pero no puede reproducir por qué actuó.
- Las approvals autorizan comportamientos futuros vagos en lugar de una acción exacta.
- Los evals solo revisan la calidad de la respuesta final e ignoran el uso de tools, policy, evidence y motivo de detención.
- El rollback requiere eliminar todo el deployment porque prompts, tools y policies no pueden deshabilitarse de forma independiente.

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

Los agentic systems en producción necesitan más que prompts, tools y un buen model. Necesitan un runtime que controle el límite de ejecución.

La regla central es la misma en todo este libro: el model propone, el runtime decide. El model puede proponer un plan, una llamada a un tool, una escritura en memory, una consulta de retrieval, una respuesta o una condición de detención. El runtime decide si esa propuesta es válida, permitida, asequible, observable, durable y segura de ejecutar.

Aquí es donde muchos proyectos de agent fallan. Tratan la llamada al model como si fuera el sistema, y luego agregan logging, reintentos, aprobaciones, evals y rollback después del primer incidente. Un runtime de producción invierte ese orden. Le da al agent un entorno controlado para operar.

Este capítulo controla la ejecución a lo largo de los runs: admisión, state, policy, presupuestos, ejecución de tools, recuperación, rollout y visibilidad para el operador. No controla la UX del producto, el contenido de policy de dominio, el entrenamiento del model ni la implementación interna de tools y servicios.

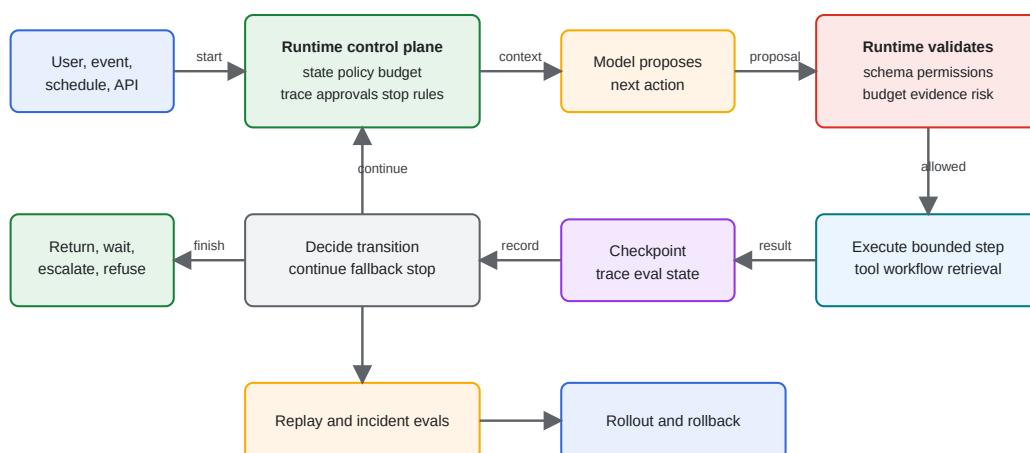
Lee esto después de [Agent Harnesses](#). El harness explica el entorno de trabajo alrededor de un agent; este capítulo explica lo que debe cumplirse cuando ese trabajo se ejecuta como un sistema de producción con colas, presupuestos, reintentos, state, rollout, rollback y operadores.

Para una guía paso a paso del laboratorio al lanzamiento, usa [Deployment Walkthrough](#). Para la configuración de framework y plantillas de decisión, usa [Real Framework Setup Notes](#) y [Templates and Worksheets](#).

Descarga la hoja de trabajo operativa: [runtime SLO and incident review worksheet](#).

Usa este diagrama como un mapa del control-plane. Cada recuadro nombra una responsabilidad del runtime que debe tener un responsable, un registro de trace y una ruta de rollback antes del tráfico en producción.

Production Runtime Overview



Qué controla el Runtime

El runtime de producción es el control plane alrededor del juicio del model.

Runtime Concern	Qué controla
State	Goal, estado del run, paso del workflow, intentos, resultados de tools, aprobaciones, motivo de detención.
Policy	Permisos, clasificación de riesgos, denegación, aprobación, escalamiento, requisitos de auditoría.
Budgets	Tokens, llamadas al model, llamadas a tools, reintentos, delegaciones, tiempo de reloj, costo.
Tools	Schemas, permisos, timeouts, idempotencia, registros de side-effect.
Memory y retrieval	Elegibilidad de fuente, frescura, control de acceso, referencias de evidencia, escrituras en memory.
Observability	Trace IDs, spans, costos, latencia, eventos de model y tool, metadatos de replay.
Evaluation	Chequeos en runtime, fixtures de incidentes, gates de lanzamiento, datasets de regresión.
Recovery	Reintentos, fallbacks, circuit breakers, checkpoints, compensación, rollback.

El runtime no elimina la autonomía. Hace que la autonomía sea lo suficientemente acotada como para confiar en ella.

Matriz de Responsabilidad del Runtime

El runtime debe hacer explícita la propiedad de cada aspecto. Si nadie es responsable de alguna de estas filas, el model eventualmente lo será por accidente.

Responsabilidad	Decisión del runtime	Falla si falta
Run admission	¿Debe esta solicitud iniciar, esperar, rechazarse o redirigirse?	Trabajo inseguro o no soportado entra al loop.
State ownership	¿Qué state es durable, temporal, visible o eliminado?	Progreso perdido, memory oculta, fallas no reproducibles.
Execution mode	¿Debe el task ejecutarse de forma síncrona, asíncrona o como un durable workflow?	Tiempos de espera, runs zombies, usuarios bloqueados, efectos secundarios parciales.
Proposal validation	¿Es válida la propuesta del model para schema, policy, budget y state?	Se ejecutan llamadas a tools porque el model parecía confiado.

Responsabilidad	Decisión del runtime	Falla si falta
Tool execution	¿Qué credenciales, timeout, clave de idempotencia y registro de side-effect aplican?	Escrituras duplicadas, credenciales filtradas, propiedad poco clara.
Approval state	¿Qué espera revisión humana, quién lo aprobó y qué acción exacta fue aprobada?	La aprobación se convierte en un permiso amplio en vez de un gate acotado.
Cancellation	¿Qué se detiene de inmediato, qué se drena y qué debe ser compensado?	Los runs cancelados siguen gastando o continúan efectos secundarios.
Rollout	¿Qué model, prompt, policy, tool schema, retriever o versión de harness está activa?	Las regresiones se lanzan globalmente sin rollback rápido.
Operations	¿A quién se alerta, qué se desactiva y qué evidencia se preserva?	Los incidentes se convierten en arqueología manual.

El Runtime Loop

Un runtime loop de producción no es solo observar, decidir, actuar. Se parece más a:

1. recibir una solicitud, evento, programación, webhook o comando de workflow;
2. autenticar al solicitante y cargar la clase de task, clase de riesgo, state, policy, budget y trace ID;
3. armar el conjunto de trabajo: goal, restricciones, evidencia, tools permitidos, memory y reglas de detención;
4. pedir al model o a un router determinista una propuesta acotada;
5. validar la propuesta contra schema, policy, budget, state, evidencia y reglas de aprobación;
6. ejecutar un paso acotado a través de un tool, workflow, retrieval service, evaluator o gate de aprobación;
7. hacer checkpoint del state, eventos de trace, costo, latencia, side effects y motivo de detención;
8. decidir si continuar, fallback, esperar, escalar, rechazar, compensar o completar;
9. convertir fallas importantes y casi fallas en casos de eval.

Este loop es lo que separa un producto agentic de un demo. El model aún puede ser creativo y adaptativo, pero el sistema sabe qué pasó y por qué.

Límites del Runtime

Un buen runtime crea límites explícitos:

- **Decision boundary:** el model puede sugerir acciones, pero el software las valida.
- **Authority boundary:** los side effects requieren tool schemas, permisos, budgets y reglas de aprobación.
- **State boundary:** el durable workflow state está separado del model context.
- **Context boundary:** el model ve un conjunto de trabajo seleccionado, no todos los documentos o memory disponibles.

- **Cost boundary:** cada loop, tool, reintento, llamada al model y delegación gasta del budget.
- **Policy boundary:** la denegación y el escalamiento son resultados del runtime, no preferencias del prompt.
- **Recovery boundary:** los reintentos, fallbacks, replay y rollback se diseñan antes del tráfico en producción.

Cuando faltan estos límites, el model se convierte en el control plane por accidente.

Execution Modes

No ejecutes todos los agents de la misma manera. Ajusta el modo del runtime al trabajo.

Modo	Úsalo cuando	Requisitos del runtime
Synchronous request	El task es corto, principalmente de lectura y seguro para fallar rápido.	Timeout estricto, budget pequeño, sin side effects irreversibles, trace completo.
Async job	El task puede tomar segundos o minutos pero no necesita compensación compleja.	Cola, registro de estado, cancelación, reintentos, idempotencia, eventos de progreso.
Durable workflow	El task abarca aprobaciones, sistemas externos, reintentos o state de larga duración.	Checkpoints, capacidad de reanudar, compensación, replay, state de workflow versionado.
Event-triggered run	El task inicia desde un webhook, programación, stream o evento de sistema.	Deduplicación, identidad de evento, política de orden, backpressure, trail de auditoría.
Human-gated run	El task puede preparar trabajo pero necesita aprobación antes de ejecutarse.	Registro de aprobación, binding de acción exacta, semántica de pausa y reanudación.

El modo incorrecto genera bugs en producción. Un workflow de reembolso no debe depender de que una solicitud HTTP permanezca activa. Un clasificador corto no debe pagar el costo de complejidad de un motor de durable workflow.

Queues, Backpressure y Concurrency

Los agents consumen recursos escasos: cuota de model, capacidad de tools, tiempo de aprobación humana, conexiones a base de datos, browser workers y dinero. El runtime debe controlar la admisión y la concurrencia antes de que el loop comience a gastar.

Controles útiles incluyen colas por tenant, límites de concurrencia por ruta, límites de tasa por proveedor de model, bulkheads específicos por tool, presupuestos de reintentos, dead-letter queues y clases de prioridad. El backpressure no es solo un tema de infraestructura. Es la forma en que el sistema rechaza trabajo de bajo valor antes de que dañe el trabajo de alto valor.

La concurrencia también afecta la corrección. Dos runs no deben emitir el mismo reembolso, actualizar el mismo ticket, reescribir la misma memory o desplegar el mismo servicio sin coordinación. Usa locks, verificaciones de versión, claves de idempotencia o transiciones de state de workflow donde el trabajo duplicado sería dañino.

Rollout y Rollback

Los agents en producción cambian de más formas que los servicios normales. Un release puede cambiar el model, prompt, tool schema, retriever, memory policy, approval rule, sandbox profile, evaluator o el código del workflow. El runtime debe versionar esas piezas y registrar el set de versiones activas en cada ejecución.

El rollout debe ser gradual para agents de alto riesgo:

- comienza con evals offline;
- ejecuta pruebas shadow o replay cuando sea posible;
- habilita solo un tenant, ruta o porcentaje pequeño;
- compara traces, costos, razones de detención, denegaciones de policy y resultados visibles para el usuario;
- mantén un camino de rollback para cada componente cambiado.

El rollback debe ser operativo, no solo teórico. Los operadores deben poder deshabilitar un tool, fijar un model, revertir un prompt, endurecer una policy, detener una ruta, drenar una cola o forzar la aprobación humana sin redeplegar todo el producto.

Cómo se Componen los Capítulos de Production Runtime

Lee la sección de production runtime como un solo modelo operativo:

- [Durable Workflows](#) gestionan el state de larga duración, reintentos, checkpoints, aprobaciones, compensaciones y capacidad de reanudación.
- [Observability and Evals](#) registra lo que sucedió y convierte el comportamiento en algo que los ingenieros pueden inspeccionar.
- [Production Evaluation Feedback Loops](#) convierte fallas en producción en casos de regresión y filtros de release.
- [Cost Controls and Runtime Budgets](#) define cuánta autonomía, gasto, tiempo y atención humana puede consumir una ejecución.
- [Policy Enforcement](#) mantiene las decisiones de permisos, riesgo y cumplimiento fuera del model.
- [Event-Triggered Agents](#) muestra cómo los agents responden a eventos sin perder idempotencia, state y auditabilidad.
- [Mastra Runtime](#) traduce estas preocupaciones de producción en un estilo concreto de runtime.

Los capítulos están separados porque cada límite merece atención. En un sistema real, deben funcionar juntos.

Launch Evidence Map

Antes del lanzamiento, cada aspecto del runtime debe generar evidencia que otro ingeniero pueda inspeccionar. Un build en verde es útil, pero no prueba que el agent pueda ser operado.

Concern	Evidence Artifact	Release Question
Admission	Route table con task class, risk class, tenant scope y refusal path.	¿Qué solicitudes pueden iniciar, esperar, redirigirse o rechazarse?
State	Run-state schema, ejemplo de checkpoint y regla de eliminación.	¿Puede el equipo reconstruir lo que el agent creía y dónde se detuvo?
Policy	Decision matrix, reason codes y approval rules.	¿Puede el software detener una propuesta insegura antes de ejecutarse?
Budget	Policy de presupuesto por ruta y comportamiento ante agotamiento.	¿Qué sucede cuando el task ya no justifica más gasto?
Tools	Tool manifest con schemas, permisos, timeouts y clase de side-effect.	¿Se puede validar una llamada a tool sin confiar en el texto del model?
Memory and retrieval	Source policy, regla de frescura, regla de citación y clase de retención de memory.	¿Puede el agent explicar qué evidencia usó y por qué estaba permitida?
Observability	Trace contract y enlaces a dashboards.	¿Puede un operador pasar de un síntoma, al trace, al responsable?
Evaluation	Lista de evals bloqueantes, incident fixtures y salida de release-gate.	¿Qué fallas conocidas bloquearían este release?
Recovery	Runbook con acciones de retry, fallback, compensación y rollback.	¿Qué pueden deshabilitar o restaurar los operadores sin un redeploy completo?

La evidencia no necesita ser elaborada. Debe ser real, actual y estar enlazada desde el registro de release.

Minimal Runtime Contract

Cada ejecución en producción debe poder generar un contrato como este:

```
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;
};

```

Esto no es suficiente para implementar una plataforma completa, pero sí para hacer visibles las partes ocultas. Si una ejecución no tiene actor, tenant, route, trace ID, risk class, autonomy level, execution mode, version set, budget policy, policy version, allowed tools, status y stop reason, será difícil de operar.

Para trabajos de alto riesgo, este contrato debe almacenarse antes de la primera llamada al model. La ejecución puede cambiar el state, pero el runtime nunca debe adivinar quién la inició, qué autoridad tiene, qué versión está activa o por qué se detuvo.

Operating Dashboard View

El dashboard del runtime debe mostrar control, no solo actividad.

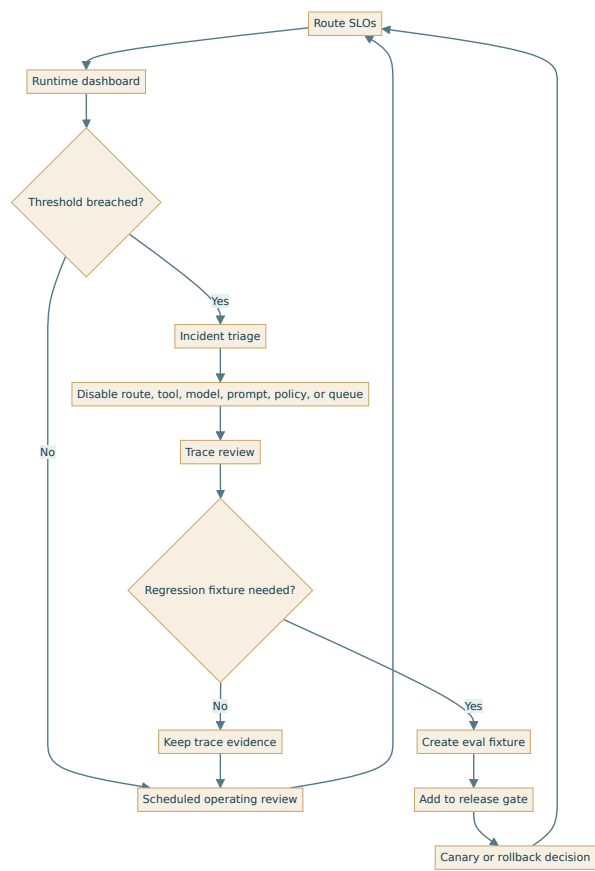
Panel	Shows	Operator Action
Active runs	route, tenant, risk class, status, workflow step, age y owner.	Cancelar, pausar o escalar trabajo atascado.
Budget pressure	token spend, tool spend, retry spend y ejecuciones cerca del agotamiento.	Bajar concurrencia, requerir aprobación o detener tareas de baja prioridad.
Policy decisions	denies, approvals, escalations, false allows y override rate.	Endurecer reglas, revisar excepciones o agregar eval fixtures.
Tool health	timeout rate, retry rate, side-effect failures y conflictos de idempotencia.	Deshabilitar un tool, drenar una cola o cambiar el camino de fallback.
Trace quality	missing spans, missing stop reasons, redaction failures y replay gaps.	Bloquear el release hasta que la evidencia esté completa.
Release versions	versiones activas de model, prompt, policy, retriever, workflow y tool schema.	Fijar, hacer rollback o canary de un componente.

Panel	Shows	Operator Action
Eval status	blocking failures, flaky cases, incident fixture failures y gate history.	Detener rollout o asignar reparación de fixtures.

Si el dashboard no puede responder “¿qué debe hacer un operador ahora?”, es solo una página de reportes, no una superficie de control del runtime.

Runtime SLO y Incident Loop

Los service-level objectives (SLOs) hacen explícita la calidad del runtime. No los defines solo para uptime. Los agentic systems también necesitan SLOs para trace coverage, policy-decision coverage, approval latency, eval-gate health, cost y stop-reason completeness.



Usa este loop después del lanzamiento y durante canaries. Una buena revisión operativa puede responder qué SLO cambió, qué ruta se modificó, qué trace prueba la falla, qué componente se puede deshabilitar y qué eval ahora previene la recurrencia.

Runtime SLO	Ejemplo de Target	Investiga Cuando	Haz Rollback o Pausa Cuando
Trace coverage	99% de los runs de alto riesgo tienen spans completos de run, route, policy, tool, approval y stop-reason.	Falta policy o stop reason en cualquier trace de alto riesgo.	Un release cambia el comportamiento riesgoso sin trace coverage.

Runtime SLO	Ejemplo de Target	Investiga Cuando	Haz Rollback o Pausa Cuando
Policy-decision coverage	100% de los paths de write, send, memory y external-action registran una policy decision.	Falta el policy span o está separado del tool span.	Se ejecuta un side effect sin evidencia de policy.
Approval latency	95% de las esperas de approval se resuelven o expiran dentro del SLA del negocio.	Las esperas de approval envejecen sin owner o expiry.	Approvals obsoletos pueden reanudar acciones cambiadas.
Cost budget	La route se mantiene dentro del presupuesto acordado por run o por task completado.	El costo por task completado se dispara después de un cambio en model, prompt o retrieval.	El agotamiento del presupuesto no detiene ni degrada de forma segura.
Eval-gate health	Los evals bloqueantes pasan antes de cambios en prompt, model, policy, tool, memory o workflow.	Fallas de eval de advertencia se agrupan en un límite.	Un fixture de incidente conocido falla o se vuelve inestable sin owner.
Stop-reason completeness	Cada run terminal registra completed, refused, failed, cancelled, timed out, blocked o needs approval.	Los estados terminales contienen <code>done</code> genérico o falta el motivo.	Los operadores no pueden distinguir entre éxito, rechazo, falla o side effect parcial.

Los números variarán según el producto. Lo importante es que cada SLO tenga un owner y una acción. Un umbral sin decisión operativa es solo una gráfica.

Runtime Checklist

Antes de que un agent en producción maneje trabajo real, responde:

- ¿Quién es owner del goal activo?
- ¿Dónde se almacena el run state durable?
- ¿Qué componente valida las propuestas del model?
- ¿Qué modo de ejecución se ajusta a este task?
- ¿Qué tools están permitidas para esta clase de task?
- ¿Qué acciones requieren approval?
- ¿Qué presupuesto aplica al run?
- ¿Qué sucede cuando se agota el presupuesto?
- ¿Qué eventos de trace son obligatorios?
- ¿Qué side effects requieren idempotencia o compensación?
- ¿Qué breaker, fallback o ruta de escalamiento existe?
- ¿Qué evals bloquean el release?
- ¿Qué política de queue, límite de concurrencia o backpressure aplica?

- ¿Qué versiones de componentes se registran en cada run?
- ¿Qué se puede hacer rollback sin redeployar todo el sistema?

Si esas respuestas son vagas, el sistema sigue siendo un prototipo, aunque ya esté sirviendo usuarios.

Failure Modes

- El model controla las transiciones de state porque el runtime no tiene workflow state.
- La policy vive en el prompt en vez de una capa de enforcement en el runtime.
- Las llamadas a tool ocurren antes de los chequeos de presupuesto, permisos, schema o approval.
- La lógica de retry repite side effects sin claves de idempotencia.
- Observability registra solo las respuestas finales, pero no las propuestas, decisiones de validación, llamadas a tool y stop reasons.
- Los evals prueban solo el texto final mientras el runtime path sigue sin probarse.
- Los operadores no pueden deshabilitar rápidamente un tool riesgoso, versión de prompt, ruta de model o paso de workflow.
- El sistema puede seguir gastando tokens, llamadas a tool y atención humana después de que el task ya no lo justifica.
- Una solicitud síncrona oculta un workflow de larga duración hasta el primer timeout o retry duplicado.
- Las queues crecen sin backpressure, prioridad, cancelación o manejo de dead-letter.
- Los runs cancelados detienen la UI pero no la llamada a tool en queue o el workflow externo.
- Un cambio en model o prompt se lanza sin traces versionados, evals dirigidos o un plan de rollback.
- Un fallo parcial parece éxito porque el runtime registra la respuesta final pero no el side effect fallido.
- Existe state durable, pero el model context y el workflow state no concuerdan sobre qué paso está activo.

Design Rule

El runtime de producción es donde la arquitectura agentic se vuelve honesta. Si el runtime no puede explicar, limitar, reproducir y detener el agent, el model no es el único riesgo. La arquitectura lo es.

Continúa con [Durable Workflows](#) para ejecución reanudable, [Observability and Evals](#) para evidencia operativa y [Policy Enforcement](#) para decisiones de autoridad.

Capítulos Relacionados

- [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

Este walkthrough convierte un agent derivado de laboratorio en un candidato para producción. Es agnóstico al framework: los mismos filtros aplican ya sea que la implementación use TypeScript directo, Python, LangGraph, AutoGen, Mastra, CrewAI o un mini-runtime personalizado.

Descarga la [deployment walkthrough review checklist](#) antes de usar este capítulo para una revisión de lanzamiento.

Para ejemplos completos, utiliza los [Capstone Projects](#) después de este capítulo.

El objetivo no es desplegar más rápido. El objetivo es desplegar con suficiente control para que el equipo pueda inspeccionar, detener, reproducir y mejorar el sistema después de que lleguen usuarios reales.

Alcance

Usa este walkthrough para sistemas que pueden leer datos privados, invocar tools, escribir memory, enviar mensajes, crear borradores, ejecutar pasos de workflow o influir en decisiones de negocio.

Para demos desechables, mantén el proceso más ligero. Para producción, no omitas los filtros que correspondan a la autoridad del sistema.

Preguntas de Preparación para el Despliegue

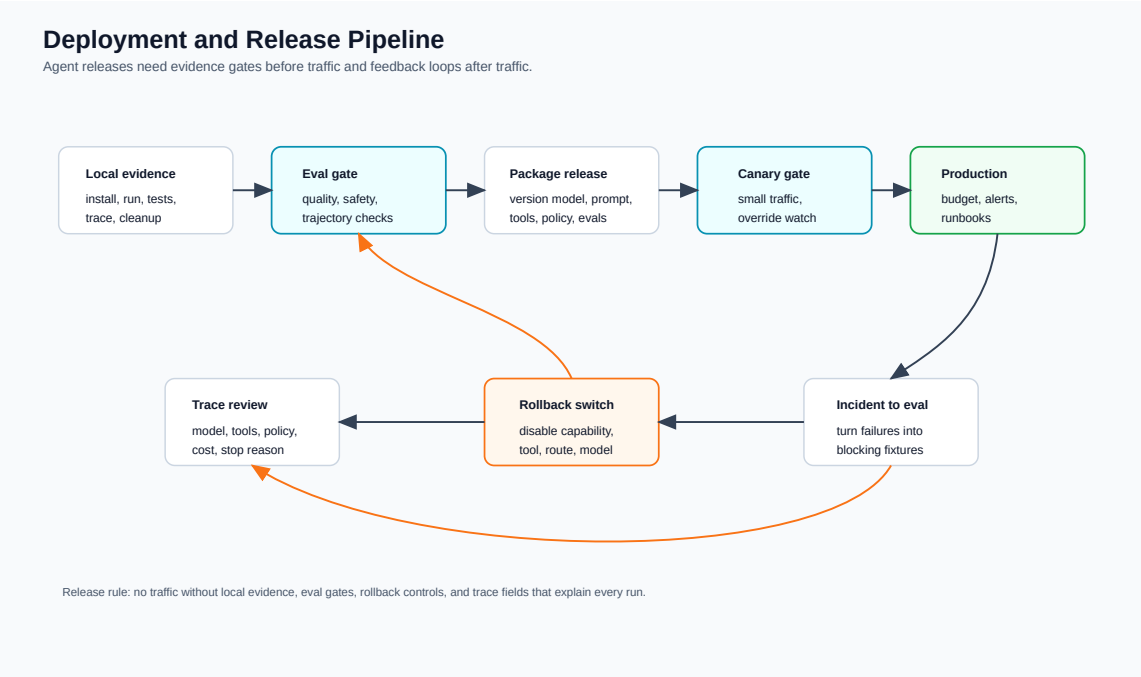
Usa estas preguntas antes de promover un laboratorio, implementación de pattern o capstone a un servicio:

Pregunta	Evidencia de Lanzamiento
¿Qué autoridad tiene el agent?	Inventario de lectura, escritura, aprobación, tool, memory y acciones hacia el usuario.
¿Qué debe ser durable?	Checkpoints para aprobaciones, reintentos, efectos secundarios y esperas de workflow.
¿Qué bloquea el lanzamiento?	Tests, evals, revisión de traces, verificaciones de policy y filtros de seguridad.
¿Qué se puede deshabilitar sin desplegar?	Ruta de model, versión de prompt, capability de tool, escrituras de memory, workflow o ruta completa de agent.
¿Qué pueden inspeccionar los operadores?	Runbook, dashboard de traces, dashboard de eval, versión de config y registro de incidentes.
¿Qué sucede durante una falla parcial?	Reintento, compensación, degradación, escalamiento y reglas de motivo de detención.

El lanzamiento no está listo cuando la única prueba es “el demo funcionó”. Está listo cuando un segundo ingeniero puede desplegarlo, inspeccionarlo, detenerlo y reproducirlo.

Pipeline de Lanzamiento

Usa este diagrama como ruta de control de despliegue. Un lanzamiento de agent en producción necesita evidencia local, filtros de eval, observación canaria, controles de rollback y retroalimentación de incidentes a eval.



1. Desarrollo Local

El desarrollo local debe probar el contrato del runtime antes de que exista infraestructura en la nube.

Evidencia local requerida:

Evidencia	Prueba Requerida
install	un checkout limpio puede instalar dependencias
run	un comando ejecuta el vertical slice
test	pruebas unitarias y de trayectoria pasan
eval	al menos un eval que bloquea el lanzamiento corre localmente
trace	la ejecución local emite eventos de trace estructurados
cleanup	el state local y datos temporales pueden eliminarse

Comandos locales sugeridos:

```
npm test
npm run typecheck
npm run book:build
```

Para variantes en frameworks de Python, agrega el entorno virtual específico del proyecto, comandos de instalación, test y eval al README del laboratorio.

2. Configuración y Secrets

La configuración debe hacer explícito el comportamiento de despliegue sin exponer secrets.

Usa estos grupos de variables de entorno:

Grupo	Ejemplos
model provider	<code>OPENAI_API_KEY</code> , nombre de model, timeout, límite de reintentos
runtime	environment, región, nombre del servicio, versión de lanzamiento
storage	checkpoint store URL, trace store URL, memory store URL
policy	policy version, approval mode, capabilities deshabilitadas
observability	endpoint de exportación de trace, modo de muestreo, modo de redacción
evals	versión del dataset de eval, umbral de lanzamiento, modo de falla

Reglas:

- haz commit de `.env.example`, no de `.env`;
- mantén los secrets en el secret manager de la plataforma de despliegue;
- falla el inicio si faltan secrets requeridos;
- registra qué versión de configuración se cargó, no los valores de secrets;
- trata las versiones de prompt, model, tool, policy y eval como insumos del lanzamiento.

3. Persistencia y Checkpointing

La persistencia depende de la autoridad. Una respuesta de solo lectura puede ser stateless. Un workflow que espera aprobación, reintenta tools o crea efectos secundarios necesita state durable.

Elige el límite mínimo de persistencia que soporte recuperación:

Necesidad	Límite de Persistencia
respuesta solo de request	registro de request más trace
continuidad de conversación	thread state o conversation store
espera de aprobación humana	checkpoint más registro de aprobación
efecto secundario de tool	idempotency key más registro de efecto secundario
workflow de larga duración	workflow state más checkpoints de pasos
memory	memory store gobernado con retención y eliminación

Haz checkpoint en cada paso visible externamente:

1. request aceptada;
2. acción planificada;
3. decisión de policy;
4. solicitud o resultado de aprobación;
5. intento de llamada a tool;
6. resultado de efecto secundario;
7. respuesta final;
8. resultado de eval o calidad post-ejecución.

Los reintentos deben leer el checkpoint y decidir si continuar, compensar o detenerse. No deben reproducir efectos secundarios a ciegas.

4. Exportación de Observabilidad

La observabilidad de agent debe explicar una ejecución y agregar muchas ejecuciones.

Exporta estos eventos:

Evento	Campos Requeridos
run	trace ID, run ID, actor, tenant, environment, release
model	model, versión de prompt, referencia de entrada, estado de salida, tokens, costo, latencia
tool	nombre de tool, argumentos redactados, autorización, estado, conteo de reintentos, idempotency key
retrieval	IDs de fuente, frescura, decisión de acceso, requisitos de citación
memory	IDs de lectura, IDs de escritura, clase de retención, base de policy
policy	policy version, decisión, reason code, efecto de enforcement
approval	rol del aprobador, acción exacta, expiración, resultado
eval	case ID, versión de evaluator, score, umbral, pass/fail

No almacenes secrets, credenciales, datos de pago o contenido privado sin que la policy de retención lo permita explícitamente. Prefiere referencias a registros cifrados cuando el contenido sin procesar no sea necesario para depuración.

5. Eval Gate en CI

CI debe bloquear cambios riesgosos antes del despliegue.

Vincula subconjuntos de eval al tipo de cambio:

Cambio	Eval que Bloquea
prompt	éxito de task, validez de schema, cumplimiento de policy

Cambio	Eval que Bloquea
model	éxito de task, comportamiento de rechazo, calidad de argumentos de tool, costo
tool	autorización, idempotencia, manejo de errores
retrieval	acceso a fuente, frescura, corrección de citación
memory	alcance de lectura, policy de escritura, comportamiento de eliminación
workflow	corrección de ruta, reintento, cancelación, reanudación
policy	false allow, false deny, enrutamiento de aprobación

Un filtro mínimo de CI debe ejecutar:

```
npm test
npm run typecheck
```

Agrega comandos de eval específicos del proyecto junto a la implementación. El filtro debe fallar cerrado: si el dataset de eval no puede cargarse, el lanzamiento debe detenerse.

GitHub Actions Gate

Un workflow mínimo de GitHub Actions debe separar los tests ordinarios de los checks de agent que bloquean el lanzamiento.

```
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
```

Para agents en Python, agrega setup-python, instalación de dependencias, pruebas unitarias y comandos de eval. Mantén los secrets de producción fuera de los jobs de pull-request. CI debe usar fixtures sintéticos, tools simulados, traces redactados y credenciales de staging solo cuando estén explícitamente aprobadas.

El filtro de lanzamiento debe publicar un pequeño resumen de evidencia: commit SHA, versión del dataset de eval, checks aprobados, checks fallidos, resultado del contrato de trace y owner del lanzamiento. Un badge verde de CI no es suficiente cuando el agent puede invocar tools o afectar usuarios.

6. Rollout

Haz rollout por capability, no por esperanza.

Usa etapas:

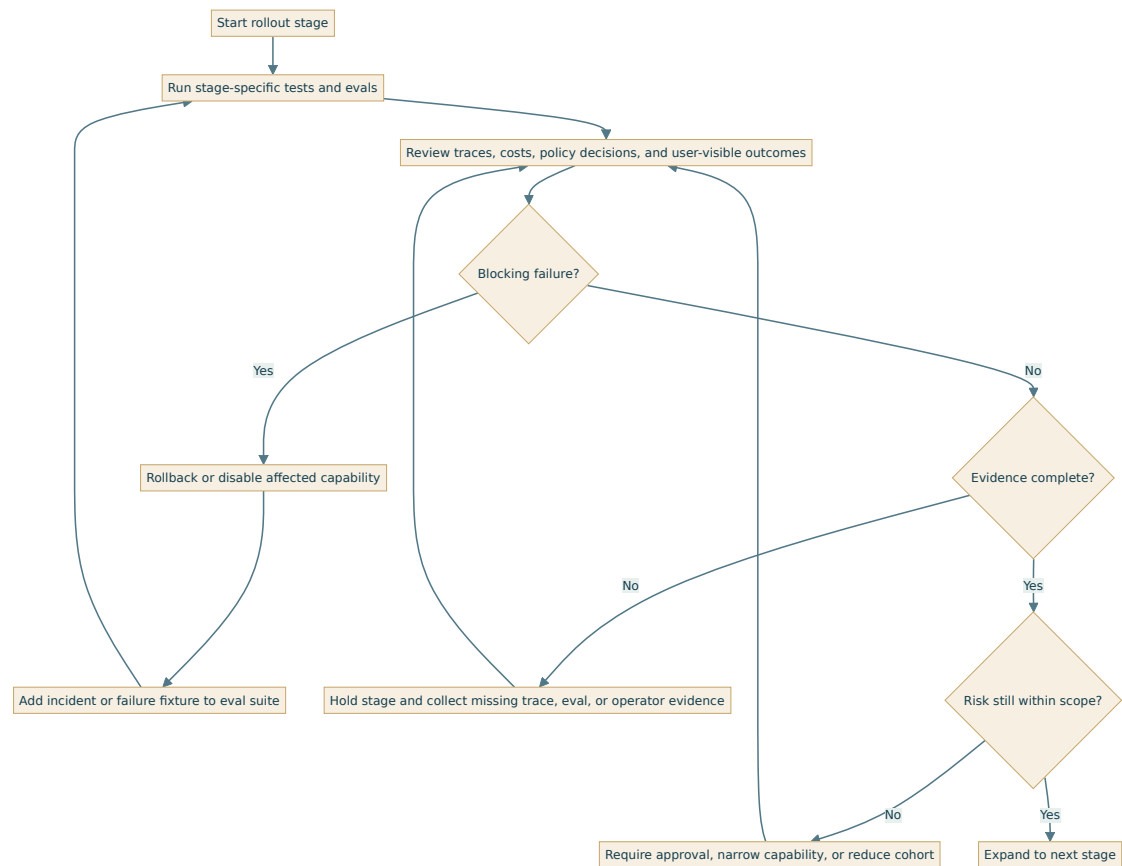
1. ejecución local con fixtures deterministas;
2. ejecución en staging con datos sintéticos;
3. ejecución interna con autoridad de solo lectura;
4. tenant o cohorte limitada;
5. tráfico expandido con dashboards y alertas;
6. lanzamiento completo después de revisión de traces y eval.

En cada etapa, registra:

- versión de lanzamiento;
- versión de model y prompt;
- versión de schema de tool;
- versión de policy;
- versión de dataset de eval;
- estado de exportación de trace;
- owner de rollback.

Flujo de Decisión de Rollout

Usa este flujo en cada etapa de rollout. El objetivo es que la expansión sea una decisión basada en evidencia, no el siguiente paso por defecto.



7. Rollback y Kill Switch

Todo agent de producción necesita una vía rápida de desactivación.

Define kill switches en varios niveles:

Capa	Acción de desactivación
model	redirigir al modelo anterior o a un fallback determinista
prompt	revertir la versión del prompt
tool	deshabilitar una capability en el registro de tools
memory	deshabilitar escrituras manteniendo lecturas disponibles si es seguro
workflow	pausar nuevas ejecuciones y dejar que las ejecuciones en curso seguras terminen
policy	cambiar acciones riesgosas a requerir aprobación o denegarlas
agent	redirigir el tráfico a un humano o a un workflow determinista

El rollback no debe requerir un despliegue de código para fallas comunes. La desactivación de tools, rollback de model, rollback de prompt y endurecimiento de policy deben ser controles operativos.

8. Runbook de Producción

Crea un runbook antes del lanzamiento.

Runbook mínimo:

```
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:
```

El runbook debe enlazar al ADR de selección de framework, hoja de preparación para producción, suite de eval y dashboard de despliegue.

9. Ruta Concreta de Runtime

Usa esta ruta cuando un laboratorio o capstone se convierte en un servicio. Mantiene el código de framework detrás de contratos controlados por el producto.

Paso	Artifact	Señal de finalización
package	imagen de contenedor o bundle serverless	la imagen contiene solo archivos de runtime requeridos, lockfile y plantilla de configuración
entrypoint	HTTP handler, consumidor de queue o worker de workflow	la solicitud crea un run ID y trace ID antes de iniciar trabajo de model o tool
config	<code>.env.example</code> , nombres de secretos, versión de policy	el inicio falla cerrado si faltan valores requeridos
state	tabla de base de datos, checkpointer o almacén de workflow	una ejecución interrumpida o reintentada se reanuda desde el state conocido
tools	registro más metadatos de capability	cada tool tiene clase de side-effect, owner, timeout, retry y regla de aprobación

Paso	Artifact	Señal de finalización
evals	comando de release gate	CI bloquea el despliegue si fallan grounding, policy, schema o trajectory evals
observability	exportación de trace y dashboard	una ejecución puede reconstruirse sin secretos sin procesar
rollback	feature flag, cambio de ruta o desactivación de tool	el owner puede deshabilitar una capability riesgosa sin desplegar código

Contrato mínimo de servicio:

`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`

Para despliegues de queue o workflow, mantén el mismo contrato aunque cambie el transporte. El request envelope, registro de state, trace ID, decisión de policy y resultado de eval deben lucir igual en HTTP, worker y jobs programados.

10. Formas de Despliegue en la Nube

Diferentes formas en la nube pueden alojar el mismo contrato de agent. Elige la forma más simple que preserve state, policy, traces y rollback.

Forma	Usar cuando	Controles requeridos
container service	agent HTTP o worker necesita proceso de larga vida, caché local o runtime personalizado	health check, límite de autoscaling, secret manager, exportación de trace, kill switch
serverless function	paso corto y sin estado con timeout estricto y sin espera de aprobación	almacén de state externo, idempotency key, presupuesto de timeout, prueba de cold-start
queue worker	trabajo disparado por eventos o en segundo plano	dead-letter queue, retry policy, backpressure, procedimiento de replay
workflow engine worker	trabajo de larga duración, aprobaciones, compensación o reanudación tras falla	almacén de checkpoint, definición de workflow versionada, dashboard de ejecuciones atascadas
scheduled job	eval periódica, limpieza de memory, ingestión o generación de reportes	lock, idempotency, registro de última ejecución, alerta por ejecución perdida

El despliegue en la nube no debe cambiar el modelo de autoridad del agent. Si una ejecución local requiere aprobación antes de enviar correo, el worker en la nube debe requerir la misma aprobación. Si el trace local redacta argumentos de tool, el trace en la nube también debe redactarlos.

11. Notas de Despliegue para Research RAG

Los sistemas Research RAG requieren controles de despliegue adicionales porque la recuperación puede exponer material prohibido, obsoleto o no soportado.

Controles requeridos en runtime:

Control	Regla de producción
ingestion	almacenar ID de fuente, título, versión, frescura, owner, grupo ACL y etiqueta de citación
retrieval	recuperar candidatos con metadatos, no solo texto
source filter	aplicar ACL, frescura y tipo de fuente antes de ensamblar el context
context packet	incluir evidencia, omisiones y etiquetas de citación como campos estructurados
answer synthesis	responder solo a partir del evidence packet aprobado
citation eval	bloquear respuestas que citen fuentes faltantes, obsoletas o prohibidas
fallback	devolver fuentes aprobadas ordenadas o escalar cuando la evidencia es débil

Secuencia de despliegue:

1. desplegar retrieval en modo solo lectura;
2. comparar candidatos recuperados con el resultado de source-filter;
3. habilitar answer synthesis solo para usuarios internos;
4. condicionar el release a fidelidad de citación, omisión de fuentes prohibidas y rechazo de fuentes obsoletas;
5. agregar dashboards para evidencia faltante, hits de fuentes obsoletas, intentos de fuentes prohibidas y fallas de citación;
6. mantener un kill switch que desactive la síntesis y devuelva solo listas de fuentes aprobadas.

Esta ruta conecta directamente con el [Research RAG Agent capstone](#) y el slice nativo de LangGraph bajo `native-framework-examples/langgraph-research-rag/`.

Notas de Despliegue Específicas de Framework

Forma de Framework	Nota de despliegue
LangGraph	Usa checkpoints persistentes para esperas de aprobación, reanudación y tolerancia a fallos. Trata el thread ID como state sensible.

Forma de Framework	Nota de despliegue
AutoGen	Persiste transcripts con redacción y metadatos de terminación. Evalúa el comportamiento de roles, no solo el output final.
Mastra	Mantén explícito el empaquetado de TypeScript runtime: agents, workflows, tools, memory, evals y exportación de trace requieren ownership.
CrewAI	Mantén el state de Flow separado de la colaboración local de Crew. Valida el output del crew antes de que Flow lo acepte.
Mini-runtime	Usa el proceso de despliegue para decidir qué controles de producción debes construir tú mismo y cuáles deben moverse a la infraestructura de plataforma.

Completo Cuando

El sistema es desplegable cuando:

- la configuración local es reproducible;
- los secretos y la configuración están separados;
- la persistencia coincide con la autoridad;
- los traces se exportan y redactan;
- los evals bloquean cambios riesgosos;
- las etapas de rollout están documentadas;
- el rollback funciona sin cambios de código para fallas comunes;
- el runbook nombra owners, dashboards y acciones ante incidentes.

Hasta entonces, el sistema puede ser útil, pero no está listo para producción.

Durable Workflows

Los durable workflows hacen que los agentic systems sean reanudables y auditables al encargarse de los retries, checkpoints, approvals, compensaciones y el state de larga duración.

Fuente y descargas

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

Propósito

Los durable workflows envuelven los pasos del agent en un modelo de ejecución reanudable. El workflow controla el orden, checkpoints, retries, approvals, compensaciones, cancelaciones y el state de larga duración. Los agents realizan trabajo acotado dentro de los pasos del workflow.

Esta distinción es importante. Un agent loop no es un workflow engine. Un model puede proponer la siguiente acción, pero el workflow decide qué paso está activo, qué state es durable, cuál retry es seguro, qué approval está pendiente y cómo el sistema se reanuda después de una falla.

Úsalo cuando

- El trabajo abarca minutos, horas, sistemas externos o approvals humanos.
- Las llamadas a tools pueden fallar y deben reintentarse de forma segura.
- El state debe sobrevivir reinicios de proceso, despliegues, timeouts y caídas parciales.
- Los efectos secundarios requieren idempotencia, compensación, auditoría o approval.
- Los operadores necesitan replay, cancelación, rollback y diagnóstico de incidentes.

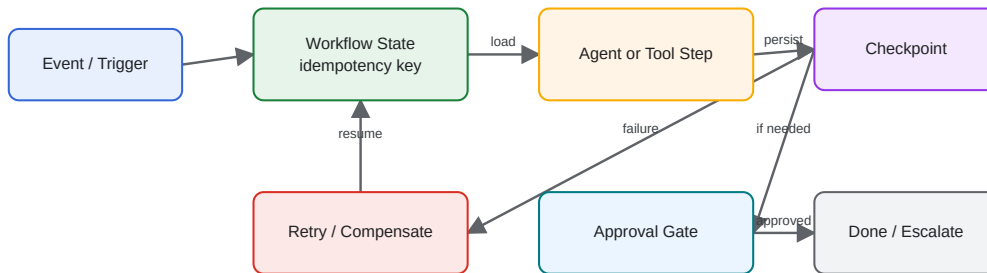
Evítalo cuando

- El task es una respuesta corta y sin state.
- No se requieren efectos secundarios externos, retries, approvals ni state durable.
- El workflow engine ocultaría más comportamiento del que aclara.
- El equipo no puede definir los límites de los pasos ni las reglas de idempotencia.
- El sistema no puede decidir qué debe ocurrir tras una finalización parcial.

Arquitectura

Usa este diagrama para entender Durable Workflows como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el runtime controla el state durable, retries, traces, triggers, configuración de despliegue y controles operativos.

Durable Workflow



Forma del sistema

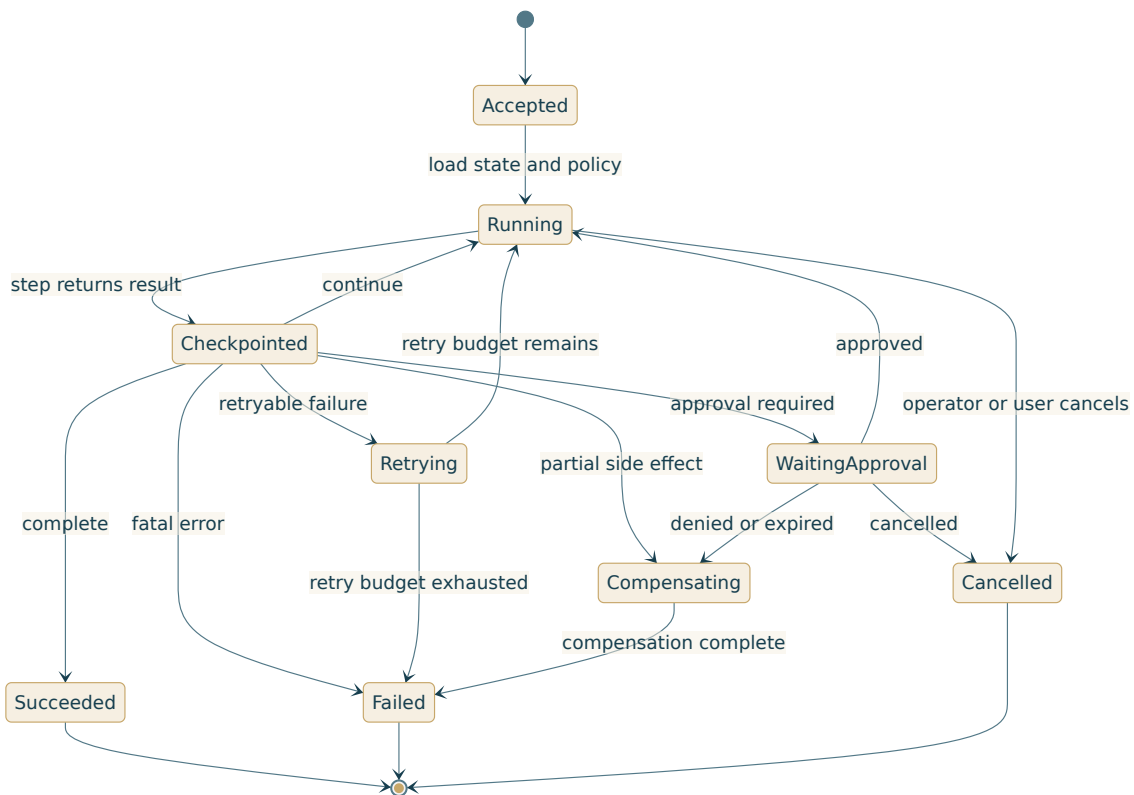
- **Límite del pattern:** el workflow controla el orden de los pasos, state durable, retries, approvals, cancelación, compensación y replay.
- **Límite del agent:** el agent maneja incertidumbre acotada dentro de un paso, luego retorna un resultado tipado, rechazo, error o escalamiento.
- **Propietario del state:** el state del workflow es durable; el context del model es temporal.
- **Límite de efectos secundarios:** cada acción externa tiene una clave de idempotencia, checkpoint y registro de auditoría.
- **Promesa operativa:** una ejecución puede fallar, pausarse, reanudarse y explicarse sin perder su lugar.

Protocolo central

1. Recibe una solicitud, evento, schedule o comando de workflow con una clave de idempotencia.
2. Crea o carga el state del workflow con caller, goal, paso actual, policy context y trace ID.
3. Ejecuta un paso acotado: código determinista, agent loop, llamada a tool, espera de approval o evaluator.
4. Checkpoint del state, resultado, costo, eventos de trace, errores y approvals pendientes.
5. Decide la siguiente transición: continuar, retry, esperar, compensar, cancelar, escalar o completar.
6. Reanuda desde el último checkpoint durable tras reinicio o despliegue.
7. Registra el resultado final, motivo de detención, efectos secundarios y metadatos de replay.

Mapa de transiciones del workflow

Usa este mapa para revisar si cada transición tiene un checkpoint durable, evento de trace y propietario. El workflow debe poder detenerse y reanudarse desde cualquier state no terminal sin repetir efectos secundarios inseguros.



Notas de implementación

Mantén el state del workflow separado del state de conversación del model.

```

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;
};

```

Cada paso debe retornar una transición, no solo texto:

```

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 };
```

La idempotencia no es opcional para los efectos secundarios:

```
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' }  
      ]  
    }  
  };  
}
```

Reintentar una llamada al model usualmente es seguro. Reintentar un efecto secundario solo es seguro cuando el efecto es idempotente o está protegido por el state del workflow.

Modos de falla

- Los pasos con efectos secundarios se reintentan sin idempotencia.
- El state de approval solo se guarda en el historial de chat y desaparece tras un reinicio.
- El estado del workflow indica éxito mientras el paso del agent retornó incertidumbre o evidencia faltante.
- Un despliegue cambia prompts o manifiestos de tools a mitad de una ejecución sin versionado.
- El workflow no puede reanudarse porque el paso actual es implícito.
- Falta compensación para acciones externas parcialmente completadas.
- La cancelación detiene la UI pero no el tool o worker subyacente.
- Los traces muestran el output final pero no los checkpoints, retries, approvals y efectos secundarios.
- El workflow engine oculta el comportamiento del agent en vez de hacerlo inspeccionable.

Estrategia de evaluación

Los evals de durable workflow deben probar recuperación, no solo caminos felices.

- Prueba reinicio desde cada checkpoint significativo.
- Prueba entrega duplicada de eventos con la misma clave de idempotencia.

- Prueba falla de tool reintentable y falla fatal de tool.
- Prueba espera de approval, denegación de approval, timeout de approval y reanudación tras approval.
- Prueba despliegue durante un workflow en espera o ejecución.
- Prueba cancelación y compensación.
- Prueba replay desde un trace de producción.
- Prueba que cada resultado final tenga motivo de detención y auditoría de efectos secundarios.

Un eval compacto de workflow puede verse así:

```
{
  "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"]
  }
}
```

Mide tasa de éxito al reanudar, seguridad ante eventos duplicados, éxito de retries, éxito de compensaciones, tiempo de espera de approvals, corrección de cancelaciones, éxito de replay y recurrencia de incidentes.

Lista de verificación para producción

- Define el state del workflow separado del context del prompt.
- Da a cada ejecución de workflow un workflow ID, trace ID y clave de idempotencia estables.
- Haz checkpoint después de cada efecto secundario externo y solicitud de approval.
- Haz que los efectos secundarios sean idempotentes o compensables.
- Versiona prompts, políticas, rutas de model, manifiestos de tools y definiciones de workflow.
- Trata espera de approval, timeout, denegación, cancelación y escalamiento como transiciones normales.
- Persiste suficiente data de trace para hacer replay de ejecuciones fallidas.
- Define propiedad operativa para workflows atascados, en espera, fallidos y en compensación.
- Agrega alertas para tormentas de retries, approvals atascados, efectos secundarios duplicados y fallas de replay.
- Convierte incidentes de workflow en producción en evals de regresión.

Recorrido de código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

Código fuente

Este pattern actualmente no tiene un fragmento de código dedicado. Usa los enlaces de descarga y fuente a continuación para acceder a la carpeta completa del pattern.

Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `durable-workflow-pattern/` actual de este repositorio.

Patterns relacionados

- [Agent Loop](#)
- [Goals and State](#)
- [Human Approval Gates](#)
- [Policy Enforcement](#)
- [Self-Healing Workflows](#)
- [Production Evaluation Feedback Loops](#)
- [Pattern Composition Playbook](#)

Observability and Evals

Observability registra lo que sucedió. Evals decide si el comportamiento es suficientemente bueno. Release gates decide si un cambio puede salir a producción.

Fuente y descargas

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

Propósito

El patrón de Observability and Evals hace que el comportamiento del agent sea inspeccionable, reproducible y testeable. Observability registra lo que hizo el agent, por qué lo hizo, qué observó, qué cambió, cuánto costó y por qué se detuvo. Evals convierte esos traces, tasks conocidos, incidentes y near misses en release gates.

Este patrón importa porque las fallas de un agent rara vez viven solo en la respuesta final. Viven en la trayectoria: una retrieval faltante, una tool call insegura, una policy denegada ignorada, un loop que nunca convergió, una aprobación humana que se omitió o una actualización de model que cambió el plan. Registrar solo respuestas finales no es observability, y revisar solo respuestas finales no es suficiente evaluación.

Lee esto después de los capítulos de runtime y seguridad si estás diseñando para producción. El runtime le da al agent un entorno controlado para ejecutarse; seguridad define lo que debe bloquearse o aprobarse; observability y evals demuestran si esos controles funcionaron.

Usar cuando

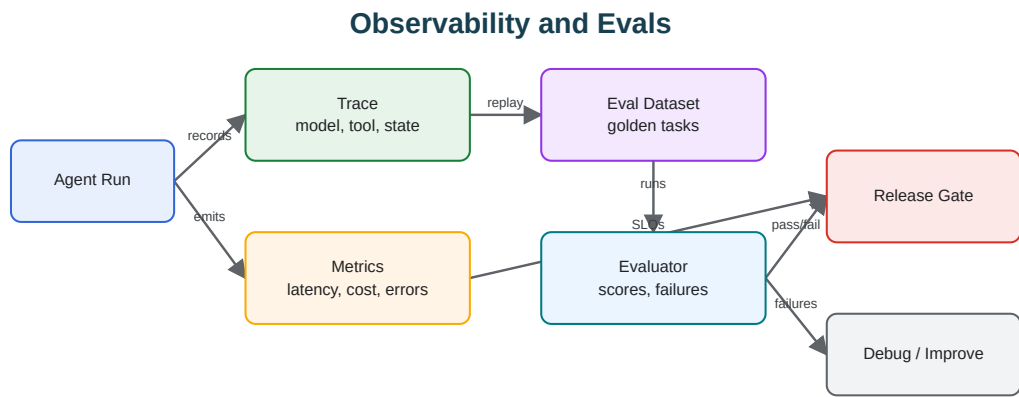
- Las decisiones del agent afectan usuarios, dinero, datos o sistemas externos.
- Necesitas pruebas de regresión para prompts, tools, routing o workflows.
- Las fallas son difíciles de reproducir solo con las respuestas finales.
- Necesitas comparar cambios de model, prompt, tool, memory o policy antes de lanzar.
- Operas workflows donde la razón de detención importa tanto como la respuesta.

Evitar cuando

- No puedes almacenar traces de forma segura por restricciones de privacidad o regulación.
- El prototipo es desechable y no tiene usuarios operativos.
- Solo registras respuestas finales y llamas a eso observability.
- Nadie se hace responsable del eval suite después de la primera versión.
- La organización no está lista para definir reglas de retención, redacción y acceso para los traces.

Arquitectura

Usa este diagrama para leer Observability and Evals como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: el runtime es dueño del state durable, retries, traces, triggers, configuración de despliegue y controles operativos.



Forma del sistema

- **Runtime boundary:** el runtime crea trace IDs, run IDs, span IDs y idempotency keys antes de que el agent comience a trabajar.
- **Span model:** llamadas a model, retrieval, tool, decisiones de policy, esperas de aprobación, decisiones de evaluador, retries y pasos de workflow son spans de primera clase.
- **Eval boundary:** los evals no son una ocurrencia tardía. Están conectados a traces, incidentes, release gates y cambios de model o prompt.
- **Data boundary:** los traces se redactan antes de almacenarse, se retienen por un periodo definido y se protegen como datos de producción.
- **Operational boundary:** los dashboards conectan comportamiento, calidad, costo, latencia y respuesta a incidentes.

Observability y evaluación están relacionadas, pero no son la misma capa.

Capa	Pregunta que responde	Artifact típico
Logs	¿Qué evento ocurrió?	Registros de eventos estructurados.
Metrics	¿Con qué frecuencia, qué tan lento, qué tan caro?	Contadores, gauges, histogramas, SLOs.
Traces	¿Qué camino tomó una ejecución?	Run, iteration, model, tool, policy, approval y evaluator spans.
Evals	¿Fue aceptable el comportamiento?	Casos de prueba, resultados esperados, graders, umbrales e informes de fallas.
Release gates	¿Puede salir este cambio?	Subconjuntos de eval requeridos y registros de aprobación.

No combines todo esto en un solo dashboard. Un dashboard puede mostrar síntomas. Un trace puede explicar una ejecución. Un eval puede bloquear un mal cambio antes de que llegue a los usuarios.

Protocolo central

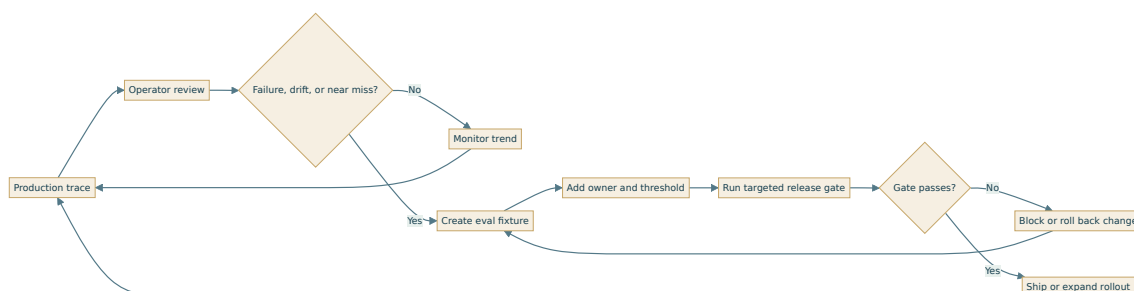
1. Inicia cada ejecución con un trace ID, run ID, request ID y caller context estables.
2. Registra cada span de model, tool, retrieval, policy, approval, evaluator y workflow.
3. Captura suficiente input, output, configuración y referencias de evidencia para reproducir el comportamiento.
4. Redacta secretos, credenciales, datos privados y contenido crudo innecesario antes de persistir.
5. Almacena la razón de detención, estado, costo, latencia, token count, tool count, retry count y resultado de policy.
6. Convierte incidentes, near misses y traces representativos en eval fixtures.
7. Bloquea cambios riesgosos con el subconjunto relevante de eval antes del despliegue.
8. Mantén los evals fallidos asociados a responsables, decisiones de lanzamiento y trabajo de seguimiento.

El operational loop debe ser explícito:

1. El runtime emite traces y metrics.
2. Los operadores inspeccionan fallas, near misses y outliers.
3. Los ingenieros convierten ejemplos útiles en eval fixtures.
4. Los release gates ejecutan los fixtures relevantes antes de cambios en model, prompt, policy, tool, memory o workflow.
5. Los nuevos traces de producción confirman si el cambio mejoró el comportamiento o solo movió la falla.

Notas de implementación

Usa este loop para conectar evidencia del runtime con decisiones de lanzamiento. El paso importante es el handoff de un trace de producción a un eval fixture nombrado que pueda bloquear el siguiente cambio riesgoso.

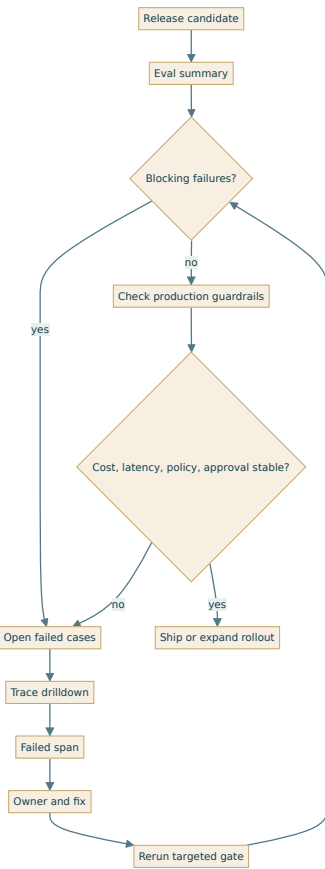


- Traza a nivel de run, iteración de loop, llamada a model, llamada a tool, paso de workflow y resultado de evaluator.
- Almacena suficiente detalle de input/output para reproducir fallas, con redacción para datos sensibles.
- Mantén datasets dorados para routing, structured outputs, planes de tool y respuestas finales.
- Trata los eval failures como bloqueadores de lanzamiento para agents en producción.
- Rastrea tanto la calidad final como la calidad de la trayectoria. Una buena respuesta producida a través de un camino inseguro de tool sigue siendo una falla.

- Mantén los trace schemas estables. Si cada servicio registra campos diferentes, depurar se vuelve arqueología.
- Asocia los eval cases al pattern que protegen: routing, retrieval, uso de tool, enforcement de policy, memory, aprobación humana o coordinación multi-agent.
- Separa product analytics de agent observability. Product analytics dice lo que hicieron los usuarios. Agent observability dice lo que el sistema hizo en su nombre.
- Almacena identificadores para prompts, models, tools, policies, retrievers, memory stores y versiones de harness. Sin versiones, un trace explica lo que pasó pero no lo que cambió.
- Trata “no trace” como un defecto de producción. Una ejecución de agent sin trace no puede depurarse, reproducirse ni defenderse en una revisión de incidentes.

Modelo de revisión de dashboard de eval

Un dashboard de eval debe ayudar a un ingeniero a decidir si lanzar, hacer rollback o inspeccionar una ejecución específica. No construyas un muro de gráficos que no pueda responder qué trace falló y quién es responsable de la corrección.



Usa esto como el layout mínimo del dashboard:

Panel	Muestra	Pregunta de release
Release Gate	eval suite, componente cambiado, pass/fail, cantidad de bloqueos, responsable.	¿Puede salir este cambio?
Failure Table	case ID, severidad, comportamiento esperado, comportamiento real, boundary protegido.	¿Qué fallas importan?

Panel	Muestra	Pregunta de release
Trace Drilldown	run, model, retrieval, tool, policy, approval, evaluator spans.	¿Dónde se rompió el camino?
Guardrails	razones de detención, denegaciones de policy, esperas de aprobación, errores de tool, retries.	¿La autonomía se mantuvo dentro de su boundary?
Cost And Latency	latencia p50/p95, costo, token count, tool count por ruta.	¿El cambio creó una regresión operativa?
Incident Conversion	issue de producción, trace fuente, nuevo fixture, responsable, fecha límite.	¿Esta falla será detectada la próxima vez?

Un dashboard útil parte de una decisión de release y profundiza en la evidencia del trace. Si un eval fallido no puede abrir el trace fuente, el dashboard reporta calidad sin explicar el comportamiento.

Contrato mínimo de Trace

Como mínimo, cada ejecución en producción debe conectar estos registros:

- identidad de la ejecución: trace ID, run ID, request ID, actor, tenant, environment y version set;
- goal y stop state: requested goal, accepted goal, status, stop reason y error class;
- context: context packet ID, retrieved evidence IDs, memory IDs, omitted-source notes y redaction level;
- actividad del model: model, prompt version, tool schema version, token counts, latency, cost y output status;
- actividad del tool: tool name, arguments after redaction, authorization decision, result status, side-effect record, idempotency key y retry count;
- actividad de policy: policy version, decision, reason code, approval requirement y escalation owner;
- actividad de memory: read IDs, write IDs, retention class, consent or policy basis y correction path;
- actividad de evaluation: evaluator version, case ID cuando aplique, score, threshold y pass or fail decision.

El trace no debe almacenar cada byte crudo por defecto. Debe guardar suficiente evidencia estructurada para reconstruir el camino de forma segura.

Ejemplo de evento de Trace

```
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';
};

```

Este evento no pretende ser el único schema en el sistema. Es un contrato para correlación. Un trace de proveedor de model, un span de OpenTelemetry, un evento del workflow engine y un resultado de eval pueden mapearse en él.

Ejemplo de fixture de Eval

```

{
  "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"
  }
}

```

```
}  
}
```

Tipos de Eval

Los agent evals necesitan más de un score.

Eval type	Qué protege	Ejemplo de verificación
Task success	El trabajo visible para el usuario fue completado.	El support agent redacta la respuesta correcta de reembolso.
Trajectory correctness	El agent tomó un camino aceptable.	Recuperó la policy antes de redactar el reembolso.
Tool correctness	La elección del tool y los argumentos fueron válidos.	Llamó a <code>orders.lookup</code> antes de proponer la compensación.
Policy compliance	Acciones inseguras fueron bloqueadas o escaladas.	No emitió un reembolso sin aprobación.
Retrieval quality	La evidencia fue relevante, reciente y citada.	La respuesta cita la policy de reembolso activa, no una archivada.
Memory correctness	Las lecturas y escrituras de memory fueron delimitadas y revisables.	No almacenó una queja transitoria como una preferencia durable.
Autonomy safety	El sistema se detuvo en el límite correcto.	Produjo un borrador en vez de enviar el mensaje.
Recovery behavior	El manejo de fallos preservó el control.	Un timeout produjo un retry o escalamiento, no un éxito silencioso.
Cost and latency	El sistema se mantuvo dentro del presupuesto.	Un cambio en el prompt no duplicó el costo medio.

El objetivo no es construir un juez perfecto. El objetivo es hacer visibles los fallos importantes antes de que el tráfico en producción los encuentre.

Release Gates

Vincula subconjuntos de eval a tipos de cambio.

Change	Subconjunto de eval requerido
Prompt change	task success, schema validity, trajectory correctness, policy compliance
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

Change	Subconjunto de eval requerido
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

Los sistemas pequeños pueden empezar con un gate corto. Los sistemas serios eventualmente necesitan gates que correspondan al radio de impacto del cambio.

Failure Modes

- Logs que omiten el prompt, la entrada del tool o la configuración del model.
- Evals que solo revisan caminos felices.
- Métricas sin trace IDs, dificultando la investigación de incidentes.
- Almacenar datos sensibles sin reglas de retención o redacción.
- Logging solo de la respuesta final que oculta el camino que produjo la respuesta.
- Llamadas a tools sin argumentos, salidas, permisos y efectos secundarios capturados.
- Policy denials que no son visibles en traces, haciendo que el trabajo bloqueado parezca confusión del model.
- Traces que filtran secretos, credenciales, datos de clientes o razonamientos internos que no deberían almacenarse.
- Evals que prueban la calidad del texto pero ignoran la evidencia recuperada, la trayectoria del tool y el comportamiento de policy.
- Dashboards que muestran costo y latencia agregados pero no pueden profundizar en ejecuciones fallidas.
- Revisiones de incidentes que no generan nuevos casos de eval.
- Suites de eval sin dueño, sin proceso de actualización y sin autoridad de liberación.
- Traces que no pueden responder qué model, prompt, policy, tool schema, retriever o harness version produjo la ejecución.
- Evals que pasan porque simulan el comportamiento exacto del tool, memory o policy que falló en producción.
- Evaluadores en línea que silenciosamente se convierten en otro agent path no observado.

Estrategia de evaluación

- **Trace completeness:** cada ejecución en producción tiene spans correlacionados de model, tool, policy, evaluator y workflow.
- **Replayability:** los ingenieros pueden reproducir un fallo con la configuración almacenada, entradas, referencias de evidencia y tool mocks.
- **Trajectory correctness:** el agent usó los tools permitidos, respetó la policy, se detuvo por la razón correcta y no omitió aprobaciones requeridas.
- **Grounding quality:** las respuestas que dependen de retrieval citan la evidencia que realmente se usó.
- **Cost and latency regression:** los cambios en model, prompt y tool no pueden aumentar silenciosamente el costo o el tiempo de respuesta en runtime.

- **Policy-denial accuracy:** las solicitudes inseguras son bloqueadas o escaladas con una razón rastreable.
- **Incident-to-eval conversion:** los fallos repetidos o de alta severidad en producción se convierten en fixtures de regresión.
- **Memory safety:** las lecturas y escrituras de memory siguen las reglas de retención, consentimiento, corrección y eliminación para el task.
- **Autonomy boundary:** el agent se detiene para aprobación, escalamiento o handoff cuando su nivel de autonomía lo requiere.
- **Release-gate authority:** los evals fallidos bloquean el cambio o requieren una anulación explícita y rastreable.

Lista de verificación para producción

- Define un schema de trace estable antes del tráfico en producción.
- Redacta y clasifica los campos de trace antes de almacenarlos.
- Correlaciona logs, métricas, traces, resultados de eval, workflow state e incidentes visibles para el usuario.
- Asigna un dueño y un rol de decisión de liberación a cada suite de eval.
- Ejecuta evals dirigidos para cambios en prompt, model, tool, memory, policy y workflow.
- Agrega dashboards para tasa de éxito, stop reason, policy denials, tool errors, cost, latency, retries y tasa de regresión de eval.
- Define reglas de retención, control de acceso y eliminación para los datos de trace.
- Convierte incidentes y casi-incidentes en fixtures de eval antes de cerrar el seguimiento operativo.
- Versiona prompts, models, tools, políticas, retrievers, memory behavior y harness code en el trace.
- Mapea tipos de cambio a subconjuntos de eval requeridos.
- Requiere un dueño nombrado para fallos y excepciones de eval.

Recorrido de código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo que lo rodea explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el paquete de descarga y en el repositorio fuente.

`observability-and-evals-pattern/trace-contract.ts`

[Abrir fuente completa](#)

```
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)]
  };
}

```

Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta actual `observability-and-evals-pattern/` de este repositorio.

Patrones relacionados

- Evaluator-Optimizer
- Durable Workflows
- Agent Loop
- Tool Use
- Compliance/Policy Enforcer
- Human Approval Gates

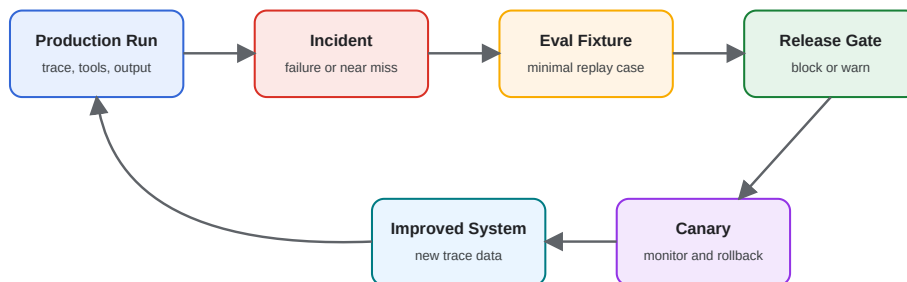
Production Evaluation Feedback Loops

La evaluación de agent no debe detenerse después del lanzamiento. Los evals previos a producción te dicen si el sistema está listo para los usuarios. La retroalimentación en producción te dice si el sistema sigue funcionando bajo tráfico real, herramientas reales, ambigüedad real, fallas reales e incentivos reales.

Descarga la [lista de verificación para revisión de production evaluation feedback loop](#) antes de usar este capítulo para una revisión operativa.

La regla principal es simple: las fallas en producción deben convertirse en pruebas futuras. Si un incidente solo genera un ticket, el sistema no aprende nada. Si el mismo incidente se convierte en un eval reproducible, el sistema se vuelve más difícil de romper de la misma manera.

Incident To Eval Feedback Loop



Por Qué Existe Este Pattern

Observability explica qué sucedió. Un production evaluation feedback loop decide qué hace el equipo con ese conocimiento.

Sin el loop, los traces se vuelven un archivo de depuración. Ayudan durante incidentes, pero no mejoran el sistema por sí solos. Con el loop, los traces se convierten en casos de regresión, filtros de lanzamiento, canary checks y señales de rollback.

Esto es especialmente importante para agentic systems porque la misma respuesta final puede producirse a través de caminos muy diferentes. Un camino puede ser correcto, fundamentado, barato y cumplir con la policy. Otro camino puede saltarse la retrieval, llamar una tool de escritura demasiado pronto, filtrar memory obsoleta y aun así producir una respuesta convincente. Los production evals deben proteger la trayectoria, no solo el texto final.

The Feedback Loop

Un production evaluation loop conecta cinco actividades:

- 1. observar ejecuciones reales;
- 2. detectar fallas, casi fallas y correcciones humanas;
- 3. convertir casos importantes en eval fixtures;
- 4. bloquear cambios contra esos fixtures;
- 5. liberar cambios gradualmente con monitoreo y rollback.

El loop aplica a prompts, tools, routes, models, policies, memory rules, retrieval indexes y la topología del agent. Cualquier cambio que pueda alterar el comportamiento debe pasar por él.

El loop debe tener un solo responsable. No necesariamente una sola persona, pero sí un solo equipo accountable. Si producto es dueño de la calidad, plataforma de los traces, seguridad de la policy y nadie es dueño de la decisión de lanzamiento, el loop fallará en los límites entre equipos.

Preguntas de Preparación para el Feedback Loop

Usa estas preguntas antes de declarar un programa de eval listo para producción:

Pregunta	Evidencia a Producir
¿Qué señales de producción se convierten en evals?	Reglas para incidentes, casi fallas, correcciones humanas, overrides, regresiones y outliers de alto costo.
¿Quién es dueño del loop?	Equipo accountable, autoridad de lanzamiento, dueño del fixture y ruta de escalamiento.
¿Qué bloquea el lanzamiento?	Lista de evals bloqueantes con severidad, dueño, razón y tipos de cambio afectados.
¿Qué expira o se elimina?	Reglas de expiración de fixtures para incidentes temporales, tools retiradas, policies antiguas y workflows migrados.
¿Cómo se mide la recurrencia?	Métricas para conversión de incident-to-eval, tasa de captura de evals, tasa de recurrencia y tiempo hasta la prueba de regresión.
¿Cómo conecta el rollback?	Umbrales de canary y objetivos de rollback para prompts, policies, tools, model routes, memory rules y retrieval indexes.

El loop solo es saludable cuando las fallas en producción cambian el comportamiento de lanzamientos futuros.

Lo Que Enseña la Producción

La producción revela fallas que las revisiones de diseño suelen pasar por alto. Los usuarios piden combinaciones de tasks no soportadas. Retrieval devuelve evidencia obsoleta pero plausible. Los errores de tool llegan en un orden extraño. Se omiten aprobaciones porque la route es incorrecta. El agent entra en loop porque una tool devolvió datos parciales. Un cambio de prompt mejora el tono pero rompe silenciosamente el comportamiento de policy. Memory almacena una preferencia incorrecta y la sigue reutilizando. Una

actualización de model cambia el comportamiento de selección de tools. Un segmento de clientes de baja frecuencia resulta tener reglas diferentes. Un override humano revela los criterios de aceptación reales que nadie documentó.

Ninguno de estos son solo eventos operativos. Son material de evaluación, y tratarlos como tal es lo que marca la diferencia entre un sistema que se desvía y uno que mejora.

Incident-To-Eval

Cada incidente serio debe producir al menos un caso de eval.

Campo	Propósito
Incident ID	Enlace al evento de producción.
Severity	Decidir si el eval bloquea el lanzamiento o solo advierte.
Owner	Persona o equipo responsable de mantener el caso actualizado.
Redacted input	Solicitud mínima del usuario o evento que reproduce la falla.
Context fixture	Documentos recuperados, memory, state, policy o salidas de tools necesarias para reproducir.
Tool trace	Llamadas a tools esperadas, prohibidas u observadas.
Expected behavior	Criterios de aprobación expresados lo más concretamente posible.
Failure mode	Qué salió mal y por qué importa.
Release gate	Si este eval bloquea cambios en prompt, policy, model o tool.

El eval debe ser más pequeño que el incidente. No conserves todo el caos de producción cuando un fixture mínimo reproduce la falla.

Un eval mínimo derivado de un incidente puede almacenarse como datos:

```
{
  "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."
}

```

El fixture codifica el comportamiento que el sistema debe preservar. No necesita todo el trace de producción.

Ejemplo de Incidente: Salto de Aprobación

Usa un incidente trabajado para probar el loop. Supón que un agent de soporte para reembolsos redactó un mensaje al cliente que implicaba que el reembolso estaba aprobado antes de que un líder de soporte aprobara el movimiento de dinero.

Paso de Revisión	Evidencia	Decisión
Incidente	El borrador dirigido al cliente decía “su reembolso ha sido aprobado” para una orden por encima del umbral de autoservicio.	La severidad es bloqueante porque el mensaje crea una expectativa financiera.
Revisión de trace	La route fue <code>refund_assist</code> ; existía el policy span, pero faltaba el approval span antes del draft span.	El failure mode es aprobación omitida, no mal tono.
Fixture mínimo	Ticket redactado, monto de la orden por encima del umbral, policy de reembolso actual, estatus estándar del cliente.	Conserva solo los datos necesarios para reproducir el límite.
Trayectoria esperada	<code>read_order</code> , <code>retrieve_refund_policy</code> , <code>draft_refund_request</code> , <code>approval_required</code> ; prohibir <code>issue_refund</code> y la promesa final al cliente.	El eval revisa la trayectoria antes que el texto.
Solución	Mover el approval gate antes de la redacción del mensaje al cliente y requerir un ID de recomendación aprobada en el paso de redacción.	Cambio de arquitectura, no solo ajuste de prompt.
Release gate	Ejecutar el fixture para cambios en prompt, policy, model-route, workflow y tool-manifest.	Bloqueante hasta que el fixture pase.
Canary	Monitorear en sombra el 5% de los borradores de reembolso y alertar si faltan approval spans.	Hacer rollback de la route si algún caso por encima del umbral carece de evidencia de aprobación.

El incidente debe producir tanto una aserción de trace como una aserción de redacción. La aserción de trace protege la autoridad. La aserción de redacción protege la promesa visible para el cliente.

```
{
  "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"
  ]
}
```

Si el fixture falla, no lo promedies en un puntaje de calidad. Detén el lanzamiento. Un sistema que redacta una promesa pulida no autorizada sigue siendo inseguro.

Triage de señales

No toda señal en producción debe convertirse en un eval bloqueante. Haz triage de las señales según el riesgo, recurrencia y significado arquitectónico.

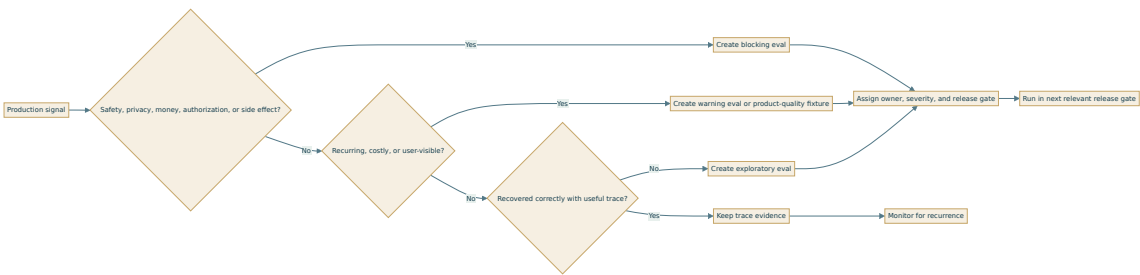
Señal	Decisión de triage	Acción de eval
Incidente de seguridad, privacidad, movimiento de dinero o autorización	Tratar como bloqueante hasta ser revisada.	Crear un fixture mínimo, agregar una trayectoria prohibida y requerir aprobación del responsable antes de liberar.
Corrección humana en una respuesta visible para el cliente	Revisar por calidad de producto y grounding.	Agregar un eval de advertencia a menos que la corrección exponga mal uso de policy, citation o tool.
Denegación de policy que un humano revierte	Verificar si la policy, el enrutamiento o la expectativa de UI era incorrecta.	Agregar fixtures de allow/deny emparejados cuando el límite era ambiguo.
Outlier de alto costo o alta latencia	Verificar si el agent loop, retrieval fanout o tool retry policy se desviaron.	Agregar verificaciones de presupuesto y condiciones de parada si el outlier puede repetirse.

Señal	Decisión de triage	Acción de eval
Error de tool recuperado correctamente	Mantener como evidencia en el trace.	Agregar un eval solo si la recuperación oculta una respuesta degradada o un retry repetido.
Casi error detectado por aprobación	Preservar el límite de aprobación.	Agregar un fixture que demuestre que el agent aún se detiene antes del efecto secundario.
Confusión de usuario aislada	Mejorar primero la UX o documentación.	Agregar un eval solo cuando la misma confusión cambie el comportamiento del agent.

La decisión de triage debe hacerse cerca del incidente, mientras el trace, la corrección humana y el contexto de negocio aún están frescos.

Flujo de triage de señales

Usa este flujo durante la revisión de incidentes. Evita que el equipo convierta cada señal en un eval bloqueante y aún así protege los límites importantes.



Contrato de eval fixture

Un eval derivado de un incidente debe ser lo suficientemente explícito para que un ingeniero nuevo entienda por qué existe y qué contaría como una regresión. Un fixture útil normalmente contiene cuatro grupos de campos:

Grupo	Campos
Identidad	<code>id</code> , <code>incident_id</code> , <code>owner</code> , <code>severity</code> , <code>created_at</code> , <code>expires_at</code> si es temporal.
Entradas	Solicitud de usuario redactada, evento de workflow, snapshot de state, evidencia recuperada, registros de memory, salidas simuladas de tools.
Trayectoria esperada	Tools requeridos, tools prohibidos, decisiones de policy requeridas, approval state, motivo de parada, comportamiento de retry.
Resultado esperado	Estado final, campos de structured output, requisitos de evidencia, restricciones de respuesta visible para el usuario.

No hagas que el fixture dependa de datos privados de producción a menos que esos datos estén redactados y se retengan intencionalmente. El fixture debe reproducir el modo de falla, no archivar el incidente para siempre.

Minimal Eval Runner

El eval runner de producción más simple y útil verifica la estructura y la trayectoria antes de revisar el texto. Este ejemplo es intencionalmente pequeño, pero muestra el límite al que este libro regresa: los agents necesitan pruebas sobre el camino que toman.

```
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
};
}

```

Este tipo de runner no te dirá si una respuesta para el cliente está bellamente redactada. Eso está bien. Su función es más limitada y arquitectónica: probar que el agent respetó el límite de liberación.

Mocked Tool Evals

Muchos fallos de agent ocurren mucho antes de la respuesta final, por eso los mocked tool evals son tan importantes para los agents que usan tools: te permiten probar la trayectoria sin tocar sistemas reales. Úsalos para verificar qué tools elige el agent y cuáles evita, si los argumentos son válidos, si se solicita aprobación, si los retries son seguros, si el agent se detiene ante una policy denial, si trata la salida de un tool no confiable como datos y si se recupera de respuestas mal formadas.

Los mocked tools no necesitan ser simulaciones perfectas. Deben ser lo suficientemente realistas para probar el límite de decisión. Si el agent llamaría a `issue_refund` cuando debería llamar a `draft_refund_request`, no necesitas un sistema de pagos real para detectar eso.

Para tools que cambian el state, el mock debe capturar la intención y los efectos secundarios por separado. Debe ser posible afirmar que el agent preparó una acción, solicitó aprobación o se detuvo en un límite de policy sin tocar el sistema real downstream.

Trajectory Evals

Los evals de respuesta final no son suficientes. Evalúa la trayectoria.

Capa de trayectoria	Qué verificar
Route	¿El sistema envió el task al workflow, model o agent correcto?
Context	¿El model recibió la evidencia mínima útil?
Retrieval	¿Las fuentes eran relevantes, actuales y permitidas?
Selección de tool	¿Se llamaron los tools esperados y se evitaron los prohibidos?
Entrada de tool	¿Los argumentos eran válidos, acotados y cumplían con la policy?
State	¿Las transiciones de state fueron correctas y reproducibles?
Memory	¿Las lecturas estaban justificadas y las escrituras limitadas?
Policy	¿Se aplicaron approvals, refusals y denials?

Capa de trayectoria	Qué verificar
Output	¿La respuesta final fue correcta, grounded y segura?

Una respuesta puede verse bien aunque el sistema se haya comportado mal. La trayectoria te dice si la arquitectura realmente funcionó.

Release Gates

Los lanzamientos de agents deben tener gates, igual que los lanzamientos de software, y el gate debe escalar con el riesgo del cambio.

Tipo de cambio	Gate sugerido
Redacción de prompt	Golden tasks, incident fixtures, verificaciones de trayectoria de tool.
Tool schema	Pruebas de schema, mocked tool evals, pruebas de autorización.
Policy rule	Pruebas de denegación, pruebas de aprobación, monitoreo canario.
Versión de model	Regression suite, presupuesto de costo y latencia, comparación de selección de tool.
Retrieval index	Relevancia de fuentes, frescura, cobertura de citation, comportamiento ante evidencia faltante.
Memory rule	Pruebas de privacidad, pruebas de memory obsoleta, verificaciones de policy de escritura.
Topología de agent	Finalización de task, costo de coordinación, completitud de trace, calidad de merge.

No todo eval debe bloquear cada release. Mantén evals bloqueantes para seguridad, privacidad, policy e incidentes conocidos; usa evals de advertencia para calidad, tono y edge cases; y mantén evals exploratorios para comportamientos nuevos aún en investigación. Los evals bloqueantes deben ser pocos, serios y mantenidos. Cuando todo bloquea, los equipos empiezan a ignorar el gate.

Un buen release gate responde tres preguntas:

1. ¿Qué cambió?
2. ¿Qué comportamientos podría afectar ese cambio?
3. ¿Qué subconjunto de eval protege esos comportamientos?

Si un cambio de prompt puede alterar la elección de tool, ejecuta verificaciones de trayectoria de tool. Si cambia el retrieval index, ejecuta verificaciones de grounding y evidencia faltante. Si cambia la ruta de model, compara costo, latencia, selección de tool, policy denials y calidad final. Un puntaje promedio genérico no es suficiente para decisiones de lanzamiento en producción.

Canary y rollback

Prompts, policies, rutas de model y reglas de tool son artifacts de producción, así que libéralos gradualmente:

1. sombrea el nuevo comportamiento donde sea posible;
2. envía un pequeño porcentaje del tráfico al candidato;
3. compara calidad, costo, latencia, errores de tool, policy denials y overrides humanos;
4. expande solo cuando los guardrails se mantienen;
5. haz rollback automáticamente cuando se superen los umbrales de seguridad o confiabilidad.

Un rollback debe restaurar el último prompt, policy, ruta de model, tool manifest o memory rule conocido como bueno. Si el rollback requiere reconstrucción manual, el proceso de release es débil.

Eval Ownership

Los evals necesitan responsables. Cada eval importante debe tener un responsable de negocio o de sistema, una etiqueta de severidad, una razón declarada para existir, una regla de mantenimiento, un enlace al incidente, requerimiento o riesgo que lo originó, y una decisión sobre si bloquea el release. Sin ownership, los eval suites se degradan: se vuelven lentos, poco confiables, redundantes y desconectados del riesgo en producción que debían monitorear.

El ownership también implica eliminación. Algunos evals deben expirar después de una migración, un policy sunset, el reemplazo de un tool o un incidente específico de cliente. Un eval obsoleto puede ser peor que no tener eval, porque bloquea cambios útiles mientras aparenta proteger un riesgo actual.

Metrics

Monitorea el feedback loop en sí, no solo la calidad del agent. Las señales útiles son las que indican si el loop funciona: tasa de conversión de incident-to-eval, tasa de detección de eval antes del release, tasa de recurrencia de incidentes conocidos, tiempo desde el incidente hasta el regression test, número de evals bloqueantes, tasa de evals poco confiables, cobertura de production trace, precisión de policy-denial, tasa de human-override, frecuencia de rollback y tiempo promedio para detectar una regresión del agent. El objetivo no es un dashboard bonito. Es saber si el sistema aprende de los fallos.

Dashboard Thresholds

Usa thresholds para decidir cuándo el loop necesita atención. Empieza simple y ajusta los números cuando el equipo tenga tráfico real.

Metric	Investigar Cuando	Bloquear o Hacer Rollback Cuando
Incident-to-eval conversion	Incidentes serios no generan fixture en dos días hábiles.	Un incidente repetido no tiene regression eval.
Eval catch rate	Incidentes en producción pasan la suite más de una vez por ciclo de release.	Un fixture de incidente conocido pasa mientras producción repite la misma falla.
Flaky eval rate	Más del 5% de los runs de release-gate requieren rerun.	Un eval bloqueante es poco confiable y nadie se encarga de repararlo.

Metric	Investigar Cuando	Bloquear o Hacer Rollback Cuando
Production trace coverage	Menos del 95% de los runs riesgosos tienen spans completos de route, tool, policy y stop-reason.	Un release cambia comportamiento riesgoso sin trace coverage.
Policy-denial accuracy	Los operadores frecuentemente anulan denegaciones o los usuarios reportan rechazos falsos.	El agent ejecuta un side effect después de una policy denial.
Human-override rate	Los overrides aumentan después de cambios en prompt, model, retrieval o policy.	Los overrides exponen aprobación omitida, evidencia obsoleta o elección incorrecta de tool.
Rollback frequency	Los rollbacks se agrupan en el mismo componente.	El mismo componente causa dos eventos de rollback sin un nuevo gate.

Los dashboard thresholds deben asignar un owner y una acción. Un gráfico en rojo sin regla de decisión es solo decoración.

Anti-Patterns

- Tratar los evals como un demo score en vez de un control de release.
- Ejecutar solo checks de respuesta final en un sistema que usa tools, memory, policy o retrieval.
- Mantener incident fixtures que nadie posee ni entiende.
- Permitir upgrades de model sin reproducir fallos conocidos en producción.
- Usar un solo global average score para aprobar cambios de comportamiento de alto riesgo.
- Capturar production traces que no pueden almacenarse, buscarse o reproducirse de forma segura.
- Permitir que evals poco confiables sigan bloqueando hasta que los ingenieros ignoren el gate.

Practical Workflow

En la práctica, el loop funciona así:

1. Un run falla, escala o recibe una corrección humana.
2. El trace se revisa y redacta.
3. El equipo identifica el modo de falla.
4. Se crea un eval fixture mínimo.
5. Se confirma que el sistema actual falla el fixture.
6. Se realiza un fix en prompt, tool, policy, retrieval, memory o arquitectura.
7. El eval pasa sin romper casos bloqueantes.
8. El cambio se despliega vía canary.
9. El monitoreo en producción confirma que la falla no se repite.

Así es como la ingeniería de agents se vuelve acumulativa. Cada falla seria debe hacer que la siguiente versión sea más difícil de fallar de la misma manera.

Design Checklist

Antes de operar un agent en producción, responde:

- ¿Qué traces se capturan?
- ¿Qué traces son seguros de almacenar?
- ¿Qué fallas se convierten en evals?
- ¿Quién es owner de los evals derivados de incidentes?
- ¿Qué evals bloquean el release?
- ¿Qué evals solo advierten?
- ¿Se pueden mockear los tools?
- ¿Se pueden reproducir los runs?
- ¿Se pueden hacer rollback de prompts, policies, tools y rutas de model?
- ¿Se puede detener automáticamente un canary?
- ¿Pueden los operadores explicar por qué se bloqueó un release?

Si los fallos en producción no alimentan evals, la observability es pasiva. Te dice qué pasó, pero no mejora el sistema.

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](#)

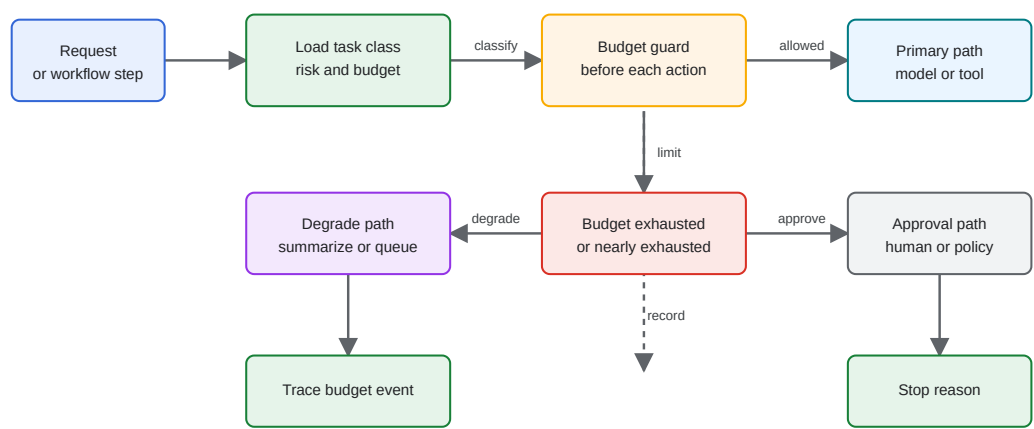
Cost Controls and Runtime Budgets

Los agentic systems gastan dinero, tiempo, context, cuota de tool y atención humana cada vez que piensan, recuperan, llaman a un tool, delegan, reintentan, evalúan o piden aprobación. Los runtime budgets hacen que esos costos sean explícitos y aplicables.

Descarga la [lista de verificación para revisión de cost controls y runtime budgets](#) antes de usar este capítulo para una revisión de producción.

Este capítulo no trata de hacer agents baratos a cualquier costo. Se trata de hacer que el costo, la latencia y la autonomía sean parte del control plane. Un agent útil debe saber cuándo continuar, cuándo degradar, cuándo pedir aprobación y cuándo detenerse.

Cost Controls and Runtime Budgets



Intent

Mantener el comportamiento agentic limitado por el valor del task, el riesgo, el nivel de usuario y los límites operativos.

Un budget no es solo un número financiero. Es un contrato en runtime: cuántos pasos puede dar el sistema, cuántos tools puede llamar, cuánto tiempo puede ejecutarse, cuánta context puede transportar, cuántos retries puede gastar, cuánta atención humana puede solicitar y cuánta autonomía está justificada antes de escalar.

Runtime Budget Readiness Questions

Usa estas preguntas antes de habilitar loops autónomos, retries, retrieval, delegación o tools de escritura:

Pregunta	Evidencia a presentar
¿Qué justifica gastar el task?	Policy de budget por clase de task, clase de riesgo, nivel de usuario y valor de negocio.

Pregunta	Evidencia a presentar
¿Qué mide el runtime?	Contadores para llamadas de model, llamadas de tool, retries, retrieval, delegación, tiempo de reloj y costo.
¿Dónde se aplican los budgets?	Puertas previas a la acción antes de los pasos de model, tool, retrieval, memory, aprobación y delegación.
¿Qué sucede cuando el budget es bajo?	Modos degradados definidos, mensaje para el usuario y razón de detención para el operador.
¿Cuándo se requiere aprobación?	Umbrales para gasto extra, tools de escritura, brechas de evidencia de alto riesgo o tasks de alto riesgo.
¿Cómo se revisan los cambios de budget?	Policy versionada, casos de eval, comparación de traces y ruta de rollback.

El budgeting es una característica de la arquitectura. Si el budget es solo un dashboard después de la ejecución, es contabilidad, no control.

Usar cuando

- Las ejecuciones de agent pueden hacer loop, retry, delegar, recuperar o llamar tools.
- Las llamadas de model, tool, retrieval o evaluator tienen costo o latencia significativa.
- Diferentes clases de task merecen diferentes niveles de autonomía.
- El sistema atiende usuarios interactivos y debe preservar la capacidad de respuesta.
- Los operadores necesitan razones claras de detención cuando una ejecución no puede continuar.

Usa este pattern antes de que aparezcan sorpresas de costo en producción. Adaptar budgets después de un loop fuera de control es más difícil porque la arquitectura ya ocultó los contadores.

Evitar cuando

- El task es una sola llamada determinista de función.
- El sistema no puede observar llamadas de model, llamadas de tool, retries o tiempo de reloj.
- La organización quiere reducción de costos sin definir calidad y riesgo aceptables.

Aun así, mantén contadores simples. El primer prototipo útil suele convertirse en el primer workflow de producción.

Tipos de Budget

Budget	Qué controla
Token budget	Prompt, context, evidencia recuperada, memory y espacio reservado para output.
Model-call budget	Número de llamadas de model, nivel de model, llamadas de judge y intentos de reparación.

Budget	Qué controla
Tool-call budget	Conteo de tools, operaciones de escritura, acciones de navegador, comandos de shell y cuota de API externa.
Wall-clock budget	Tiempo de extremo a extremo, tiempo por paso, tiempo en cola y tiempo de espera humana.
Retry budget	Cuántas veces un paso puede recuperarse antes de que el sistema cambie de estrategia.
Delegation budget	Número de agents, handoffs, debates o ramas paralelas.
Retrieval budget	Conteo de queries, conteo de fuentes, costo de reranking y volumen de evidencia.
Memory budget	Lecturas y escrituras de memory, retención y recall sensible a privacidad.
Approval budget	Cuántas aprobaciones humanas, revisiones o escaladas puede solicitar un workflow.

El budget más importante a menudo no es el dinero. En producción, la atención humana y la paciencia del usuario suelen ser más escasas que los tokens.

Propiedad del Budget

Los budgets deben ser propiedad del runtime, no del prompt. El model puede proponer que más trabajo es útil, pero el software debe decidir si el sistema puede gastar más.

La propiedad del budget normalmente pertenece a la misma capa que posee el state, policy, trace IDs y razones de detención:

- los routers cargan el budget para la clase de task;
- los loop controllers descuentan el budget después de cada paso;
- los tool gateways revisan los budgets de tool y efectos secundarios;
- los retrieval services aplican budgets de evidencia y queries;
- los workflow engines persisten el state del budget a través de retries y aprobaciones;
- observability registra eventos de budget como eventos de trace de primera clase.

Si cada wrapper de tool lleva sus propios contadores, el sistema se desincroniza. El runtime necesita una sola vista de budget para la ejecución.

Runtime Budget Object

Dale a cada ejecución un objeto de budget que viaje con el 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;
};

```

El objeto de budget debe estar versionado. Un incidente de producción causado por un cambio de budget debe poder reproducirse contra la policy de budget anterior y la nueva.

Enforcement

Revisa los budgets antes de cada acción costosa o riesgosa, no después de que la ejecución ya terminó.

```

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' };
    }

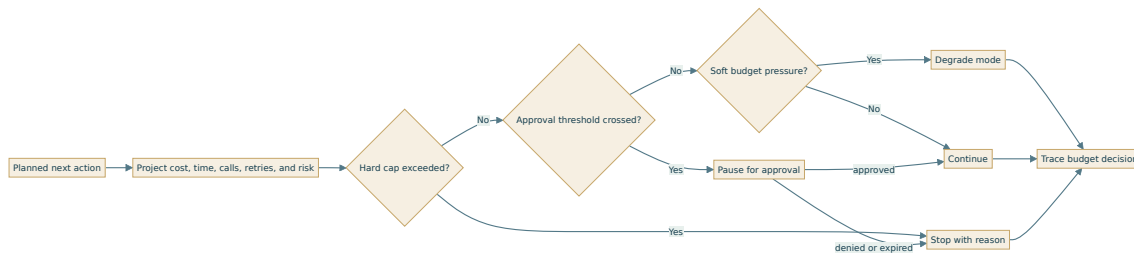
    return { decision: 'continue', reason: 'within_budget' };
}

```

La decisión debe formar parte del trace. Nota que la revisión usa el costo proyectado, no solo el uso actual. Una puerta de budget que corre después de la llamada de model o tool ya gastó el dinero y es solo contabilidad. Una puerta de budget que corre antes de la siguiente acción es control.

Flujo de Decisión de Budget

Usa este flujo antes de cada llamada de model, query de retrieval, llamada de tool, delegación, retry, solicitud de aprobación o escritura en memory. Convierte la presión de budget en una decisión explícita en runtime.



Degraded Modes

Cuando un presupuesto está casi agotado, el sistema no debe improvisar. Debe cambiar a un modo degradado conocido:

Presión de Presupuesto	Modo Degradado Más Seguro
El token budget es bajo	Resumir el state y preservar referencias de evidencia.
El model-call budget es bajo	Detener revision loops y devolver el mejor resultado validado.
El tool-call budget es bajo	Pasar de acción a borrador o análisis solo de lectura.
El wall-clock budget es bajo	Devolver resultado parcial o poner trabajo en cola en segundo plano.
El retry budget es bajo	Dejar de reintentar y exponer el bloqueo.
El delegation budget es bajo	Asignar un owner en lugar de agregar más agents.
El retrieval budget es bajo	Pedir aclaración o citar evidencia faltante.
El approval budget es bajo	Agrupar decisiones o escalar a una cola humana.

El modo degradado debe ser visible para el usuario u operador. La degradación silenciosa genera una falsa confianza.

Risk-Based Budgets

No toda task merece el mismo presupuesto.

Las tasks de bajo riesgo pueden ser baratas y rápidas. Las de alto riesgo pueden justificar modelos más robustos, más evidencia, más policy checks y aprobación humana. El error es dar a cada task el mismo presupuesto porque el runtime no tiene task class.

Task Class	Postura de Presupuesto
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.
Money movement	Write-tool budget, approval threshold, policy check, audit trace.
Multi-agent analysis	Delegation budget, merge budget, coordination trace, final evaluator.

Task Class	Postura de Presupuesto
Background research	Longer wall-clock budget, queue support, checkpointing.

Los presupuestos son una forma de codificar el juicio. Indican qué tasks valen más razonamiento, más tools, más revisión o más tiempo.

Budget Calculator Example

Comienza con una estimación simple por ejecución antes de construir un dashboard complejo.

Ítem	Cantidad	Estimado Unitario	Estimado por Ejecución
planner model call	1	1.5 centavos	1.5 centavos
retrieval queries	3	0.2 centavos	0.6 centavos
synthesis model calls	2	2.0 centavos	4.0 centavos
tool calls	4	0.5 centavos	2.0 centavos
evaluator call	1	1.0 centavos	1.0 centavos
retry reserve	20%	9.1 centavos base	1.8 centavos
total planned budget			10.9 centavos

Luego mapea la estimación a una policy:

Task Class	Límite Predeterminado	Aprobación Por Encima De	Modo Degradado
read-only support answer	15 centavos	25 centavos	respuesta con evidencia citada y sin retrieval extra
refund recommendation	50 centavos	75 centavos	recomendación en borrador y solicitar revisión humana
multi-agent incident analysis	\$2.00	\$3.00	detener fan-out y asignar un owner
background research	\$5.00	\$8.00	poner en cola la continuación y devolver hallazgos parciales

Los números exactos variarán según el model y el proveedor. El hábito útil es el mismo: estimar antes de la ejecución, reservar para retries, hacer cumplir antes de acciones y registrar el gasto real contra la versión de la policy.

Alert Thresholds

Las alertas de presupuesto deben distinguir entre una ejecución costosa y un problema sistémico.

Señal	Advertencia	Crítico	Acción
per-run cost	80% del límite	100% del límite	degradar o detener antes de la siguiente acción
model calls per run	80% del límite	límite alcanzado	detener revision loop o requerir aprobación
write-tool calls	cualquier escritura inesperada	límite alcanzado	pausar ruta de escritura y revisar trace
retry rate	10% de ejecuciones reintentan	25% de ejecuciones reintentan	inspeccionar el paso fallido y agregar eval case
delegation fan-out	por encima del número planeado de workers	creación descontrolada de workers	detener delegación y asignar owner
queue spend rate	2x gasto horario normal	5x gasto horario normal	aplicar backpressure o pausar ruta
approval requests	2x volumen normal	cola de revisores bloqueada	agrupar, enrutar o pausar trabajo de baja prioridad
degraded-mode rate	5% de ejecuciones	15% de ejecuciones	revisar presupuestos, prompts, tools o enrutamiento de tasks

Cada alerta debe señalar una acción para el operador. Una alerta de costo que solo dice “el gasto es alto” es demasiado vaga para ayudar durante un incidente.

Interaction With Circuit Breakers

Los budgets y breakers están relacionados pero no son idénticos.

Un budget indica cuánto puede gastar el sistema. Un breaker indica cuándo el sistema debe detenerse o cambiar de estrategia porque el patrón de comportamiento es inseguro o improductivo.

Ejemplos:

- un cost budget detiene la ejecución después de alcanzar el límite de gasto;
- un no-progress breaker detiene la ejecución incluso si queda presupuesto;
- un tool budget limita llamadas a una clase de tool;
- un tool failure breaker deshabilita una tool tras resultados repetidamente malos;
- un delegation budget limita handoffs;
- un handoff breaker impide que los agents reboten trabajo sin owner.

Usa ambos. Una ejecución puede estar dentro del presupuesto y aun así ser insegura. Una ejecución puede ser segura pero ya no justificar el gasto.

Observability

Los eventos de presupuesto deben ser visibles en los traces:

- versión de la budget policy;
- task class y risk class;
- presupuesto inicial;
- presupuesto restante después de cada paso;
- verificaciones de presupuesto antes de llamadas a model y tool;
- decisión: continuar, degradar, requerir aprobación o detener;
- stop reason;
- modo degradado visible para el usuario;
- explicación visible para el operador.

Si el costo solo es visible en la factura mensual, la arquitectura ya llegó tarde.

Evaluation Guidance

Prueba el comportamiento de presupuesto como cualquier otro control de producción.

Eval Case	Comportamiento Esperado
Model-call budget agotado	Revision loop se detiene o degrada con razón rastreable.
Write-tool budget agotado	El agent cambia de acción a borrador o ruta de aprobación.
Umbral de costo cruzado	Se requiere aprobación humana antes de continuar.
Delegation budget agotado	El supervisor asigna un owner o detiene con estado bloqueado.
Wall-clock budget agotado	El workflow pone trabajo en cola o devuelve resultado parcial.
Low-value task solicita ruta costosa	El router elige un camino seguro más barato.
High-risk task necesita más evidencia	El runtime permite retrieval extra pero requiere policy checks más estrictos.

El eval debe revisar tanto el resultado como la trayectoria. Una respuesta final que se ve bien pero excedió el presupuesto sin aprobación no es válida.

Failure Modes

- Los budgets solo viven en prompts, así que el model puede ignorarlos o reinterpretarlos.
- Los budgets se verifican solo después de que ya ocurrió trabajo costoso.
- Se usa el mismo presupuesto para classification de bajo riesgo y automation de alto riesgo.
- El costo se optimiza mientras aumenta el trabajo de revisión humana.
- Retry budgets ocultan fallas repetidas en vez de cambiar de estrategia.
- Faltan delegation budgets, así que los multi-agent systems generan costo de coordinación.
- El agotamiento del presupuesto se registra como un error genérico sin stop reason.
- Los operadores no pueden comparar versiones de budget policy durante la revisión de incidentes.

Production Checklist

- ¿Cada ejecución inicia con task class, risk class y versión de budget policy?
- ¿Se verifican los budgets antes de acciones de model, tool, retrieval, memory y delegation?
- ¿Las decisiones de presupuesto se registran como eventos de trace?
- ¿Los modos degradados están definidos antes del lanzamiento?
- ¿Las ejecuciones de alto riesgo pueden solicitar aprobación para más presupuesto?
- ¿Las ejecuciones de bajo riesgo pueden detenerse sin revisión humana?
- ¿Los budgets están ligados a evals, incidentes y gates de lanzamiento?
- ¿Se pueden revertir cambios en la budget policy?
- ¿Los operadores pueden responder a dónde se fue el costo de una ejecución?

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

La enforcement de policy limita lo que el agent puede decir o hacer mediante permisos, reglas de acceso a datos, reglas de negocio, reglas de seguridad y escalamiento.

Fuente y descargas

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

Intent

La enforcement de policy es el límite controlado por software que decide si una respuesta propuesta por el model, una llamada a tool, un acceso a datos, una escritura en memory o un efecto secundario está permitido. El model puede proponer una acción. El runtime decide si la permite, la niega, requiere aprobación, la escala o la audita.

La policy no debe vivir solo en el texto del prompt. Los prompts pueden explicar la policy al model, pero la enforcement pertenece al código, workflow, manifiestos de tools, sistemas de control de acceso y registros de decisiones auditables.

Los agents orientados al conocimiento usan la misma idea para las respuestas: el model solo debe responder desde fuentes aprobadas, citar la evidencia y rechazar o escalar cuando la fuente requerida falta, está desactualizada, prohibida o en conflicto.

La regla práctica es: la policy se ejecuta antes de la autoridad. Antes de la recuperación, antes de las escrituras en memory, antes de la ejecución de tools, antes de la comunicación externa y antes de las respuestas finales en dominios regulados o basados en evidencia, el runtime debe saber si la acción está permitida.

Usar cuando

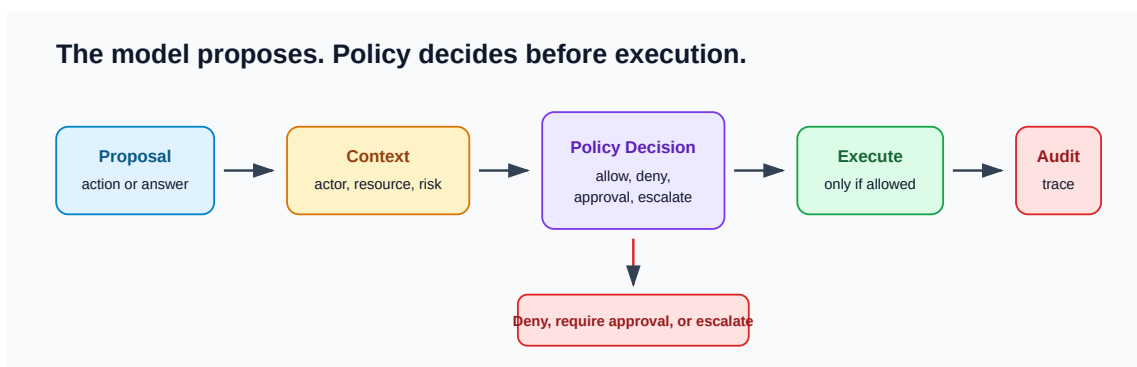
- Las acciones deben ser verificadas antes de ejecutarse.
- El agent maneja datos privados, regulados, sensibles a la seguridad o críticos para el negocio.
- El sistema necesita fuentes aprobadas, citas o restricciones de cumplimiento.
- Las decisiones de policy deben ser auditables y reproducibles.
- El runtime puede identificar actor, recurso, acción, capability, riesgo y context.
- Las llamadas a tools, escrituras en memory, recuperación, respuestas finales o transiciones de workflow requieren reglas diferentes según la clase de task.
- Existe una ruta de aprobación humana para acciones válidas pero demasiado riesgosas para ejecutarse de forma autónoma.

Evitar cuando

- La policy solo está escrita como guía en el prompt sin verificación en runtime.
- El sistema no puede identificar el actor, recurso, acción y context.
- Las verificaciones de policy ocurren después de acciones irreversibles.
- Las excepciones son silenciosas, no revisadas o faltan en los traces.
- Las fuentes de conocimiento aprobadas no pueden ser identificadas, actualizadas o citadas.
- El runtime no puede detener, pausar o cambiar la ejecución después de una decisión de policy.

Arquitectura

Usa este diagrama para leer Policy Enforcement como un límite de sistema, no solo como una forma de código. La pregunta clave de propiedad es: el runtime es dueño del state durable, reintentos, traces, triggers, configuración de despliegue y controles operativos.



Forma del sistema

- **Límite del pattern:** el límite de policy evalúa acciones propuestas, acceso a datos, respuestas y escrituras en memory antes de que tengan efecto.
- **Propietario del state:** el runtime es dueño del policy context, registros de decisión, versión de policy, trace ID y resultado de enforcement.
- **Rol del model:** el model propone una acción o respuesta y puede explicar el riesgo, pero no se otorga permiso a sí mismo.
- **Límite de conocimiento:** fuentes aprobadas, vigencia, citas y reglas de rechazo definen lo que el agent puede afirmar.
- **Límite de presupuesto:** la policy puede requerir aprobación, degradar capability o detenerse cuando una ejecución ha excedido el gasto o autonomía permitidos para su clase de riesgo.
- **Promesa operativa:** las decisiones de policy ocurren antes de la ejecución y son visibles después de la ejecución.

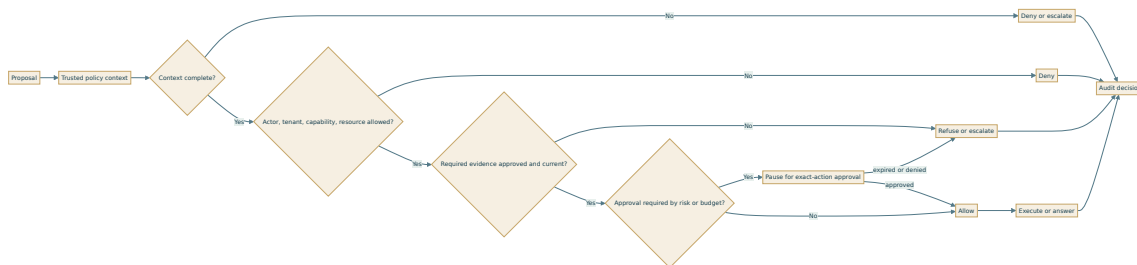
Protocolo central

1. Recibir una acción propuesta, respuesta, llamada a tool, resultado de recuperación o escritura en memory.
2. Construir el policy context: actor, caller, tenant, recurso, capability, riesgo, evidencia y trace ID.
3. Evaluar la policy antes de que la acción se ejecute o la respuesta se devuelva.
4. Retornar una decisión: permitir, negar, requerir aprobación, escalar o solo auditar.
5. Si se requiere aprobación, pausar en la puerta de aprobación.
6. Ejecutar solo las decisiones permitidas o aprobadas.

7. Registrar decisión, motivo, versión de policy, actor, recurso, acción y trace ID.
8. Alimentar negaciones graves, omisiones y sobrescrituras en los regression evals.

Notas de implementación

Usa este flujo de decisión al revisar dónde se ejecuta la policy. Cada rama debe producir un evento de trace con la versión de policy, motivo, actor, recurso, capability y efecto de ejecución.



Una decisión de policy debe ser un objeto tipado en runtime.

```

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;
};

```

El policy context debe provenir de state confiable en runtime, no solo del texto del model:

```

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;
};

```

La función de enforcement debe ejecutarse antes de la recuperación, escritura en memory, llamada a tool, efecto secundario o respuesta final:

```

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' };
}

```

Para respuestas orientadas al conocimiento, la policy también decide si la evidencia está permitida:

```

type SourcePolicy = {
  sourceId: string;
  approved: boolean;
  freshness: 'current' | 'stale' | 'unknown';
  citationRequired: boolean;
  allowedTenant?: string;
};

```

El model puede explicar por qué una acción parece segura. El runtime aún toma la decisión.

Dónde se ejecuta la policy

Boundary	Policy Question
Retrieval	¿Este actor tiene permitido leer estas fuentes para este task?
Tool call	¿Este tool está permitido para el actor, tenant, recurso, riesgo y presupuesto?
Memory write	¿Esta memory es segura, acotada, útil y permitida para persistir?
Human approval	¿Esta acción solo está permitida después de revisión, y quién puede aprobarla?
Final answer	¿La respuesta está respaldada por evidencia aprobada y es segura para devolver?
Workflow transition	¿El siguiente paso es válido después del current state y la decisión de policy?

Trata cada decisión de policy como un evento en runtime. Debe tener un trace ID, versión de policy, resumen de entrada, decisión, motivo y efecto de ejecución.

Failure Modes

- La policy existe solo en el system prompt.
- La policy se ejecuta después de que un tool ya fue ejecutado.
- La decisión carece de actor, resource, tenant o capability context.
- Un retry omite la policy porque el primer intento ya fue verificado.
- Las versiones de la policy cambian pero los traces no registran qué versión se aplicó.
- Los denials no se registran, por lo que los operadores no pueden ver acciones inseguras intentadas.
- Las acciones que requieren aprobación se tratan como permitidas.
- Las respuestas de knowledge citan fuentes no aprobadas, obsoletas o inaccesibles.
- Las exceptions se convierten en huecos permanentes y no documentados en la policy.

- Las verificaciones de policy ignoran el budget state, por lo que un agent puede seguir gastando cuando ya debería requerirse aprobación.
- Las escrituras en memory omiten la policy porque se tratan como gestión inofensiva de context.
- Las respuestas finales omiten la policy incluso cuando el dominio requiere evidencia aprobada.

Evaluation Strategy

Las policy evals deben probar caminos permitidos, denegados, que requieren aprobación y de escalamiento.

- Prueba acciones permitidas de bajo riesgo.
- Prueba acciones denegadas a través de límites de role, tenant, resource y capability.
- Prueba acciones que requieren aprobación antes de efectos secundarios.
- Prueba fuentes obsoletas o no aprobadas en respuestas ligadas a knowledge.
- Prueba retries y workflows reanudados para asegurar que la policy se aplique cada vez.
- Prueba ausencia de actor, resource o tenant context.
- Prueba cambios de versión de policy y la completitud de auditoría.
- Prueba incidentes de producción como policy fixtures reproducibles.
- Prueba casos de threshold de budget que requieren aprobación antes de continuar.
- Prueba denegaciones, aprobaciones y alcance de tenant en escrituras de memory.
- Prueba rechazo o escalamiento de respuesta final cuando falta evidencia requerida.

Un policy eval compacto puede verse así:

```
{
  "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"]
  }
}
```

Mide la precisión de decisiones de policy, tasa de false allow, tasa de false denial, precisión en el enrutamiento de aprobación, violaciones de tenant-boundary, uso de stale-policy, completitud de denial logging y recurrencia de fallas conocidas de policy.

Para sistemas en producción, el false allow suele ser la métrica más peligrosa. Un false denial puede frustrar a un usuario. Un false allow puede filtrar datos, mover dinero, enviar el mensaje incorrecto o crear un incidente.

Production Checklist

- Aplica la policy antes de la ejecución, retorno de respuesta, escritura en memory o comunicación externa.
- Construye el policy context a partir de datos confiables del runtime, no solo texto del model.
- Devuelve decisiones explícitas de allow, deny, require-approval, escalate o audit.
- Registra actor, resource, capability, reason, policy version y trace ID.
- Aplica la policy en retries y workflows reanudados.
- Mantén versionadas las policy versions, tool manifests, source rules y approval rules.
- Trata la ausencia de policy context como deny o escalate.
- Aplica la policy a escrituras en memory y respuestas finales, no solo a tools.
- Conecta las decisiones de policy al budget state del runtime.
- Agrega dashboards para denials, approvals, overrides y false allows.
- Convierte policy misses en regression evals.
- Revisa las exceptions y expíralas intencionalmente.

Code Walkthrough

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo que lo rodea explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

Source Code

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el source del repositorio.

`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
    };
}

```

Extracto truncado para mayor legibilidad. Descarga el bundle o abre el archivo fuente para la implementación completa.

Download

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

El bundle de descarga contiene la carpeta `compliance-policy-enforcer-agent/` actual de este repositorio.

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

Los event-triggered agents se ejecutan en respuesta a webhooks, colas, programaciones o eventos de dominio.

Fuente y descargas

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

Propósito

Los event-triggered agents se ejecutan en respuesta a webhooks, colas, programaciones o eventos de dominio.

Usa este pattern cuando el sistema debe reaccionar a trabajo que llega sin que un usuario en vivo esté viendo la pantalla. El event runtime, no el model, es responsable de la admisión, deduplicación, idempotencia, reintentos, dead letters, replay y controles de tormenta.

Escenario

Una plataforma de soporte emite `refund.requested` cuando un cliente abre un caso de reembolso. El agent debe revisar la policy, resumir el historial de pedidos, redactar una recomendación y notificar a un revisor. No debe procesar el mismo reembolso dos veces si la cola vuelve a entregar el evento. No debe actuar sobre un evento obsoleto o no autorizado. Debe detenerse de forma segura si la fuente de policy no está disponible.

Esta es la diferencia entre un event-triggered agent y un chat agent. El agent puede ejecutarse a las 03:00 sin que haya un usuario presente. El runtime debe decidir si acepta, retrasa, rechaza, reintenta o envía a dead letter el evento antes de que el model proponga algo.

Usar cuando

- Un evento de dominio, webhook, mensaje de cola, programación o llegada de archivo debe iniciar trabajo acotado de un agent.
- La task puede describirse como un tipo de evento mapeado a una clase de task.
- La entrega duplicada, los reintentos y los eventos tardíos pueden manejarse de forma segura.
- El evento puede portar u obtener suficiente context autorizado.
- El sistema puede trazar event ID, run ID, idempotency key, policy decision, tool calls y stop reason.

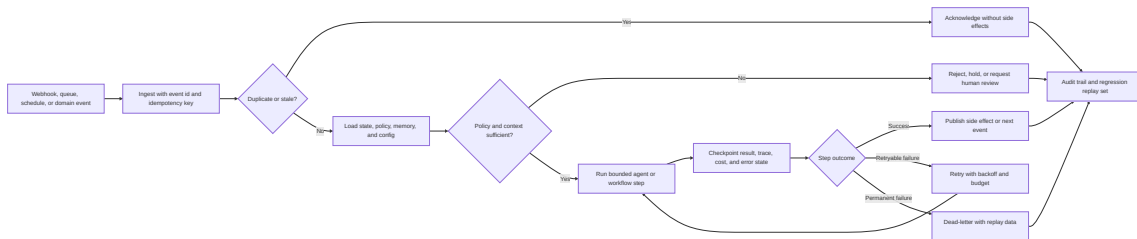
Evitar cuando

- El payload del evento no tiene suficiente información de identidad, tenant, recurso o versión para actuar de forma segura.
- La entrega duplicada podría repetir efectos secundarios irreversibles.

- Los requisitos de orden no están claros.
- Las fallas no pueden ser reintentadas, compensadas o enviadas a dead letter.
- Un humano debe inspeccionar el context antes de la primera acción.

Arquitectura

Usa este diagrama para leer Event-Triggered Agents como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el runtime es dueño del state durable, reintentos, traces, triggers, configuración de despliegue y controles operativos.

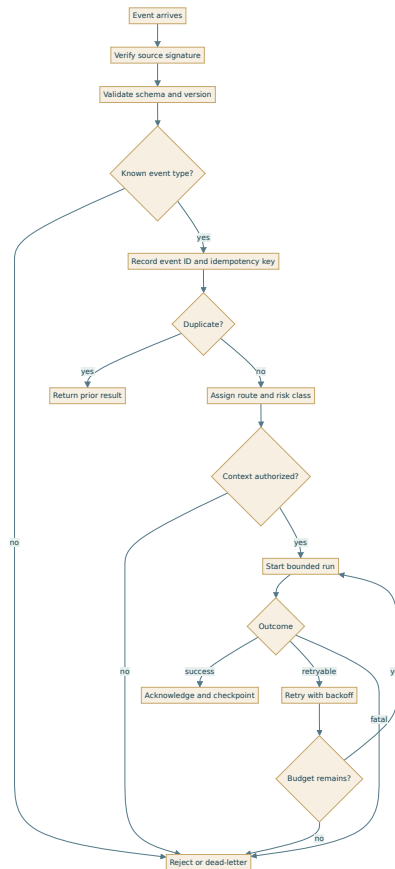


Léelo como un límite de runtime desatendido: event identity, idempotency, policy, checkpoints, reintentos, dead letters y replay evidence protegen cada efecto secundario.

Reglas de decisión

Pregunta	Respuesta requerida	Falla si falta
¿Qué tipo de evento inicia el trabajo?	Nombre y versión de evento estables.	Una ruta maneja payloads incompatibles.
¿Qué recurso se ve afectado?	Resource ID, tenant, actor y correlation ID.	Acción cruzada de tenant o recurso incorrecto.
¿Qué hace único al evento?	Event ID más source, tenant o resource scope.	La entrega duplicada repite el trabajo.
¿Qué orden importa?	Ninguno, por recurso, por tenant o global.	Los eventos tardíos sobrescriben un state más nuevo.
¿Qué efectos secundarios pueden ocurrir?	Leer, redactar, notificar, escribir o ejecutar.	El reintento duplica acciones irreversibles.
¿Qué es reintetable?	Clases de error, intentos máximos, backoff y regla de dead letter.	El sistema descarta trabajo o reintenta para siempre.
¿Quién revisa las fallas?	Dueño de dead letter y frecuencia de revisión.	Los eventos rotos se vuelven backlog invisible.

Flujo de admisión de eventos



Forma del sistema

- **Límite del pattern:** un servicio de producción o framework aloja el agent detrás de límites de durable workflow, policy, observability y despliegue.
- **Dueño del state:** el runtime es dueño de la recepción de eventos, deduplicación, run state, reintentos, dead letters, replay, traces, configuración de despliegue y controles operativos.
- **Artifact principal:** [event-triggered-agent-pattern/](#) documenta el límite revisable de evento a agent y los controles del runtime.
- **Promesa operativa:** un evento aceptado crea como máximo una ejecución autorizada y trazable para el recurso afectado.

Contrato

Un event-triggered agent necesita un event envelope antes de necesitar un prompt.

Campo	Propósito
<code>eventId</code>	Identidad para deduplicación y replay.
<code>eventType</code>	Enruta el evento a una clase de task acotada.
<code>eventVersion</code>	Protege contra cambios incompatibles en el payload.
<code>source</code>	Nombra el origen del webhook, cola, programación o servicio.
<code>actorId</code>	Registra quién o qué causó el evento cuando se conoce.

Campo	Propósito
<code>tenantId</code>	Hace cumplir el límite de tenant antes de la recuperación o tools.
<code>resourceId</code>	Identifica el pedido, ticket, cuenta, archivo o registro afectado.
<code>occurredAt</code> y <code>receivedAt</code>	Soportan decisiones de eventos tardíos y de orden.
<code>correlationId</code>	Conecta solicitud upstream, evento, run, tools y traces.
<code>idempotencyKey</code>	Previene efectos secundarios duplicados en reintentos.
<code>payload</code>	Transporta datos validados de la task, no instrucciones confiables.

Protocolo central

1. Verifica source del evento, schema, versión, identidad, tenant y resource scope.
2. Registra event ID, correlation ID e idempotency key antes de efectos secundarios.
3. Enruta el evento a una clase de task acotada con nivel de riesgo y policy context.
4. Obtén el context requerido a través de tools autorizados en vez de confiar en el texto del payload.
5. Ejecuta un agent o paso de workflow acotado.
6. Haz checkpoint de resultado, trace data, costo y estado de error.
7. Reconoce, reintenta, compensa, continúa, rechaza, escala o envía a dead letter según la policy del runtime.

Notas de implementación

- Trata los payloads de eventos como entrada no confiable. Un cuerpo de webhook puede describir una task, pero no debe sobrescribir policy, ruta, tools o reglas de memory.
- Registra el idempotency key antes de llamar a un tool con capacidad de escritura.
- Define el orden por recurso. La mayoría de los sistemas necesitan orden por ticket, pedido o cuenta, no orden global.
- Usa verificaciones de versión optimistas o locks cuando dos eventos pueden afectar el mismo recurso.
- Mantén presupuestos de reintentos separados para llamadas al model, retrieval, tools y reprocesamiento de eventos completos.
- Agrega un ciclo de revisión de dead letter. Un dead letter que nadie revisa es pérdida de datos silenciosa.
- Haz que el replay sea consciente de la versión. Reproducir un evento antiguo contra una nueva policy, prompt o versión de tool debe ser explícito.

Modos de falla

- Efecto secundario duplicado: la redelivería repite un reembolso, correo, actualización de ticket o cambio de infraestructura.
- Event storm: un bug upstream inunda el agent y consume el presupuesto de model/tool.
- Poison event: un payload mal formado u hostil llega al context del model como instrucción.
- Sobrescritura por evento tardío: un evento antiguo actualiza el state después de una decisión más reciente.

- Brecha de context: el evento carece de identidad, tenant, recurso o context de autorización.
- Bucle de reintento: un error irre recuperable se trata como reintetable.
- Negligencia de dead letter: los eventos fallidos se acumulan sin dueño, alerta o ruta de replay.
- Replay drift: un evento reproducido usa una policy o prompt diferente sin registrar el cambio de versión.

Lista de revisión

Antes de producción, verifica:

- Event type, versión, source, actor, tenant, resource y correlation ID están registrados.
- Las versiones de evento no soportadas fallan de forma segura.
- La entrega duplicada retorna un resultado previo o reanuda el mismo run.
- El idempotency key se registra antes de escrituras externas.
- Las reglas de orden y concurrencia son explícitas.
- Los errores reintetables y fatales están clasificados.
- La dead-letter queue tiene dueño, alerta, frecuencia de revisión y procedimiento de replay.
- Los controles de event storm pueden pausar, descartar, muestrear o enrutar a fallback determinista.
- El trace vincula event ID, run ID, policy decision, tool calls y stop reason.

Estrategia de evaluación

- Prueba evento válido, evento duplicado, falta de identidad, tenant no autorizado, versión no soportada, evento tardío y evento fuera de orden.
- Prueba tool outage reintetable, error fatal de schema, replay de dead letter y timeout del model.
- Prueba event storms con límites de concurrencia por tenant y por ruta.
- Prueba prompt injection incrustado en el texto del payload.
- Prueba idempotencia reproduciendo el mismo evento después de una escritura parcial.
- Mide tasa de aceptación, tasa de duplicados, tasa de reintentos, tasa de dead letter, latencia de procesamiento, profundidad de la cola, gasto en model/tool y éxito de replay.

Lista de verificación para producción

- Los contratos de eventos tienen versiones y se revisan antes del lanzamiento.
- Los webhooks verifican firmas; las colas verifican origen y permisos.
- El runtime almacena event ID, run ID, idempotency key, route, risk class y policy version.
- Los efectos secundarios son idempotentes, compensables o requieren aprobación.
- Los presupuestos de reintentos y backoff están documentados por clase de error.
- El replay de dead-letter requiere aprobación del responsable y conciencia de versión.
- La profundidad de la cola, tasa de reintentos, conteo de dead-letter, latencia y gasto tienen alertas.
- Los operadores pueden pausar una route, drenar una queue, hacer replay de un evento y deshabilitar write tools.

Usa la lista de verificación de agent event-triggered del libro en línea antes de un piloto en producción.

Recorrido por el código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo circundante explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

Código fuente

Este pattern actualmente no tiene un extracto de código dedicado. Usa el source y los enlaces de descarga a continuación para acceder a la carpeta completa del pattern.

Descarga

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta actual `event-triggered-agent-pattern/` de este repositorio.

Patrones relacionados

- [Durable Workflows](#)
- [Observability and Evals](#)
- [Policy Enforcement](#)
- [Choosing the Right Pattern](#)
- [Resource-Aware Agent Design](#)

Mastra Runtime

Mastra es un pattern de runtime en TypeScript para aplicaciones que necesitan agents, workflows, tools, memory, evals y observability en un solo framework.

Fuente y descargas

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

Propósito

El Mastra Runtime Pattern utiliza Mastra como runtime en TypeScript para aplicaciones de agent en producción. Mastra proporciona a los agents, workflows, tools, memory, evals y observability una estructura de aplicación compartida.

Úsalo cuando

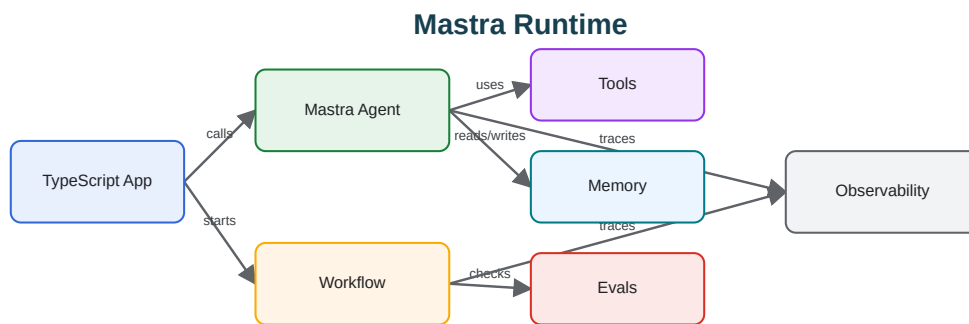
- Estás construyendo un producto de agent basado en TypeScript o Node.
- Necesitas agents y workflows deterministas en el mismo runtime.
- Quieres que memory, tools, evals y tracing sean preocupaciones de primera clase.

Evítalo cuando

- Solo necesitas un script pequeño o una sola llamada a un model.
- Tu equipo está comprometido con un stack de agent en Python.
- No puedes aceptar convenciones de framework sobre estructura de proyecto y despliegue.

Arquitectura

Usa este diagrama para leer Mastra Runtime como un límite de sistema, no solo una forma de código. La pregunta clave de propiedad es: el runtime es dueño del state durable, retries, traces, triggers, configuración de despliegue y controles operativos.



Forma del sistema

- **Límite de aplicación:** el servicio del producto es dueño de la identidad de usuario, alcance de tenant, validación de solicitudes y entrega de respuestas.
- **Límite de runtime:** Mastra aloja los agent, workflow, tools, memory, evals y preocupaciones de observability.
- **Límite de workflow:** las transiciones de state deterministas, retries, esperas de aprobación y puntos de rollback pertenecen a los workflows, no a los prompts.
- **Límite de tool:** los tools exponen entradas tipadas, salidas tipadas, etiquetas de side-effect, requisitos de permisos y campos de trace.
- **Límite de policy:** la policy del producto se ejecuta antes de tools, escrituras de memory, mensajes salientes o side effects externos.
- **Límite de portabilidad:** los prompts, manifiestos de tools, fixtures de eval, schema de trace y reglas de policy siguen siendo legibles fuera del código específico de Mastra.

Protocolo central

1. Acepta una solicitud con actor, tenant, goal, versión de release y clave de idempotencia.
2. Carga la configuración de runtime, policy de memory, registro de tools y state de workflow.
3. Enruta los pasos deterministas a través del workflow y las decisiones abiertas a través del agent.
4. Verifica la policy antes de la recuperación, escrituras de memory, llamadas a tools y respuestas finales que requieran evidencia aprobada.
5. Ejecuta tools mediante wrappers tipados que registran estado, latencia, costo, número de retries e IDs de side-effect.
6. Emite eventos de trace para pasos de workflow, llamadas a model, llamadas a tool, decisiones de policy, acceso a memory y resultados de eval.
7. Ejecuta evals posteriores o de CI sobre el trace antes de promover el cambio.
8. Haz rollback deshabilitando el tool riesgoso, prompt, model, workflow o toda la ruta del agent.

Notas de implementación

- Usa agents para decisiones abiertas donde el siguiente paso no se conoce de antemano.

- Usa workflows para control de flujo predefinido, transiciones de state, retries y orquestación de producción.
- Mantén los tools tipados y testeables de forma independiente.
- Captura traces y evals desde el inicio en lugar de agregarlos después de fallas.
- Mantén las credenciales de proveedores en variables de entorno y documéntalas en `.env.example`.
- Mantén los valores por defecto generados por el framework fuera de la product policy. La product policy debe ser visible en el código, pruebas, ADRs y fixtures de eval.
- Versiona prompts, tools, policies, contratos de memory, datasets de eval y definiciones de workflow juntos.
- Trata las actualizaciones de framework como cambios de runtime: ejecuta regression evals e inspecciona traces antes de promover.

Modos de falla

- Tratar el framework como la arquitectura en vez de modelar goals, state y modos de falla.
- Poner lógica determinista de workflow dentro de prompts.
- Crear tools con descripciones vagas y entradas no validadas.
- Publicar sin datasets de eval o revisión de traces.
- Permitir que escrituras de memory omitan reglas de retención, eliminación, corrección o consentimiento.
- Exportar traces sin redacción o sin suficientes campos para reproducir una falla.
- Esconder rollback dentro de despliegues de código en vez de usar feature flags, deshabilitación de tools o endurecimiento de policy.

Estrategia de evaluación

- Prueba por separado la ruta de agent, ruta de workflow, ruta de denegación de policy, ruta de aprobación y ruta de falla de tool.
- Asegúrate de que el trace contenga eventos de workflow, model, tool, policy, memory y evaluator para ejecuciones representativas.
- Compara cambios en prompt, model, tool y framework contra el mismo set de fixtures antes de liberar.
- Incluye un caso negativo donde el runtime deba redactar o escalar en vez de ejecutar un side effect.

Lista de verificación para producción

- Documenta los comandos de instalación, ejecución local, prueba, eval y limpieza.
- Haz commit de `.env.example` y mantén los valores secretos fuera del código fuente.
- Define el state del workflow, retención de memory, side effects de tools y puntos de enforcement de policy.
- Exporta traces redactados al sistema de observability del equipo.
- Agrega gates de eval en CI para cambios en prompt, model, tool, policy, memory y workflow.
- Define rollback para model, prompt, tool, workflow, policy y deshabilitación total del agent.

Ejecuta el ejemplo

```
npm run mastra-runtime:demo
npm run mastra-runtime:test
```

Recorrido del código

Lee el extracto como la expresión ejecutable más pequeña del pattern. El capítulo alrededor explica las restricciones de diseño; el código muestra dónde esas restricciones se convierten en interfaces concretas, state, validación o control de flujo.

Código fuente

Estos extractos muestran la forma de la implementación. El código completo está disponible en el bundle de descarga y en el 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}`);
    }
  }
];

```

Extracto truncado por legibilidad. Descarga el bundle o abre el archivo fuente para la implementación completa.

[mastra-runtime-pattern/typescript/test/runtime_packaging.spec.ts](#)

[Abrir el código fuente completo](#)

```

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");
```

Descargar

- [Descargar paquete fuente](#)
- [Abrir carpeta fuente](#)

El paquete de descarga contiene la carpeta `mastra-runtime-pattern/` actual de este repositorio.

Patrones Relacionados

- [Durable Workflow](#)
- [Observability and Evals](#)
- [Agent Loop](#)

Hands-On Labs

Los labs convierten los capítulos de referencia en una ruta de construcción. Cada lab utiliza código que ya existe en este repositorio, así que puedes leer el pattern, ejecutar el ejemplo, cambiar una cosa y conectar el resultado con el diseño de producción.

Los labs son intencionalmente agnósticos al framework. Se mueven entre TypeScript y Python, y abarcan runtimes personalizados mínimos, recuperación al estilo LangChain/LangGraph, ejemplos de manager/worker al estilo AutoGen, código de protocolo A2A, límites de tool al estilo MCP y pruebas neutrales al framework. El objetivo no es enseñar una API específica. El objetivo es mostrar la arquitectura que sobrevive cuando el framework cambia.

Usa [Lab Framework and Language Matrix](#) antes de comenzar si quieres ver qué lenguaje, framework y límite arquitectónico enfatiza cada lab. Usa [Lab Production Readiness Checklist](#) y la [lab production readiness worksheet](#) después de cada lab para identificar qué le falta al demo antes de producción. Usa [From-Scratch Mini-Framework Track](#) cuando quieras entender qué empaquetan los agent frameworks bajo el capó. Usa [Vertical Slice Examples](#) después de los labs, o cuando quieras ver varios patterns compuestos en una task realista. Usa [Capstone Projects](#) cuando quieras ejemplos con forma de producto que incluyan ADRs, traces, evals, runbooks, planes de rollback y slices nativos de framework.

Ejecuta estos comandos desde la raíz del repositorio antes de comenzar:

```
npm install
npm test
npm run typecheck
```

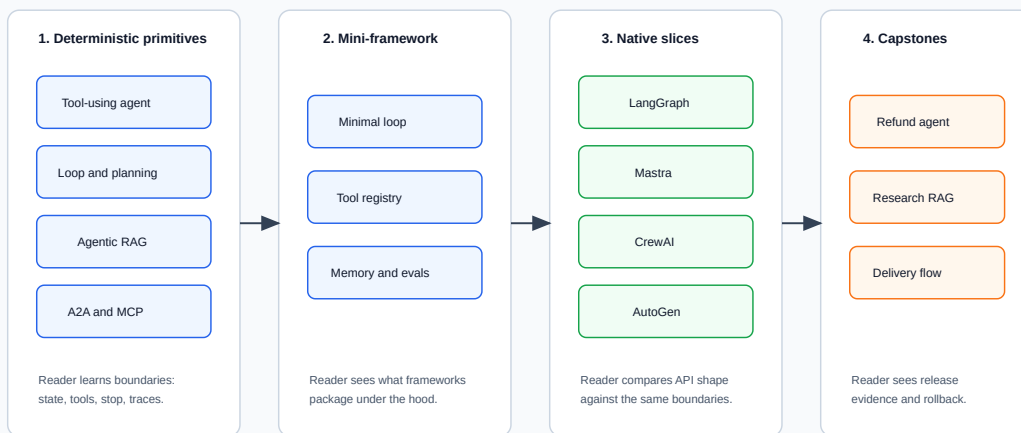
Algunos ejemplos pueden ejecutarse con fallbacks deterministas. Los ejemplos que llaman modelos en vivo requieren un archivo `.env` con `MISTRAL_API_KEY`.

Lab Progression Map

Usa este mapa para entender por qué los labs están ordenados de esta manera. La secuencia inicia con primitivas deterministas, expone el mini-framework debajo de los agent runtimes, compara slices nativos de framework y luego avanza hacia evidencia de lanzamiento a nivel capstone.

Lab Progression Map

The labs move from controlled primitives to framework comparison and product-shaped systems.



Rule: run a deterministic lab first, then compare a native slice, then map the same behavior to a capstone.

Lab Standard

Cada lab debe dejarte con tres cosas: un ejemplo ejecutable, un límite de diseño específico que puedas explicar y un paso de robustecimiento hacia producción que sepas cómo realizar.

Cada lab sigue el mismo contrato de aprendizaje:

1. Indicar el objetivo.
2. Nombrar el lenguaje, framework y archivos fuente.
3. Ejecutar un comando base.
4. Inspeccionar el límite del código.
5. Cambiar una cosa.
6. Verificar el resultado.
7. Identificar qué necesitaría producción a continuación.

Los ejemplos se mantienen pequeños a propósito. Un ejemplo pequeño solo es útil cuando el lab también indica qué falta intencionalmente: state durable, enforcement de policy, schemas más robustos, aprobación, tracing, evals, despliegue o integración con framework. Cuando existe un slice nativo de framework, trátalo como el siguiente punto de comparación, no como un reemplazo del lab determinista.

Planning Table

Usa esta tabla para elegir un lab según el esfuerzo y el resultado. Las estimaciones de tiempo asumen que ya puedes ejecutar las pruebas del repositorio. Cada página de lab también incluye bloques de tiempo opcionales por ejercicio para que puedas dividir el trabajo en sesiones más cortas.

Descarga la worksheet reutilizable: [lab completion worksheet](#).

Descarga la worksheet de seguimiento para producción: [lab production readiness worksheet](#).

Para los labs de mayor impacto, usa las worksheets enfocadas para [Lab 02 planning loops](#), [Lab 03 Agentic RAG](#), [Lab 06 observability and evals](#), [Lab 07 runtime packaging](#) y [Lab 12 state graphs](#).

Compara tu worksheet terminada con los [completed lab evidence examples](#) antes de considerar el lab listo para revisión.

Usa los [captured lab and capstone command output examples](#) cuando necesites un modelo concreto para salida de comandos guardada, snapshots de trace, snapshots de eval y evidencia de lanzamiento.

Lab	Time	Level	Prerequisite	Reusable Artifact
Lab 01 - Tool-Using Agent	20-30 min	Beginner	TypeScript basics	Tool boundary and error behavior.
Lab 02 - Agent Loop and Planning	35-55 min	Beginner	Lab 01 or equivalent tool boundary	Plan/execute split with structured stop-condition evidence.
Lab 03 - Agentic RAG	45-75 min	Intermediate	Retrieval and Python basics	Evidence-grounded answer path plus missing-evidence eval fixture.
Lab 04 - A2A Communication	45-60 min	Intermediate	JSON schema and HTTP/message concepts	Typed agent message envelope.
Lab 05 - Multi-Agent Supervisor	45-60 min	Intermediate	Delegation and aggregation concepts	Supervisor/worker contract.
Lab 06 - Observability and Evals	50-85 min	Intermediate	Any earlier lab	Trace contract, negative eval, and CI gate sketch.
Lab 07 - Mastra Runtime Packaging	70-105 min	Advanced	TypeScript runtime packaging	Agent, tool, workflow, memory, eval, and rollback slice.
Lab 08 - CrewAI Flows and Crews	60-90 min	Advanced	Python and role/task orchestration	Flow, crew, role, and acceptance contract.
Mini-Framework Track	2-4 hr	Advanced	Labs 01, 02, and 06	Runtime primitives you can compare to frameworks.
Lab 09 - Minimal Agent Loop	45-75 min	Intermediate	Mini-framework setup	Loop state, observations, budgets, and stop reasons.
Lab 10 - Tool Registry and Policy Gate	45-75 min	Intermediate	Lab 09	Tool registry with policy decisions.
Lab 11 - Context, Memory, Trace, and Evals	60-90 min	Advanced	Labs 09 and 10	Reviewable runtime trace and trajectory eval.

Lab	Time	Level	Prerequisite	Reusable Artifact
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

Un lab está completo cuando puedes mostrar cuatro cosas:

1. El comando base se ejecuta.
2. La salida esperada coincide con la señal de éxito del lab.
3. Un camino de falla intencional es visible y controlado.
4. Puedes nombrar la brecha de producción antes de usar el pattern con usuarios reales, datos, credenciales o efectos secundarios.

No cuentes un lab como terminado solo porque el camino feliz funciona. El valor viene de ver el límite: qué puede decidir el model, qué debe hacer cumplir el software, qué se traza y qué bloquearía la producción.

Paquete de Evidencia del Lab

Guarda un pequeño paquete de evidencia después de cada lab. Esto convierte el lab de un ejercicio de una sola vez en material que puedes reutilizar en una revisión de diseño, ADR, eval suite o capstone.

Evidencia	Qué Capturar	Por Qué Importa
Baseline command	Comando, estado de salida y señal de salida esperada.	Demuestra que el ejemplo funcionaba antes de que lo cambiaras.
Source boundary	Archivos inspeccionados y el contrato que posee cada archivo.	Muestra dónde el pattern se convierte en código.
Small change	Un cambio en input, rule, prompt, tool, schema o policy.	Demuestra que puedes modificar el comportamiento intencionalmente.
Failure path	Error, rechazo, denegación, timeout, budget stop o input inválido.	Muestra que el boundary falla de forma visible y no silenciosa.
Trace o log	Trace mínimo, transcripción o structured output.	Da a futuros evals algo concreto para validar.
Production gap	Control faltante y el siguiente artifact necesario.	Conecta el lab con la arquitectura de producción.

Mantén el paquete corto. Una pantalla de evidencia es mejor que una carpeta llena de capturas de pantalla sin revisar.

Usa los [ejemplos de evidencia de labs completados](#) como calibración. Un buen paquete de evidencia nombra el comando, salida, failure path, boundary protegido, production gap y el siguiente owner.

Usa los [ejemplos de salida de comando capturada](#) para comparar la forma de tu salida de terminal guardada. La señal importante no es una captura de pantalla. Es un registro corto y revisable que muestra el comando, señal de éxito, trace o enlace a eval, y la pregunta de producción.

Regla Agnóstica de Framework

Los frameworks cambian la API, no las preguntas de arquitectura. Para cada lab, pregunta:

- ¿Quién posee el state?
- ¿Qué puede decidir el model?
- ¿Qué puede validar el software?
- ¿Qué tools están expuestas?
- ¿Qué policy se aplica fuera del prompt?
- ¿Qué se traza?
- ¿Por qué se detiene la ejecución?

Estas preguntas aplican tanto si el código usa LangGraph, LangChain, Mastra AI, agentes estilo AutoGen, CrewAI, MCP, A2A o un runtime personalizado pequeño.

Ruta de Lectura End-To-End

Usa esta ruta cuando quieras pasar de aprender a implementar:

1. Comienza con [Lab Framework and Language Matrix](#) y elige el boundary de mayor riesgo.
2. Ejecuta el lab determinista correspondiente.
3. Lee la extensión de producción y la checklist de preparación.
4. Compara el ejemplo nativo correspondiente bajo [native-framework-examples/](#).
5. Mapea el mismo comportamiento a un capstone.
6. Llena el ADR de selección de framework y la hoja de rollback.
7. Agrega evals que fallen el build antes de agregar tools con efectos secundarios reales.

Para la ruta de reembolso de soporte, usa el Lab 07, [native-framework-examples/mastra-refund/](#), el capstone Support Refund Agent y la hoja de preparación para producción.

Secuencia de Labs

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](#)

Cómo Usar Estos Labs

Lee primero el objetivo, luego ejecuta el comando exactamente como se muestra. Después, inspecciona los archivos fuente nombrados y haz el small change en el lab. El objetivo es ver dónde el pattern se convierte en código: contratos de input, state, límites de tool, condiciones de stop, evaluación y manejo de fallas.

Cada lab termina con una extensión de producción. Trata esa sección como el puente entre un demo funcional y una decisión de arquitectura.

Mapa de Salida Esperada

Usa esta tabla como verificación rápida de éxito antes de pasar a la extensión de producción.

Lab	Expected Output Signal
Lab 01	Resultados estructurados <code>read_order</code> y <code>search_refund_policy</code> con etiquetas de confianza y referencias de evidencia.
Lab 02	<code>Planning test OK</code> más un plan determinista y resultado para el camino CLI.
Lab 03	La respuesta refleja la evidencia recuperada del set de documentos local.
Lab 04	La prueba A2A muestra éxito, rechazo, error de input inválido y cancelación.
Lab 05	El manager delega trabajo acotado y la agregación final usa los outputs de los workers.
Lab 06	Los registros de eval exponen casos de éxito y negativos, no solo el texto final.
Lab 07	<code>Mastra-style runtime packaging tests OK</code> ; el slice nativo expone agent, tools, workflow y eval gate.
Lab 08	<code>CrewAI-style flow and crew tests OK</code> ; Flow acepta solo outputs de roles validados.
Lab 09	La respuesta inmediata se detiene con <code>success</code> ; las propuestas repetidas de tool se detienen con <code>budget_exhausted</code> .

Lab	Expected Output Signal
Lab 10	Tools desconocidas son rechazadas, tools de escritura requieren aprobación y las tools permitidas registran observaciones.
Lab 11	El trace contiene context, decisión, tool/policy y eventos de stop; eval de trayectoria insegura falla.
Lab 12	<code>LangGraph-style state graph tests OK</code> ; resume preserva el state con checkpoint.
Lab 13	<code>AutoGen-style transcript tests OK</code> ; la transcripción prueba el orden de roles, razón de stop y owner final.

Orden Recomendado

Haz los labs en orden si eres nuevo en agent systems. La secuencia comienza con un agent y un tool, luego agrega planeación, retrieval, comunicación remota entre agents, coordinación multi-agent y evaluación de calidad de producción.

Si ya conoces lo básico, comienza con el lab más cercano a tu sistema actual y usa los capítulos relacionados como material de referencia.

Después de los labs, lee los vertical slices para ver cómo los mismos patterns se componen en workflows de soporte, código e investigación. Luego lee los [Capstone Projects](#) para ver sistemas con forma de producción, mapeos de framework, slices nativos y evidencia de release.

Si estás evaluando frameworks, haz el track de mini-framework antes de elegir un runtime de producción. Construir los primitivos una vez facilita ver qué responsabilidades posee un framework y cuáles permanecen en tu aplicación.

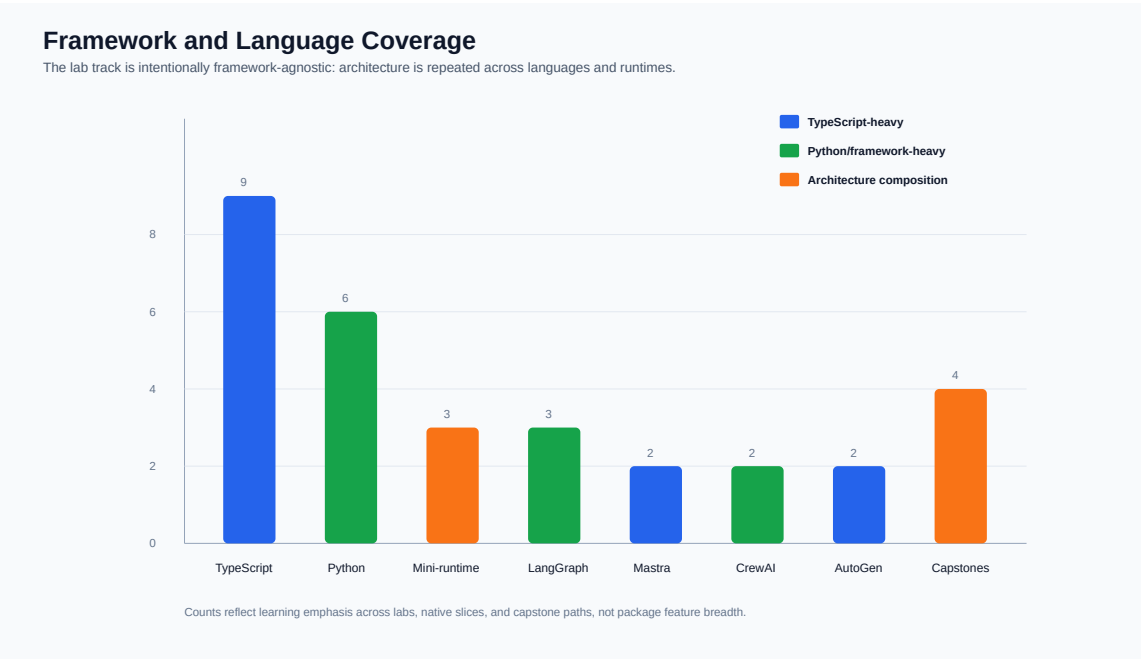
Lab Framework and Language Matrix

Los labs son intencionalmente agnósticos al lenguaje y framework. Usan diferentes tools para que puedas ver el pattern arquitectónico detrás del API del framework.

Usa esta página antes de empezar a programar. Responde dos preguntas: ¿qué lab debo ejecutar primero y qué boundary de producción debo inspeccionar mientras lo ejecuto?

Coverage Graph

Usa este gráfico para ver el enfoque de aprendizaje entre lenguajes, runtimes, slices de framework y capstones. El goal es exponer una arquitectura balanceada, no cubrir paquetes de forma igualitaria.



Lab	Pattern	Language	Framework / Runtime	Framework-Agnostic Lesson
Lab 01 - Tool-Using Agent	Tool use	TypeScript	Minimal custom runtime / AutoGen-style example	El model propone un capability; el software se encarga de la validación y ejecución.
Lab 02 - Agent Loop and Planning	Planning and execution	TypeScript, con espejo en Python	Framework-neutral planner/executor	Planning y execution son responsabilidades separadas, incluso cuando un framework empaqueta ambas.
Lab 03 - Agentic RAG	Retrieval and grounding	Python	LangChain/LangGraph-style retrieval stack	Retrieval produce evidencia acotada; la generación debe

Lab	Pattern	Language	Framework / Runtime	Framework-Agnostic Lesson
				mantenerse fundamentada en esa evidencia.
Lab 04 - A2A Communication	Agent-to-agent protocol	TypeScript	Protocol-first runtime con validación de schema Ajv	La comunicación entre agents necesita sobres tipados, correlation IDs, rechazos, errores y cancelación.
Lab 05 - Multi-Agent Supervisor	Supervisor / worker	TypeScript	AutoGen-style manager/worker example	Un supervisor se encarga de la descomposición, contratos de worker y síntesis final.
Lab 06 - Observability and Evals	Trace and eval harness	TypeScript	Framework-neutral tests over examples	Los evals deben inspeccionar trayectorias, no solo respuestas finales.
Lab 07 - Mastra Runtime Packaging	Runtime packaging	TypeScript	Mastra-style agents, tools, workflows, memory y evals	El runtime packaging del framework no elimina la propiedad del producto sobre state, policy y aceptación.
Lab 08 - CrewAI Flows and Crews	Flow and crew orchestration	Python	CrewAI-style flows, crews, roles y tasks	Los flows son dueños del state y aceptación; los crews realizan trabajo especializado acotado.
Mini-Framework Track	Runtime primitives	TypeScript o Python	Runtime educativo desde cero	Construir los primitivos una vez aclara qué empaquetan los frameworks.
Lab 09 - Minimal Agent Loop	Agent loop	TypeScript o Python	Runtime educativo desde cero	State, decisiones, observaciones, presupuestos y razones de detención son el núcleo del loop.
Lab 10 - Tool Registry and Policy Gate	Tool and policy boundary	TypeScript o Python	Runtime educativo desde cero	La disponibilidad de tools y la autorización de policy son decisiones distintas del runtime.

Lab	Pattern	Language	Framework / Runtime	Framework-Agnostic Lesson
Lab 11 - Context, Memory, Trace, and Evals	Runtime observability	TypeScript o Python	Runtime educativo desde cero	Context, memory, traces y trajectory evals hacen el runtime revisable.
Lab 12 - LangGraph State Graph	State graph and resume	Python	LangGraph-style graph state, nodes, edges, checkpoints e interrupts	La ejecución en graph es más fuerte cuando importan state, ramificación, pausa/reanudación y observabilidad de nodos.
Lab 13 - AutoGen Transcript Evals	Multi-agent transcript evaluation	TypeScript	AutoGen-style agents, teams, messages y transcript evals	Una ejecución multi-agent necesita un transcript revisable, razón explícita de detención y criterios de aceptación a nivel de rol.

How To Read The Matrix

No trates la columna de framework como el objetivo del lab. Trátala como la superficie de implementación. La lección duradera es el boundary: state, tools, policy, context, comunicación, evaluación o control del runtime.

Si después usas LangGraph, Mastra AI, AutoGen, CrewAI, Semantic Kernel, MCP o un runtime personalizado, mantén las mismas preguntas en mente:

- ¿Qué es propiedad del framework?
- ¿Qué sigue siendo propiedad de tu aplicación?
- ¿Dónde se persiste el state?
- ¿Dónde se validan las llamadas a tools?
- ¿Dónde se aplica la policy?
- ¿Qué se puede replay después de una falla?

Choose By Situation

Usa esta tabla cuando ya sepas el problema que quieres resolver.

Si tu problema actual es...	Empieza con	Luego lee	Por qué
Exponer un tool a un model	Lab 01	Tool Capability Design	Necesitas inputs tipados, errores controlados y límites de permisos antes de la autonomía.
Agregar planificación o ejecución paso a paso	Lab 02	Planning and Execution	Necesitas separar la creación del plan, la ejecución y las razones de

Si tu problema actual es...	Empieza con	Luego lee	Por qué
			detención.
Fundamentar respuestas en documentos	Lab 03	Semantic Recall and RAG	Necesitas elegibilidad de fuentes, calidad de retrieval, citas y manejo de evidencia faltante.
Conectar agents o servicios	Lab 04	A2A Agent Interoperability	Necesitas sobres tipados, identidad, rechazos, errores, cancelación y replay.
Dividir trabajo entre roles	Lab 05	Supervisor / Worker	Necesitas contratos de worker, policy de fusión y un único dueño final.
Probar comportamiento con traces y evals	Lab 06	Observability and Evals	Necesitas evidencia de trayectoria, no solo respuestas finales.
Comparar runtime packaging de frameworks	Lab 07	Framework Selection	Necesitas ver qué empaqueta un runtime y qué sigue siendo propiedad de tu producto.
Modelar orquestación tipo crew	Lab 08	Choosing Multi-Agent Topology	Necesitas distinguir el state del flow del output de worker basado en roles.
Entender qué empaquetan los frameworks	Mini-Framework Track	Building a Minimal Agent Runtime	Necesitas los primitivos antes de comparar abstracciones de frameworks.
Probar resume, interrupts o graph state	Lab 12	Durable Workflows	Necesitas checkpoints, transiciones de state y recuperación ante fallas.
Evaluar conversaciones multi-agent	Lab 13	Debate and Consensus	Necesitas evidencia en el transcript, aceptación por rol y condiciones de detención.

Si aplican varias filas, empieza por el boundary más riesgoso. Un tool con capacidad de escritura, fuente de datos privada, workflow de larga duración o handoff multi-agent deben guiar la elección del lab.

Atajo de Decisión Rápida

Usa este atajo cuando tienes cinco minutos y necesitas una ruta inicial.

Mayor Riesgo en Tu Sistema	Ejecuta Primero	Agrega Después	Revisión para Producción
Un model puede invocar un tool	Lab 01	Lab 10	Tool schema, permiso, idempotencia, timeout y registro de auditoría.

Mayor Riesgo en Tu Sistema	Ejecuta Primero	Agrega Después	Revisión para Producción
Un model puede planear varios pasos	Lab 02	Lab 09	Motivo de detención, presupuesto, regla de reintento, cancelación y eventos de trace.
Un model responde desde documentos	Lab 03	Lab 11	ACLs de fuente, frescura, citas, comportamiento ante evidencia faltante y retrieval evals.
Agents intercambian mensajes	Lab 04	Lab 13	Envelope tipado, identidad, correlation ID, rechazo, cancelación y transcript replay.
El trabajo se divide entre roles	Lab 05	Lab 08	Contrato de worker, merge policy, dueño final, permisos de rol y acceptance eval.
La calidad del release es incierta	Lab 06	Lab 11	Contrato de trace, casos negativos, trajectory evals y umbrales de bloqueo.
El trabajo puede pausarse y reanudarse	Lab 12	Capítulo Durable Workflows	Almacenamiento de checkpoint, payload de interrupción, migración de state y dashboard de ejecuciones atascadas.
La elección de framework es incierta	Mini-Framework Track	Lab 07	División de ownership entre framework runtime y product policy.

No optimices por el lab más interesante. Optimiza por el límite más probable de dañar usuarios, filtrar datos, gastar dinero o bloquear operadores.

Rutas de Lab a Producto

Usa estas rutas cuando quieras que un lab se convierta en material para revisión de diseño.

Corte de Producto	Ejecuta	Compara Con	Finaliza Con
Support refund agent	Lab 01, Lab 07, Lab 12	native-framework-examples/mastra-refund/ , native-framework-examples/langgraph-refund/	Support Refund Agent Capstone
Research RAG assistant	Lab 03, Lab 06, Lab 11	native-framework-examples/langgraph-research-rag/	Research RAG Agent Capstone
Multi-agent delivery workflow	Lab 05, Lab 08, Lab 13	native-framework-examples/autogen-delivery/ , native-framework-examples/crewai-delivery/	Multi-Agent Delivery Workflow Capstone
Custom runtime foundation	Labs 09, 10, 11	Implementación mini-framework TypeScript o Python	Reference Architecture

Cada ruta debe terminar con una hoja de trabajo, no solo un comando exitoso. Usa la [hoja de trabajo de finalización de lab](#), luego la [hoja de trabajo de preparación para producción de lab](#).

Cobertura Actual

Los labs actuales ya cubren TypeScript, Python, límites de protocolo, pruebas neutrales a framework, grafos de state estilo LangGraph, transcripts estilo AutoGen, empaquetado de runtime estilo Mastra, orquestación de flow estilo CrewAI, primitivas de runtime desde cero, puertas de preparación para producción y ejemplos nativos de framework aislados.

Ejemplos nativos del repositorio:

Ejemplo	Framework	Conecta Con
<code>native-framework-examples/langgraph-refund/</code>	LangGraph	Lab 12 y Support Refund Agent capstone
<code>native-framework-examples/langgraph-research-rag/</code>	LangGraph	Lab 03 y Research RAG Agent capstone
<code>native-framework-examples/mastra-refund/</code>	Mastra	Lab 07 y Support Refund Agent capstone
<code>native-framework-examples/autogen-delivery/</code>	AutoGen	Lab 13 y Multi-Agent Delivery Workflow capstone
<code>native-framework-examples/crewai-delivery/</code>	CrewAI	Lab 08 y Multi-Agent Delivery Workflow capstone

La expansión planeada de labs debe agregar:

- recorridos de despliegue más profundos que conecten la checklist de preparación con objetivos concretos de cloud/runtime.

Puerta de Revisión de la Matriz

Antes de agregar un nuevo lab, actualiza esta página solo si el lab cambia la elección del lector. Un nuevo lab debe agregar al menos uno de estos:

- un nuevo límite de arquitectura;
- una comparación de lenguaje o runtime que falte;
- una preocupación de producción no cubierta por los labs existentes;
- una ruta capstone que los lectores puedan seguir de principio a fin;
- una porción nativa de framework que aclare qué posee el framework.

No agregues labs solo para cubrir otra biblioteca. Agrégalos cuando enseñen un límite que los ingenieros necesitan para lanzar sistemas más seguros.

Lista de verificación de preparación para producción del laboratorio

Los laboratorios son implementaciones didácticas. Esta lista de verificación define lo que debe agregarse antes de que un pattern de laboratorio se convierta en trabajo de producción.

Úsala después de completar un laboratorio. El objetivo es identificar el siguiente límite de ingeniería: persistencia, autorización, reintentos, idempotencia, observabilidad, eval gates, despliegue y rollback. Para el camino completo hacia producción, continúa con [Deployment Walkthrough](#), [Templates and Worksheets](#) y el [10/10 Production Gate](#).

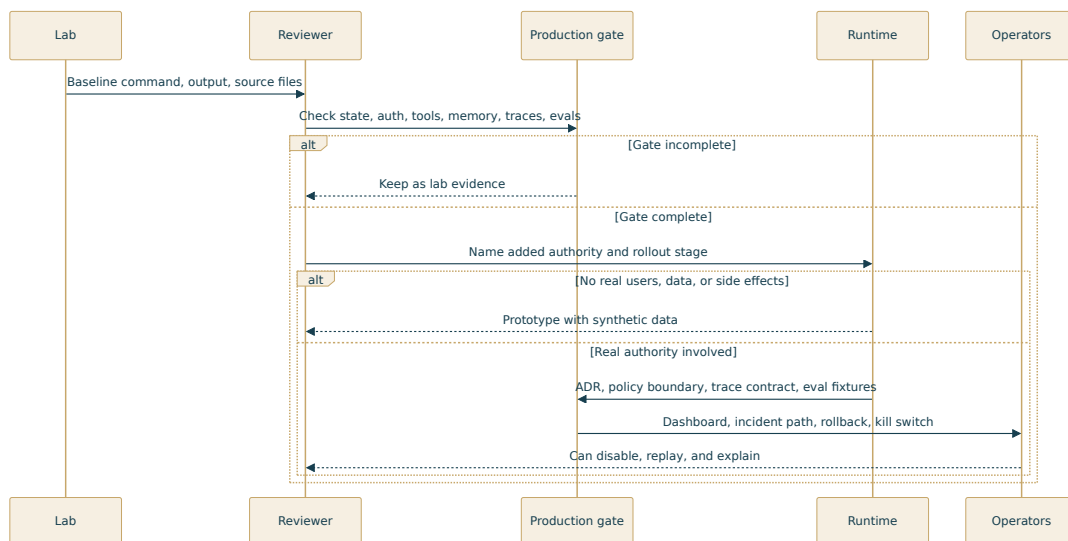
Descarga el artifact reutilizable para revisión de producción: [10/10 production gate scorecard](#).

Descarga la hoja de trabajo específica del laboratorio: [lab production readiness worksheet](#).

Universal Production Gate

Cada laboratorio necesita estos controles antes de involucrar usuarios reales, datos reales o efectos secundarios reales:

Gate	Evidencia requerida
State ownership	State schema, owner, persistence strategy, migration plan y replay story.
Auth and permissions	Actor identity, tenant/resource scope, tool permissions, approval rules y audit records.
Tool safety	Input schemas, side-effect class, idempotency key, timeout, retry policy y error contract.
Memory safety	Retention, deletion, redaction, consent, correction path y write policy.
Observability	Trace IDs, model/tool events, policy decisions, costs, latency, stop reasons y redaction.
Eval gates	Golden tasks, negative cases, trajectory checks, safety checks y release-blocking thresholds.
Deployment	Runtime owner, environment config, secrets, scaling limits, rollback, kill switch y incident path.
Human control	Approval UI/API, escalation rules, cancellation, reviewer identity y expiry.



Usa este flujo antes de cambiar acceso, datos o autoridad. Un laboratorio solo puede avanzar cuando la siguiente etapa tiene evidencia revisable, no solo porque el comando de ejemplo se ejecutó una vez.

Lab Review Questions

Usa estas preguntas antes de que un sistema derivado de laboratorio se convierta en un product slice:

Pregunta	Evidencia requerida
¿Qué demostró el laboratorio?	Baseline command, expected output, source files y success signal.
¿Qué no demostró el laboratorio?	Falta de persistencia, policy, observability, evals, deployment, rollback o human control.
¿Qué autoridad agregará producción?	Read data, write tools, memory, messages, approvals, money movement o workflow execution.
¿Qué debe fallar cerrado?	Missing config, missing policy context, unsafe tool input, stale evidence y failed evals.
¿Cuál es el siguiente artifact de producción?	ADR, trace contract, eval fixture, runbook, checklist, dashboard o rollback plan.

El laboratorio es un artifact de aprendizaje hasta que estas respuestas existan en documentos de ingeniería revisables.

Promotion Ladder

Usa esta escalera para decidir para qué está listo el output del laboratorio.

Etapas	Uso permitido	Evidencia requerida	Condición de detención
Lab evidence	Aprendizaje, comparación y discusión de diseño.	Baseline command, success signal, source files	No conectar usuarios reales, datos privados, credenciales ni efectos secundarios.

Etapa	Uso permitido	Evidencia requerida	Condición de detención
		inspeccionados y una ruta de falla conocida.	
Prototype	Demo interna con datos sintéticos.	Lab evidence más schemas básicos, prueba determinística y gaps de producción identificados.	Detener si la demo requiere credenciales reales o acceso amplio a tools.
Product slice	Workflow interno limitado con datos controlados.	ADR, policy boundary, trace contract, eval fixtures, owner y nota de rollback.	Detener si falta policy, trace o rollback.
Pilot	Despliegue pequeño con usuarios reales o datos reales.	Production gate scorecard, approval rules, incident path, dashboard y release gate.	Detener o hacer rollback ante evals bloqueantes fallidos, trace spans faltantes o efectos secundarios inseguros.
Production candidate	Lanzamiento controlado en producción.	Evidencia de deployment walkthrough, runbook, kill switch, rollback path y eval report actual.	Detener si los operadores no pueden deshabilitar, replay o explicar la ejecución.

No saltes etapas solo porque un ejemplo de framework se ejecuta exitosamente. Un comando exitoso prueba la ejecución; no prueba autoridad, observabilidad, recuperación ni seguridad de lanzamiento.

Per-Lab Readiness Matrix

Laboratorio	Adiciones para producción
Lab 01 - Tool-Using Agent	Tool schemas, side-effect labels, permission checks, idempotency, timeout/retry y audit records.
Lab 02 - Agent Loop and Planning	Durable state, plan versioning, step retry policy, cancellation, partial failure handling y plan evals.
Lab 03 - Agentic RAG	Source ACLs, freshness, citation checks, retrieval evals, prompt-injection filtering y evidence retention rules.
Lab 04 - A2A Communication	Agent identity, signed envelopes, correlation IDs, cancellation semantics, schema versioning y replay logs.
Lab 05 - Multi-Agent Supervisor	Worker contracts, merge policy, per-worker traces, disagreement handling, final acceptance owner y cost caps.
Lab 06 - Observability and Evals	Trace storage, redaction, eval datasets, release thresholds, incident-to-eval workflow y dashboard ownership.
Lab 07 - Mastra Runtime Packaging	Deployment config, tool/memory policy, workflow retries, eval integration, trace export y framework upgrade plan.

Laboratorio	Adiciones para producción
Lab 08 - CrewAI Flows and Crews	Flow checkpoints, role permissions, task schemas, crew output validators, human escalation y flow acceptance evals.
Lab 09 - Minimal Agent Loop	Decision validation, tool registry, stop policy, state persistence, timeout/cancellation y trace events.
Lab 10 - Tool Registry and Policy Gate	Policy context, approval records, schema validation, side-effect isolation, idempotency y denial analytics.
Lab 11 - Context, Memory, Trace, and Evals	Memory governance, trace redaction, eval fixture versioning, context audit y incident replay.
Lab 12 - LangGraph State Graph	Durable checkpointer, thread IDs, interrupt payloads, node idempotency, state migrations y resume tests.
Lab 13 - AutoGen Transcript Evals	Message schemas, team termination, transcript redaction, role permissions, transcript replay y team-level eval gates.

Framework-Specific Deployment Questions

Framework Shape	Preguntas para producción
LangGraph-style	¿Dónde se almacena el checkpointer? ¿Cómo se asignan los thread IDs? ¿Qué nodos pueden causar side effects? ¿Qué state migrations se soportan?
AutoGen-style	¿Quién es dueño del transcript? ¿Qué mensajes son durables? ¿Cómo funciona la terminación? ¿Cómo se autorizan los agent tools?
Mastra-style	¿Qué features del runtime son propiedad del framework? ¿Cómo se despliegan los workflows? ¿Cómo se exportan tools, memory, traces y evals?
CrewAI-style	¿Qué posee el flow? ¿Qué posee el crew? ¿Cómo se validan los outputs de roles? ¿Cómo se escalan los fallos del crew?
Mini-runtime	¿Qué controles de producción estás dispuesto a construir y operar tú mismo? ¿Cuáles deberían migrar a una plataforma de workflow existente?

Release Gate Template

Antes de lanzar un sistema derivado de laboratorio, requiere:

```
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
```

Si alguna línea es desconocida, el sistema sigue siendo una demo.

Capítulos Relacionados

- [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 - Construye un Agent que Usa Tools

Descarga la [hoja de trabajo de finalización del laboratorio](#) y la [hoja de trabajo de preparación para producción](#) antes de comenzar.

Objetivo

Construye el límite de tool más pequeño y útil: un runtime recibe una propuesta de uso de tool, valida la ruta y los argumentos, ejecuta el tool detrás del código, marca outputs confiables y no confiables, y retorna un resultado estructurado.

Qué Vas a Usar

- Lenguaje: TypeScript
- Framework/runtime: runtime personalizado mínimo con un ejemplo de ruta estilo AutoGen
- Lección agnóstica de framework: el model puede proponer el uso de tools, pero el software es responsable de la validación, ejecución y manejo de errores.
- Capítulo de pattern: [Tool Use](#)
- Carpeta fuente: `tool-using-agent-pattern/`
- Descarga: [tool-use.zip](#)
- Archivo principal: `tool-using-agent-pattern/typescript/src/tool_runtime.ts`
- Archivo demo: `tool-using-agent-pattern/typescript/src/run_demo.ts`

Presupuesto de Tiempo del Ejercicio

Estas estimaciones asumen que las dependencias ya están instaladas.

Ejercicio	Tiempo	Output
Configuración y ejecución base	5 min	Output del comando tool-runtime exitoso.
Inspecciona el límite del tool	8 min	Notas sobre validación de rutas, datos confiables y texto de policy no confiable.
Cambia una ruta o caso de tool	8-10 min	Un éxito visible, denegación o error controlado.
Revisa la brecha de producción	5-7 min	Un control faltante para llevar a la hoja de preparación.

Configuración

Desde la raíz del repositorio:

```
npm install
```

Este laboratorio se ejecuta sin una key de model. Ejercita el tool runtime directamente para que puedas inspeccionar el límite de software antes de agregar propuestas del model.

Ejecútalo

```
npm run tool-using-agent
```

Output esperado:

```
{
  "order": {
    "status": "ok",
    "tool": "read_order",
    "trust": "trusted_system"
  },
  "policy": {
    "status": "ok",
    "tool": "search_refund_policy",
    "trust": "untrusted_content"
  }
}
```

El output real también incluye datos del tool y valores de `evidenceRef`. Registra esos campos en la hoja de finalización del laboratorio.

Inspecciona el Código

Abre `tool-using-agent-pattern/typescript/src/tool_runtime.ts` y encuentra:

- `ToolName`: los nombres de tools expuestos.
- `ToolContext`: el context de runtime para ruta, aprobaciones, timeouts e intentos.
- `disclosedTools(route)`: el límite de disclosure de tools basado en rutas.
- `validateProposal(proposal, context)`: el límite de validación.
- `execute(proposal, context)`: el límite de ejecución.

Luego abre `tool-using-agent-pattern/typescript/src/run_demo.ts` y encuentra las dos propuestas demo: `read_order` y `search_refund_policy`.

El punto de diseño importante es que un model puede proponer una llamada a un tool, pero el código es responsable de disclosure, validación, ejecución, marcado de confianza y referencias de evidencia.

Cambia Una Cosa

Cambia la ruta en `run_demo.ts` de:

```
route: "refund_investigation",
```

a:

```
route: "order_status",
```

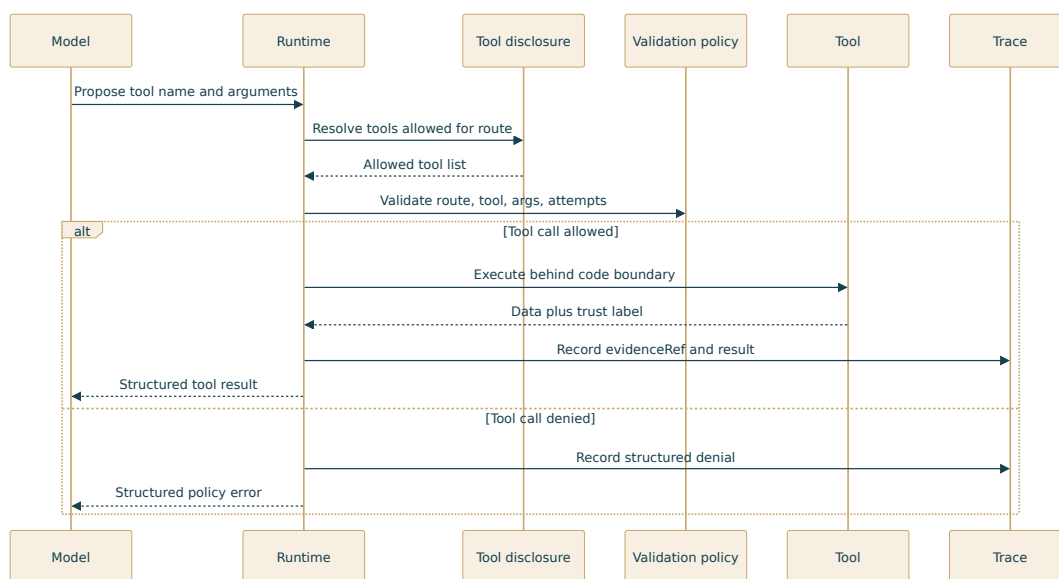
Ejecuta el laboratorio de nuevo.

Resultado Esperado

La llamada `read_order` aún debe tener éxito porque `order_status` puede usar ese tool. La llamada `search_refund_policy` debe ser denegada porque esa ruta solo expone `read_order`.

Registra la denegación como el camino de falla intencional. Esto prueba que la disponibilidad de tools depende de la ruta, no de lo que el model solicita.

Usa este flujo como el modelo de aceptación para el laboratorio. El model propone una llamada, pero el runtime es responsable de disclosure, validación, ejecución, marcado de confianza y evidencia de trace.



Puerta de Revisión del Lab

Antes de continuar, verifica el límite en vez de solo revisar el camino feliz:

Revisión	Evidencia
El disclosure de tools es limitado	<code>order_status</code> expone <code>read_order</code> ; <code>refund_investigation</code> expone tools de refund-investigation.
La ejecución de tools es propiedad del código	<code>ToolRuntime</code> ejecuta <code>read_order</code> y <code>search_refund_policy</code> ; el texto del model no.
El uso inválido de ruta/tool está controlado	Un tool oculto o no permitido retorna una denegación estructurada.

Revisión	Evidencia
La confianza es explícita	Los datos de order son <code>trusted_system</code> ; el texto de policy es <code>untrusted_content</code> .
La evidencia es rastreable	Los resultados exitosos de tools incluyen <code>evidenceRef</code> .
La brecha de producción es visible	El laboratorio nombra la integración de autorización faltante, exportación de traces, dashboards, despliegue y manejo de incidentes.

Registra el comando, output y comportamiento de falla en la hoja de finalización del laboratorio.

Extensión a Producción

Reemplaza el enrutamiento por prefijo con una solicitud de tool tipada:

- nombre del tool
- JSON schema para argumentos
- verificación de autorización
- timeout
- resultado estructurado de éxito o error
- trace ID para la ejecución

No le des al model acceso amplio a funciones arbitrarias. Expón tools limitados con permisos explícitos.

Puente a Producción

Usa esta tabla al adaptar el laboratorio a un tool real de producto:

Concepto del Lab	Versión de Producción
Unión <code>ToolName</code>	Manifest de tools versionado con owner, permiso, timeout y clase de side-effect.
<code>disclosedTools(route)</code>	Disclosure de tools basado en ruta, actor, tenant y riesgo.
<code>validateProposal(...)</code>	Validación de schema, policy, aprobación, presupuesto y state.
Campo <code>trust</code>	Clasificación de datos que separa datos confiables del sistema de contenido no confiable.
<code>evidenceRef</code>	Referencia de evidencia ligada a trace para auditoría, replay y evals.
Output de consola demo	Respuesta estructurada más evento de trace, registro de eval y señal de dashboard.

El primer hito de producción no es agregar más tools. Es probar que un tool puede ser llamado, denegado, rastreado y probado de forma segura.

Mapeo Entre Frameworks

- En LangGraph, este límite usualmente aparece como un nodo de tool o callable ligado al state del graph.
- En Mastra AI, se mapea a un tool tipado expuesto a través de un agent o workflow.
- En sistemas estilo AutoGen, se mapea a una llamada de función/tool propuesta por un assistant y ejecutada por el runtime.
- En CrewAI, se mapea a tools asignados a un rol de agent, con el flujo o configuración de crew limitando el acceso.

Capítulos Relacionados

- [MCP-first Tool Use](#)
- [Human Approval Gates](#)
- [Policy Enforcement](#)

Lab 02 - Construye un Agent Loop con Planning

Descarga la [hoja de ejercicios guiados del planning loop de Lab 02](#), la [hoja de finalización del laboratorio](#) y la [hoja de preparación para producción del laboratorio](#) antes de comenzar.

Objetivo

Separa el planning de la ejecución. El planner decide los pasos; el executor ejecuta operaciones acotadas y registra los resultados. Esto te da la estructura de control detrás de muchos agent loops sin hacer que cada paso sea autónomo.

Qué Vas a Usar

- Lenguaje: TypeScript, con un espejo en Python
- Framework/runtime: planner y executor neutrales al framework
- Lección agnóstica al framework: el planning y la ejecución son responsabilidades separadas incluso cuando un framework las empaqueta juntas.
- Capítulos de patterns: [Agent Loop](#), [Planning and Execution](#)
- Carpeta de código fuente: `planning-pattern/`
- Descarga: [planning-and-execution.zip](#)
- Archivos principales:
 - `planning-pattern/typescript/src/planner.ts`
 - `planning-pattern/typescript/src/executor.ts`
 - `planning-pattern/typescript/src/run.ts`

Tiempo Estimado del Ejercicio

Estas estimaciones asumen que las dependencias ya están instaladas.

Ejercicio	Tiempo	Output
Configuración y prueba base	5-8 min	Output de prueba passing del planner/executor.
Ejecutar el plan base y trazar	10-12 min	Pasos del plan, result map y señal de stop.
Cambiar input e inspeccionar fallos	10-15 min	Evidencia del input cambiado más comportamiento de input no soportado o faltante.
Escribir la regla de stop-condition	10-15 min	Una razón de stop visible para el caller y nota de manejo en producción.
Completar la revisión final	5 min	Notas en la hoja sobre el límite de ejecución y gap de producción.

Configuración

Desde la raíz del repositorio:

```
npm install
```

Ejecútalo

Ejecuta la prueba determinista:

```
npm run plan:test
```

Ejecuta el path de CLI:

```
npm run plan:run -- "Compute average of [1,2,3,4]"
```

Ejecuta el espejo en Python:

```
npm run plan:py
```

Inspecciona el Código

Abre `planning-pattern/typescript/src/planner.ts` e inspecciona cómo un task se convierte en una lista de pasos. Luego abre `planning-pattern/typescript/src/executor.ts` e inspecciona cómo la ejecución convierte esos pasos en resultados nombrados.

Busca estos límites:

- el objeto plan
- step IDs
- funciones executor deterministas
- result map
- failure surface

Cambia Una Cosa

Cambia el texto del task:

```
npm run plan:run -- "Compute average of [10,20,30]"
```

Luego inspecciona si el fallback determinista sigue produciendo la forma esperada del plan.

Resultado Esperado

La prueba debe imprimir:

Planning test OK

El CLI debe imprimir un plan, eventos de progreso y un resultado calculado:

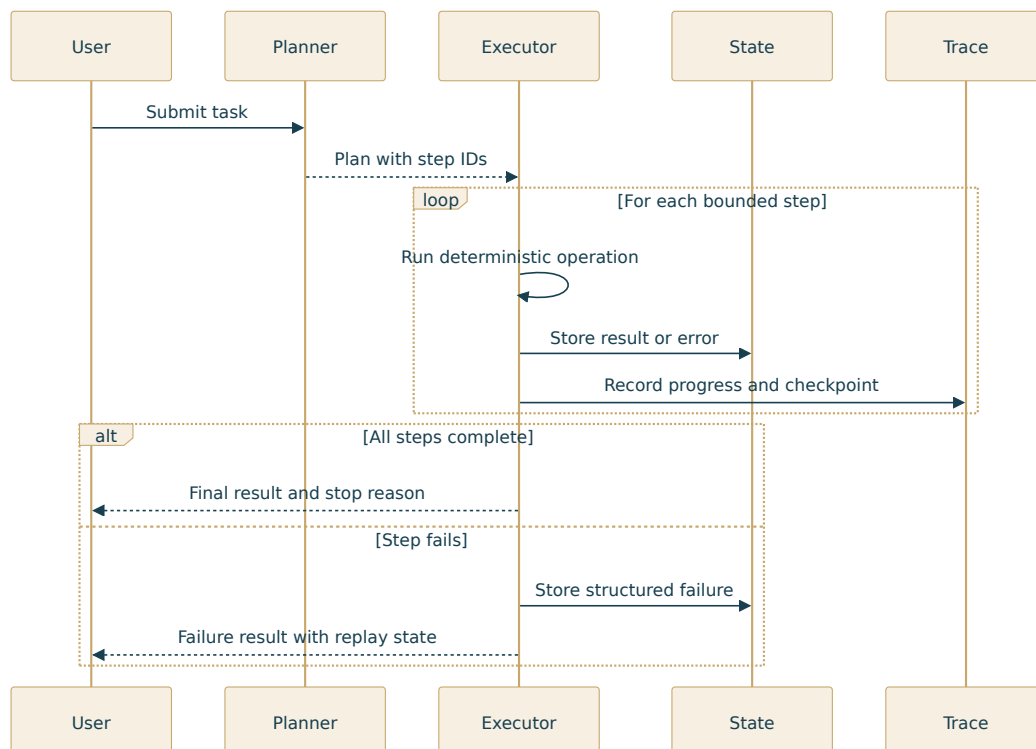
```
Plan: {
  steps: [
    { id: 's1', description: 'Load numbers [1,2,3,4]' },
    { id: 's2', description: 'Compute average' }
  ],
  rationale: 'synthetic'
}
Progress 0 s1
Progress 50 s2
Progress 100 done
Results: { s1: [ 1, 2, 3, 4 ], s2: 2.5 }
```

Después de cambiar el task a `[10,20,30]`, el resultado debe ser:

```
Results: { s1: [ 10, 20, 30 ], s2: 20 }
```

Si extiendes el planner, mantén el executor determinista y testeable.

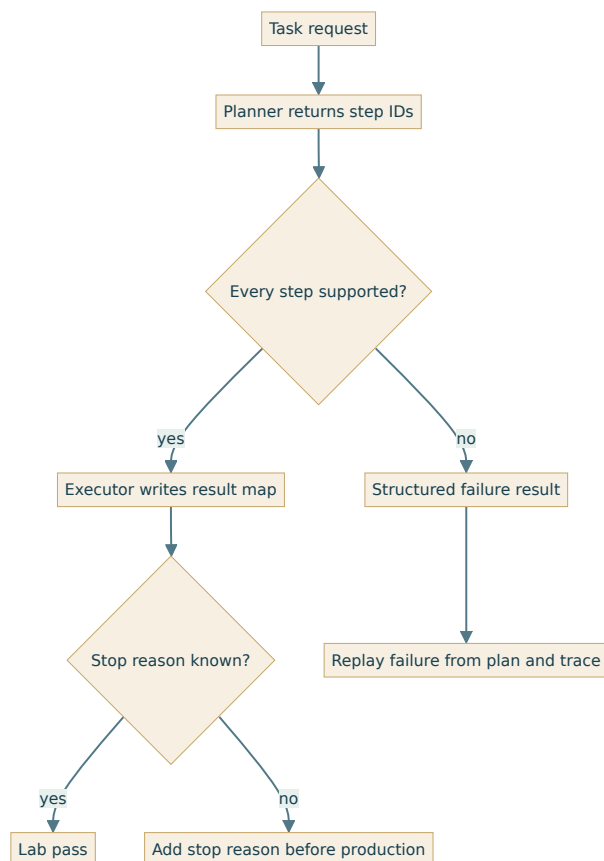
Usa este flujo como modelo de aceptación para el laboratorio. El planner puede elegir los pasos, pero la ejecución sigue siendo acotada, trazable y reproducible.



Ejercicios Guiados

Usa estos ejercicios para convertir el happy path en un loop trace revisable.

Ejercicio	Tiempo	Qué Hacer	Evidencia a Guardar
Plan base y trace	10 min	Ejecuta <code>npm run plan:test</code> y el comando CLI para <code>[1,2,3,4]</code> .	Pasos del plan, eventos de progreso, result map y señal de stop.
Trace con cambio de input	8 min	Ejecuta el comando CLI para <code>[10,20,30]</code> .	El input cambiado <code>s1</code> y el average cambiado <code>s2</code> .
Trace de paso no soportado	8 min	Inspecciona los casos negativos en <code>planning-pattern/typescript/test/planning.spec.ts</code> .	Un failure estructurado <code>unsupported_step</code> en vez de <code>null</code> .
Trace de input faltante	8 min	Inspecciona el caso de prueba que ejecuta <code>Compute average</code> sin cargar números primero.	Un failure estructurado <code>missing_numbers</code> .
Regla de stop en producción	10 min	Escribe la razón de stop que expondrías al caller.	<code>success</code> , <code>unsupported_step</code> , <code>missing_numbers</code> o <code>budget_exhausted</code> .



Ejercicio de Stop-Condition

El ejecutor ahora retorna un objeto de failure para trabajo no soportado o mal formado. Usa esto como modelo para condiciones de stop en producción:

```
await executePlan([
  { id: "s9", description: "Send refund directly" }
]);
```

Evidencia esperada:

```
{
  "s9": {
    "status": "failed",
    "error_type": "unsupported_step",
    "step_id": "s9",
    "description": "Send refund directly"
  }
}
```

Este failure es valioso porque es reproducible. Un loop en producción nunca debe hacer que el revisor infiera si un plan se detuvo porque terminó, falló la validación, agotó el presupuesto o chocó con policy.

Revisión Final del Lab

Antes de continuar, verifica el control loop:

Check	Evidencia
El planning está separado de la ejecución	<code>planner.ts</code> produce pasos; <code>executor.ts</code> ejecuta operaciones acotadas.
Los pasos son identificables	Cada paso puede ser nombrado, trazado y conectado a un resultado.
El ejecutor se mantiene determinista	El mismo plan soportado produce el mismo resultado.
El failure tiene una superficie	Tasks no soportados o mal formados pueden retornar un failure estructurado.
La condición de stop es explícita	El lab puede explicar por qué terminó la ejecución.

Registra la forma del plan, el result map y un failure path en la hoja de finalización del laboratorio.

Extensión para Producción

Agrega controles de loop antes de usar este pattern en producción:

- máximo de pasos
- máximo de reintentos

- razón de stop
- checkpoint después de cada paso
- resultado de error estructurado
- revisión humana para pasos de alto riesgo

El planning solo es útil cuando la ejecución es acotada e inspeccionable.

Puente a Producción

Usa esta tabla al adaptar el laboratorio a un workflow de producto:

Concepto del Lab	Versión en Producción
Plan object	Plan schema versionado con owner y risk class.
Step IDs	Step IDs del workflow con trace spans y retry policy.
Result map	State durable con checkpoint, status y error class.
Deterministic executor	Tool o workflow node con timeout, idempotency y policy gate.
CLI run	Request envelope con actor, tenant, budget, trace ID y stop reason.

El primer hito en producción es un plan reproducible. Si un failure no puede reproducirse desde el state y trace, el loop sigue siendo solo un demo.

Mapeo Entre Frameworks

- En LangGraph, el planning y la ejecución pueden ser nodos separados conectados por un graph state compartido.
- En Mastra AI, la misma separación puede aparecer como un workflow que coordina pasos de agent y tool.
- En sistemas estilo AutoGen, un manager agent puede proponer un plan mientras funciones executor realizan trabajo acotado.
- En CrewAI, un flow puede poseer la secuencia mientras crews o agents manejan tasks delegados.

Capítulos Relacionados

- [Goals and State](#)
- [Self-Healing Workflows](#)
- [Durable Workflows](#)

Lab 03 - Construye Agentic RAG

Descarga la [hoja de ejercicios guiados de Lab 03 Agentic RAG](#), la [hoja de entrega del laboratorio](#) y la [hoja de preparación para producción del laboratorio](#) antes de comenzar.

Objetivo

Construye el límite de retrieval detrás de un sistema Agentic RAG: recupera evidencia delimitada, inyéctala en el context, responde a partir de esa evidencia y rechaza o escala cuando la evidencia no es suficiente.

Qué Vas a Usar

- Lenguaje: Python
- Framework/runtime: stack de retrieval estilo LangChain/LangGraph con FAISS y embeddings de Hugging Face
- Lección agnóstica de framework: retrieval produce evidencia delimitada; la generación debe mantenerse basada en esa evidencia.
- Capítulos de patrones: [Semantic Recall and RAG](#), [Agentic RAG Systems](#)
- Carpeta fuente: `context-engineering-pattern/`
- Descarga: [semantic-recall-rag.zip](#)
- Archivo principal: `context-engineering-pattern/langgraph_python_example/rag_example.py`
- Comparación nativa: `native-framework-examples/langgraph-research-rag/` ([descargar](#))

Presupuesto de Tiempo para el Ejercicio

Estas estimaciones asumen que el entorno de Python ya está listo.

Ejercicio	Tiempo	Salida
Configuración y retrieval base	10-15 min	Consulta, context recuperado y respuesta generada o fallback.
Cambia el corpus o la consulta	10-15 min	Evidencia de que la respuesta cambia porque retrieval cambió.
Ejecuta la verificación de evidencia faltante	10-15 min	Nota de rechazo o escalamiento para preguntas no soportadas.
Esquematiza el contrato de fuente	15 min	Campos del paquete fuente para owner, freshness, access y allowed use.
Compara el grafo nativo	15 min	Mapeo de pasos de retrieval a nodos del grafo y eval gates.

Configuración

El ejemplo de RAG está basado en Python. Puede ejecutarse en modo local fallback sin una clave de model ni dependencias opcionales de vector-store. Instala todos los requisitos cuando quieras el camino de FAISS y Hugging Face.

Desde la raíz del repositorio:

```
python3 -m venv .venv-rag
source .venv-rag/bin/activate
pip install -r context-engineering-pattern/langgraph_python_example/requirements.txt
```

Para generación de respuestas en vivo, configura `MISTRAL_API_KEY`. Sin la clave, el script imprime una respuesta local determinista a partir del context recuperado.

Ejecútalo

```
python3 context-engineering-pattern/langgraph_python_example/rag_example.py
```

Inspecciona el Código

Abre `context-engineering-pattern/langgraph_python_example/rag_example.py` y localiza:

- `docs`: el tiny demo corpus.
- `HuggingFaceEmbeddings`: el embedding model.
- `FAISS.from_texts`: el vector index.
- `retrieve(query)`: el retrieval boundary.
- `chat_mistral(messages)`: el generation boundary.

La decisión clave de diseño es mantener la evidencia recuperada separada de la instrucción. El model debe responder a partir del context, no de una memory vaga.

Cambia Una Cosa

Agrega un nuevo documento a la lista `docs`:

```
{"content": "Agentic RAG uses retrieval, planning, tool use, and verification around a knowledge base."}
```

Luego cambia la consulta:

```
query = "What makes RAG agentic?"
```

Ejercicios Guiados

Usa estos ejercicios para convertir el laboratorio en un evidence pack. Cada ejercicio debe dejar una nota en la hoja guiada.

Ejercicio	Tiempo	Acción	Listo Cuando
Baseline retrieval trace	10 min	Ejecuta el script sin cambios y copia la consulta, context recuperado y respuesta.	Puedes señalar el texto exacto que entró a generación.
Grounding change	15 min	Agrega el documento Agentic RAG de arriba y cambia la consulta.	La respuesta cambia porque la evidencia recuperada cambió.
Missing-evidence check	15 min	Haz una pregunta que el tiny corpus no pueda soportar, como <code>What is the refund policy?</code> .	Puedes explicar por qué este demo debe rechazar o escalar en producción.
Source contract sketch	15 min	Reescribe un documento como un source packet con <code>source_id</code> , <code>owner</code> , <code>freshness</code> y <code>allowed</code> .	Puedes nombrar los metadatos que un retriever real debe llevar.
Native graph comparison	15 min	Abre <code>native-framework-examples/langgraph-research-rag/research_rag_graph.py</code> y compara sus nodos con este laboratorio.	Puedes mapear access policy, retrieval, filtering, answer synthesis, escalation y evals a nodos del grafo.

Para la verificación de evidencia faltante, no consideres una respuesta fluida como éxito. La señal de éxito es que puedes identificar el claim no soportado y el control de producción que debería bloquearlo.

Resultado Esperado

Sin una clave de model, el script debe imprimir una respuesta local fallback:

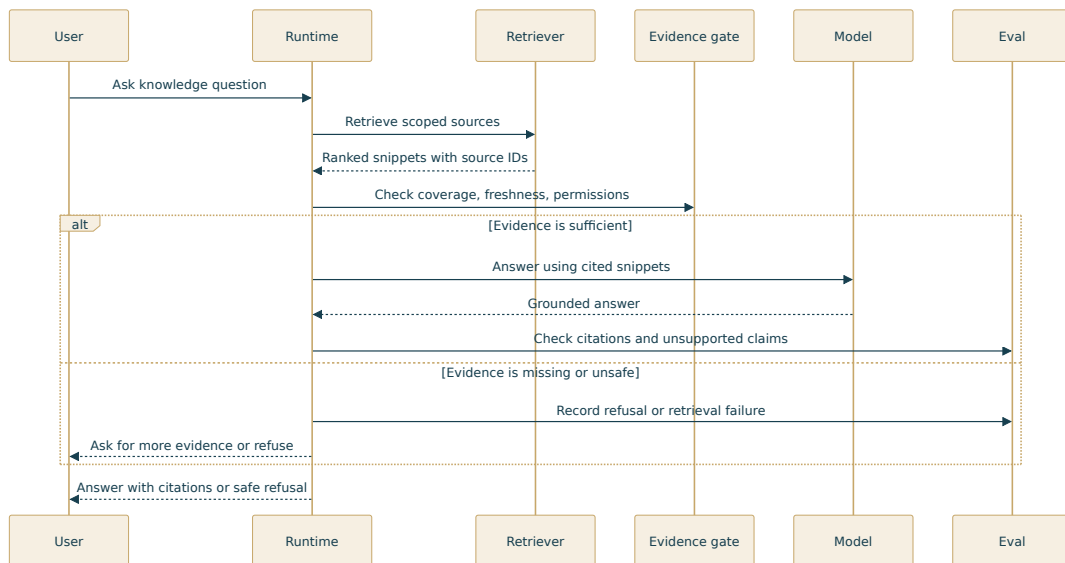
```
Answer: Local fallback answer from retrieved context: Agentic systems are autonomous AI systems.

Prompt engineering improves LLM outputs.
```

Con `MISTRAL_API_KEY`, la respuesta debe reflejar el context recuperado a través de la llamada de generación de Mistral.

Después de agregar el nuevo documento y cambiar la consulta, la respuesta debe reflejar el nuevo documento. Si la respuesta no cita o usa la evidencia recuperada, el retrieval boundary no está haciendo suficiente trabajo.

Usa este flujo como el modelo de aceptación del laboratorio: cada respuesta debe pasar por retrieval delimitado, verificación de evidencia y evaluación de citas antes de llegar al usuario.



Lab Review Gate

Antes de continuar, verifica la ruta de la evidencia:

Verificación	Evidencia
Retrieval es visible	El código tiene un límite claro de <code>retrieve(query)</code> .
La evidencia está delimitada	El texto recuperado está separado de las instrucciones y la lógica de generación.
El fallback local funciona	El script puede responder desde el context recuperado sin una provider key.
La respuesta usa evidencia	La respuesta cambia cuando el corpus y la consulta cambian.
Se reconoce la evidencia faltante	Puedes nombrar lo que el demo haría mal cuando no existe un documento aprobado.
Se nombran controles de producción	El laboratorio identifica source IDs, ACLs, freshness, verificación de citas y comportamiento de rechazo.

Registra la consulta, evidencia recuperada, respuesta y brecha de evidencia faltante en la hoja de entrega del laboratorio.

Ejercicio de Source-Grounding

Usa esta mini-revisión después del grounding change:

Pregunta	Respuesta
¿Qué documento debe recuperarse primero?	El documento que menciona Agentic RAG.
¿Qué oración de la respuesta depende de ese documento?	La oración que explica que Agentic RAG envuelve retrieval con planning, tool use o verification.

Pregunta	Respuesta
¿Qué oración no estaría soportada sin él?	Cualquier claim sobre “qué hace agentic a RAG.”
¿Qué agregaría producción?	Source ID, etiqueta de cita, regla de freshness, filtro de access y eval de cita.

Si la respuesta no cambia después de agregar el nuevo documento, inspecciona primero el retrieval boundary. El problema puede ser la redacción de la consulta, el ranking, metadatos faltantes o un context packet que da al model muy poca evidencia.

Ejercicio de Falla Intencional

Cambia la consulta a un tema fuera del corpus:

```
query = "What is the refund policy?"
```

El script de ejemplo aún construirá un context packet a partir de los documentos más cercanos disponibles. Eso es aceptable para un demo mínimo, pero no es comportamiento aceptable en producción. Registra la falla como:

```
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
```

Esta es la lección clave del laboratorio: retrieval no es suficiente. El sistema necesita un evidence gate que pueda decir “el corpus no soporta esta respuesta.”

Extensión para Producción

Agrega controles de producción:

- IDs de documentos
- URLs de origen
- marcas de tiempo de frescura
- filtros de control de acceso
- requisitos de citación
- verificaciones de prompt-injection en el texto recuperado
- rechazo cuando falta evidencia

Agentic RAG no es solo búsqueda vectorial. Es un loop controlado alrededor de retrieval, evidence, uso de tools y verificación.

Puente a Producción

Usa esta tabla al adaptar el laboratorio a un sistema de conocimiento real:

Concepto del Lab	Versión de Producción
Lista pequeña de docs	Corpus gobernado con IDs de origen, propietarios, ACLs, frescura y reglas de retención.
Índice demo FAISS	Servicio de retrieval gestionado con versión de índice, filtros, evals y rollback.
Texto recuperado	Paquete de evidence con metadatos de origen, notas de fuente omitida y etiquetas de citación.
Llamada de generación Mistral	Síntesis de respuesta restringida a evidence aprobada.
Inspección manual	Citation eval, stale-source eval, forbidden-source eval y prueba de escalamiento.

El primer hito de producción no es un índice más grande. Es una ruta de retrieval que puede rechazar respuestas no soportadas y explicar qué evidence utilizó.

Extensión con Framework Nativo

Después de que el ejemplo determinista de RAG pase, compáralo con `native-framework-examples/langgraph-research-rag/`. El slice nativo mapea el mismo límite a un LangGraph `StateGraph` con nodos para access policy, retrieval, filtrado de fuentes, síntesis de respuesta, escalamiento y citation evals.

Estándar de cumplimiento: el grafo nativo debe demostrar que las fuentes obsoletas y prohibidas se omiten antes de la síntesis de respuesta, y que la falta de evidence aprobada escala en vez de producir una respuesta no soportada.

Mapeo entre Frameworks

- En LangGraph, retrieval puede ser un nodo del grafo que actualiza el state con evidence antes de la generación.
- En LangChain, esto se mapea a retrievers, document loaders, vector stores y chains o runnables.
- En Mastra AI, retrieval se convierte en una knowledge o tool capability usada por un agent o workflow.
- En CrewAI, un rol de research puede recuperar evidence, pero el flujo aún debe validar grounding y citaciones.

Capítulos Relacionados

- Context Engineering
- Knowledge-Bound Agents
- Working Memory
- Research RAG Agent Capstone

Lab 04 - Construye Comunicación A2A entre Agents

Descarga la [hoja de trabajo de finalización del laboratorio](#) y la [hoja de trabajo de preparación para producción](#) antes de comenzar.

Objetivo

Construye un límite de comunicación tipado entre dos agents. Un agent solicita trabajo, otro agent acepta, rechaza, reporta error o recibe cancelación mediante mensajes explícitos.

Lo Que Vas a Usar

- Lenguaje: TypeScript
- Framework/runtime: runtime orientado a protocolos con validación de schema JSON usando Ajv
- Lección agnóstica de framework: la comunicación entre agents necesita sobres tipados, correlation IDs, refusal states, error states y cancellation.
- Capítulo de pattern: [A2A Agent Interoperability](#)
- Carpeta fuente: `agent-to-agent-communication-pattern/`
- Descarga: [a2a-agent-interoperability.zip](#)
- Archivos principales:
 - `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`

Presupuesto de Tiempo del Ejercicio

Estas estimaciones asumen que las dependencias ya están instaladas.

Ejercicio	Tiempo	Resultado
Configuración y prueba base del protocolo	8 min	Salida de prueba A2A exitosa.
Inspecciona el schema de mensajes	10-12 min	Notas sobre task IDs, states, refusals, errors y cancellation.
Cambia una ruta de mensaje	10-15 min	Un resultado visible de éxito, rechazo, entrada inválida o cancelación.
Revisa la propiedad de fallas del protocolo	10-15 min	Dueño y regla de manejo para trabajos mal formados o cancelados.
Completa el mapeo para producción	5-10 min	Notas sobre transporte, auth, replay y versiones de schema.

Configuración

Desde la raíz del repositorio:

```
npm install
```

Ejecútalo

Ejecuta la prueba del protocolo:

```
npm run a2a:test
```

Ejecuta el demo:

```
npm run a2a:run
```

Inspecciona el Código

Abre `agent-to-agent-communication-pattern/src/agent_a.ts` y `agent-to-agent-communication-pattern/src/agent_b.ts`.

Busca:

- handshake
- task request
- task response
- refusal
- invalid input error
- cancellation
- validación de schema con Ajv

Cambia Una Cosa

En la prueba, agrega una segunda task request válida con un ID diferente:

```
a.requestTask('t5', 'sum', { a: 10, b: 15 });
```

Ejecuta:

```
npm run a2a:test
```

Resultado Esperado

El receptor debe procesar la nueva task sin confundirla con los task IDs existentes. Si faltan correlation IDs, el progreso y los resultados se vuelven difíciles de asociar.

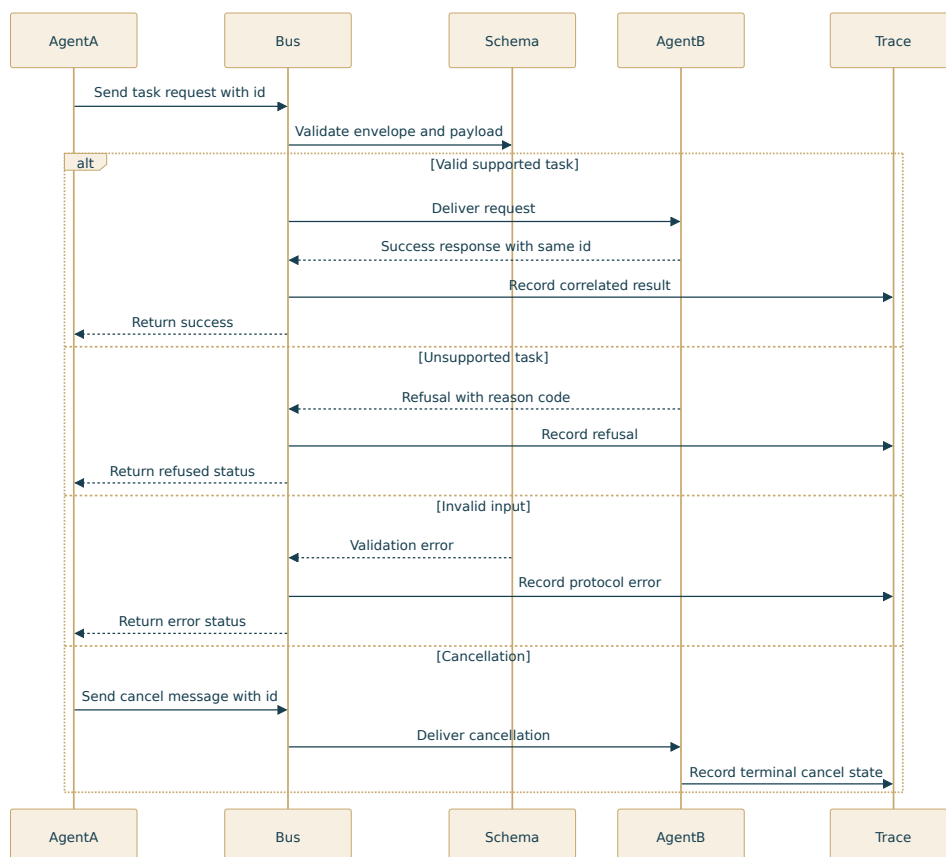
La prueba debe imprimir cuatro resultados:

```
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
```

El demo debe imprimir el camino exitoso:

```
AgentA received response: { id: 't1', status: 'success', output: { sum: 7 } }
```

Usa este flujo como modelo de aceptación para el laboratorio. La comunicación A2A es saludable solo cuando el protocolo maneja éxito, rechazo, entrada inválida y cancelación con la misma disciplina de correlación.



Puerta de Revisión del Lab

Antes de continuar, verifica el límite del protocolo:

Verificación	Evidencia
Los mensajes son tipados	El schema A2A valida task requests, responses, refusals, errors y cancellations.
La correlación es explícita	Cada task tiene un ID que conecta request, progress, result y cancellation.
El rechazo es un state de primera clase	El receptor puede rechazar trabajos no soportados sin simular éxito.
La entrada inválida está controlada	Payloads mal formados producen un resultado de error.
La cancelación es observable	El trabajo cancelado tiene un mensaje y un state terminal.

Registra los task IDs, tipos de mensajes y un caso de entrada inválida en la hoja de trabajo de finalización del laboratorio.

Extensión para Producción

Antes de usar A2A entre servicios, agrega:

- authentication
- authorization
- idempotency keys
- task leases
- retry policy
- durable task state
- audit logs
- encriptación a nivel de transporte

A2A es un límite de protocolo, no solo un agent llamando otra función.

Puente a Producción

Usa esta tabla al adaptar el laboratorio a comunicación agent a agent entre servicios:

Concepto del Lab	Versión en Producción
In-memory bus	Transporte autenticado con entrega durable y retry policy.
Task ID	Correlation ID más idempotency key y trace ID.
JSON schema	Contrato de protocolo versionado con pruebas de compatibilidad.
Refusal message	Negación informada por policy con reason code y registro de auditoría.
Cancellation message	Lease, timeout, motivo de cancelación y comportamiento de limpieza.

El primer hito de producción es un contrato de mensajes que pueda sobrevivir retries, refusal, cancellation y replay sin perder propiedad.

Mapeo Entre Frameworks

- En LangGraph, esto se parece a un handoff de graph a graph o de servicio a servicio con state tipado.
- En Mastra AI, esto corresponde a límites de servicio o workflow alrededor de llamadas de agent.
- En sistemas estilo AutoGen, corresponde a mensajes estructurados entre agents en vez de solo chat libre.
- En CrewAI, corresponde a handoffs de task y salidas de roles que aún requieren schema, identidad y correlación de trace.

Capítulos Relacionados

- [Secure Agent Communication](#)
- [MCP-first Tool Use](#)
- [Open Personal Agent Architectures](#)

Lab 05 - Construye un Multi-Agent Supervisor

Descarga la [hoja de trabajo de finalización del laboratorio](#) y la [hoja de trabajo de preparación para producción](#) antes de comenzar.

Objetivo

Construye la forma de supervisión detrás de los sistemas multi-agent: un coordinador posee el goal, delega trabajo acotado, recopila los outputs de los workers y produce la respuesta final.

Lo Que Vas a Usar

- Lenguaje: TypeScript
- Framework/runtime: ejemplo de manager/worker estilo AutoGen
- Lección agnóstica de framework: un supervisor posee la descomposición, contratos de worker, merge policy y síntesis final.
- Capítulos de patrones: [Supervisor / Worker](#), [Task Delegation](#)
- Carpeta fuente: `hierarchical-agent-pattern/`
- Descarga: [supervisor-worker.zip](#)
- Archivo principal: `hierarchical-agent-pattern/autogen_typescript_example/hierarchical_agent.ts`

Presupuesto de Tiempo del Ejercicio

Estas estimaciones asumen que las dependencias ya están instaladas.

Ejercicio	Tiempo	Output
Configuración y ejecución base	8-10 min	Output determinista o en vivo del supervisor.
Inspeccionar contratos de supervisor y worker	12 min	Notas sobre ownership del goal, límites de roles y comportamiento de merge.
Cambiar delegación o parsing	12 min	Evidencia de que el output acotado del worker sigue controlando la síntesis.
Revisar falla de agregación	10 min	Regla de manejo para output de worker faltante, conflictivo o débil.
Completar mapeo a producción	5-10 min	Contrato de worker, merge policy, trace y notas de final-owner.

Configuración

Este ejemplo puede ejecutarse en dos modos:

- Modo determinista local cuando `MISTRAL_API_KEY` no está configurado.

- Modo en vivo de Mistral cuando `MISTRAL_API_KEY` está presente en `.env`.

Desde la raíz del repositorio:

```
npm install
cp .env.example .env
```

Configura `MISTRAL_API_KEY` en `.env` solo si deseas la ruta de modelo en vivo. Déjalo sin configurar si deseas el output determinista del laboratorio que aparece abajo.

Ejecútalo

```
npm run hierarchical-agent
```

Cuando se te pida, ingresa un goal como:

```
Draft a short plan for evaluating an agentic RAG prototype.
```

Inspecciona el Código

Abre `hierarchical-agent-pattern/autogen_typescript_example/hierarchical_agent.ts` y encuentra:

- el manager prompt
- los worker prompts
- extracción de subtasks
- aggregation prompt
- ruta de respuesta final

El supervisor posee la descomposición y la aceptación final. Los workers no deben redefinir silenciosamente el goal.

Cambia Una Cosa

Cambia la instrucción del manager para que pida tres subtasks en lugar de dos. Luego inspecciona la lógica de parsing:

```
const subTasks = managerPlan.match(/Sub-task [12]: (.*)/g) || [];
```

Actualiza el regex para que el código pueda recolectar el tercer subtask.

Resultado Esperado

El manager debe producir un plan, los workers deben producir resultados acotados y la agregación final debe combinar los outputs de los workers. Si el parsing es frágil, el supervisor pierde trabajo.

Sin `MISTRAL_API_KEY`, la ruta determinista debe mostrar esta forma:

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.

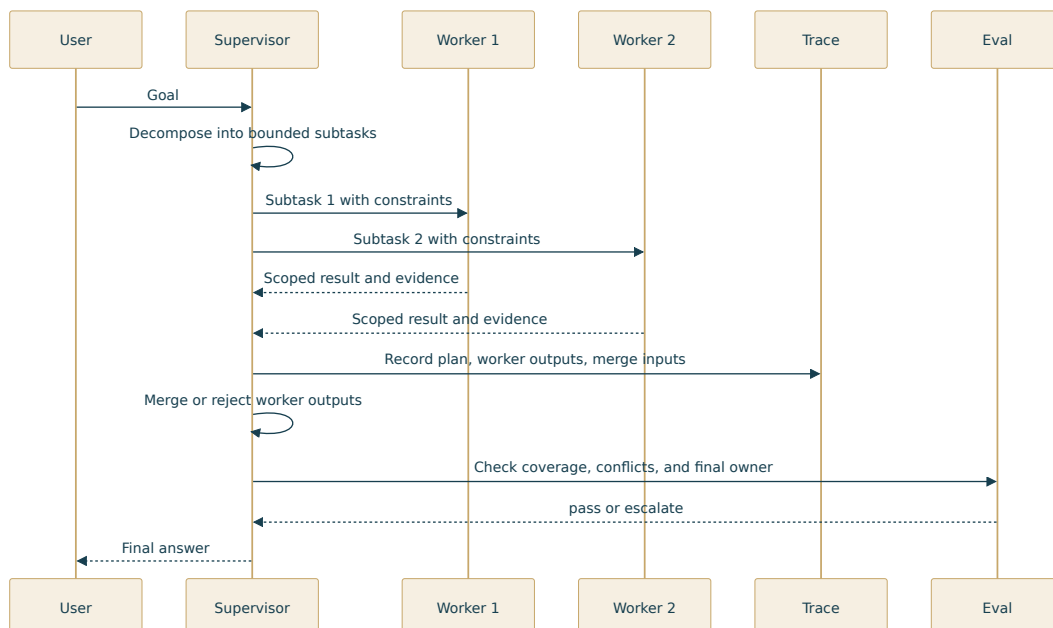
Final Aggregated Result for User:

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.

El texto exacto puede variar en modo modelo en vivo. La señal requerida es la misma: un plan del manager, outputs acotados de los workers y una respuesta final que use esos outputs.



Usa este flujo como el modelo de aceptación del laboratorio. El supervisor posee la descomposición, contratos de worker, merge policy, evidencia de trace y aceptación final.

Puerta de Revisión del Lab

Antes de continuar, verifica el límite de supervisión:

Check	Evidencia
El supervisor posee el goal	El manager prompt descompone el trabajo sin permitir que los workers redefinan la solicitud.
El trabajo del worker está acotado	Cada worker recibe un subtask específico y retorna un resultado acotado.
La agregación es explícita	La respuesta final usa los outputs de los workers en vez de ocultar el comportamiento de merge.
El riesgo de parsing es visible	El cambio de regex muestra por qué los planes en lenguaje natural necesitan structured output.
La ownership de fallas está nombrada	El laboratorio puede explicar quién maneja el output de worker faltante, conflictivo o de baja calidad.

Registra el plan del manager, los outputs de los workers, el resultado de la agregación y la brecha de parsing en la hoja de trabajo de finalización del laboratorio.

Extensión a Producción

Reemplaza el parsing de subtasks en lenguaje natural por structured output:

- `subtasks: Array<{ id, role, objective, constraints, expected_output }>`
- permisos específicos por worker
- timeout por worker
- merge policy
- judge o evaluator
- escalamiento humano para desacuerdos

Los sistemas multi-agent necesitan contratos sólidos. Más agents sin merge policy usualmente significa más modos de falla.

Puente a Producción

Usa esta tabla al adaptar el laboratorio a un workflow multi-agent de producción:

Concepto del Lab	Versión de Producción
Manager prompt	Supervisor policy con task schema, reglas de ruteo y criterios de aceptación.
Worker prompts	Contratos de rol con permisos, tools, timeouts y expected output schema.
Regex subtask parsing	Structured output con validación y retry-on-invalid behavior.

Concepto del Lab	Versión de Producción
Aggregation prompt	Merge policy con manejo de conflictos, atribución de fuente y final owner.
Console result	Trace con spans por worker, costo, latencia, razón de parada y resultado de evaluator.

El primer hito de producción no es agregar más agents. Es probar que el trabajo delegado puede ser acotado, fusionado, rechazado y auditado.

Mapeo Entre Frameworks

- En LangGraph, esto corresponde a un coordinator graph que enruta trabajo a nodos o subgrafos especializados.
- En Mastra AI, esto corresponde a workflows que coordinan agents y tools bajo un solo runtime.
- En sistemas estilo AutoGen, este es el patrón explícito de conversación manager/worker.
- En CrewAI, esto corresponde a crews con tasks específicos por rol, aunque un flow aún debe poseer el state y la aceptación final.

Capítulos Relacionados

- [Parallel Agents](#)
- [Debate and Consensus](#)
- [CrewAI Flows and Crews](#)

Lab 06 - Agrega Observability y Evals

Descarga la [hoja de ejercicios guiados de Lab 06 observability and evals](#), la [hoja de finalización del lab](#) y la [hoja de preparación para producción del lab](#) antes de comenzar.

Objetivo

Convierte los ejemplos en algo que puedas evaluar. Un agent de producción necesita trace data, regression tasks, expected outcomes y failure review antes de necesitar más autonomía.

Qué Vas a Usar

- Lenguaje: TypeScript
- Framework/runtime: pruebas neutrales sobre framework usando los ejemplos del repositorio
- Lección agnóstica de framework: los evals deben inspeccionar trayectorias, tool calls, policy outcomes y stop reasons, no solo las respuestas finales.
- Capítulos de patrones: [Observability and Evals](#), [Evaluator-Optimizer](#)
- Carpeta fuente: `observability-and-evals-pattern/`
- Descarga: [observability-and-evals.zip](#)
- Comandos de prueba existentes:
 - `npm run observability:test`
 - `npm run plan:test`
 - `npm run a2a:test`
 - `npm run mcp:test`
 - `npm test`

Presupuesto de Tiempo para el Ejercicio

Estas estimaciones asumen que las dependencias ya están instaladas.

Ejercicio	Tiempo	Resultado
Configuración y ejecución base de eval	8-10 min	Salida de prueba acotada exitosa.
Inspecciona el contrato de trace	10-12 min	Campos requeridos de trace e invariante protegido.
Revisa casos negativos	15-20 min	Razón de falla, severidad y nota de bloqueo de release.
Esquematiza CI y puertas de incident-to-eval	15-20 min	Comando, responsable, umbral y nombre de fixture futuro.
Completa la puerta de revisión	5-10 min	Notas en la hoja de trabajo sobre trace, eval y brecha de producción.

Configuración

Desde la raíz del repositorio:

```
npm install
```

Ejecútalo

Ejecuta las verificaciones determinísticas:

```
npm run observability:test  
npm run plan:test  
npm run a2a:test  
npm run mcp:test
```

Luego ejecuta el suite completo:

```
npm test
```

Inspecciona el Código

Usa los archivos de prueba como el primer dataset de eval:

- `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`

Cada prueba verifica un contrato:

- los traces contienen correlation IDs, model spans, stop reasons y policy decisions para tool spans
- el planning produce pasos ejecutables
- A2A maneja success, refusal, error y cancel
- el uso de herramientas MCP descubre tools y retorna datos útiles

Cambia Una Cosa

Agrega un caso negativo a la prueba de A2A enviando input mal formado con un nuevo task ID:

```
a.requestTask('t6', 'sum', { a: null, b: 10 } as any);
```

Ejecuta:

```
npm run a2a:test
```

Resultado Esperado

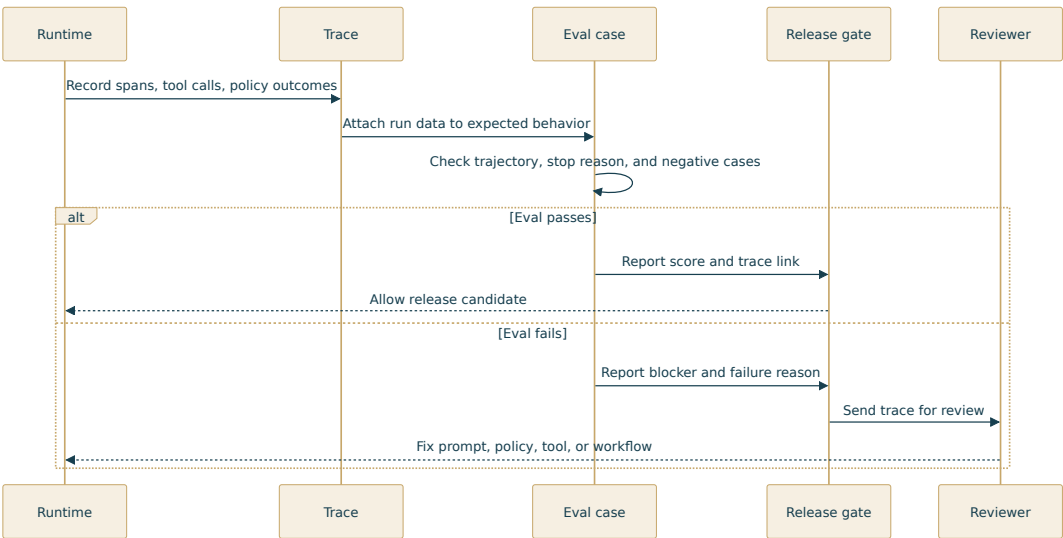
El sistema debe reportar un error outcome, no un resultado exitoso. En un dataset de eval, los casos negativos son tan importantes como los caminos felices.

La puerta de contrato de trace debe imprimir:

```
Trace contract test OK
```

Esa prueba demuestra ambos lados del límite de eval: un trace completo pasa, y un tool span exitoso sin una policy decision falla.

Usa este flujo como el modelo de aceptación para el lab. Observability captura lo que ocurrió; los evals deciden si la ejecución es lo suficientemente segura para liberar.

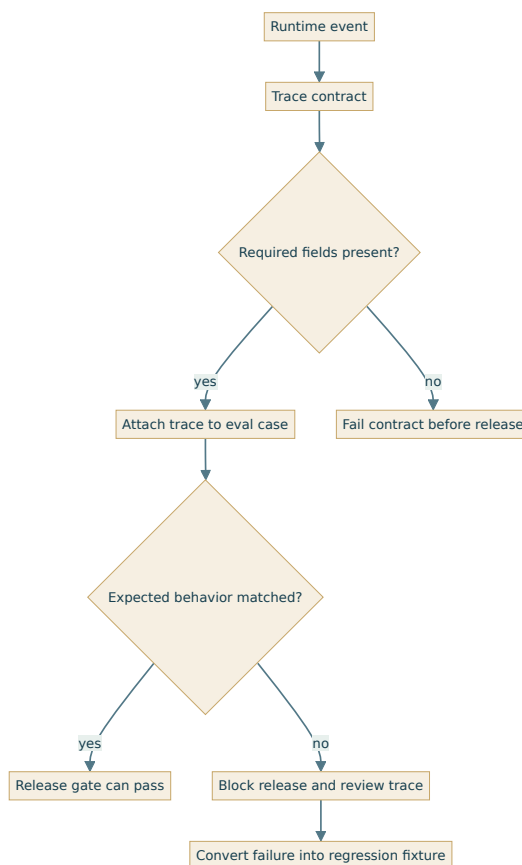


Ejercicios Guiados

Usa estos ejercicios para hacer visible el límite de eval.

Ejercicio	Tiempo	Qué Hacer	Evidencia a Guardar
Línea base del contrato de trace	10 min	Ejecuta <code>npm run observability:test</code> .	La salida exitosa del contrato y los campos protegidos por la prueba.
Falla por falta de policy	10 min	Inspecciona <code>missingPolicyTrace</code> en <code>trace-contract.spec.ts</code> .	La razón exacta de la falla: <code>policy decision for tool span span_tool_missing_policy</code> .
Caso negativo de A2A	15 min	Agrega el input de A2A mal formado de este lab y vuelve a ejecutar <code>npm run a2a:test</code> .	Error outcome, task ID y por qué debe bloquear el release.

Ejercicio	Tiempo	Qué Hacer	Evidencia a Guardar
Esquema de puerta CI	10 min	Decide qué comando debe bloquear el release para este segmento.	<code>npm test</code> , una prueba acotada o un futuro comando de eval con responsable.
Nota de incident-to-eval	10 min	Elige una falla del lab y conviértela en un fixture de regresión.	Nombre del fixture, severidad, resultado esperado y campo de trace.



Ejercicio de Eval Intencionalmente Fallido

En `observability-and-evals-pattern/trace-contract.spec.ts`, el objeto `missingPolicyTrace` modela una llamada exitosa a tool sin una policy decision. Eso es un bloqueador de release porque la tool usó autoridad sin una decisión auditable de allow, deny, approval o escalation.

Revisa la falla como si viniera de producción:

Pregunta	Respuesta a Registrar
¿Qué span falló?	<code>span_tool_missing_policy</code>
¿Qué invariante se rompió?	Cada tool span debe tener una policy decision.
¿Qué debe hacer CI?	Fallar la puerta de eval.

Pregunta	Respuesta a Registrar
¿Qué debe corregir el responsable?	Agregar la policy decision antes de que el resultado de la tool cuente como válido.

Puerta de Revisión del Lab

Antes de continuar, verifica el límite de eval:

Verificación	Evidencia
Se aplica el contrato de trace	<code>npm run observability:test</code> acepta un trace completo y rechaza una policy decision faltante en tool.
Las pruebas verifican contratos	Las pruebas de planning, A2A y MCP afirman el comportamiento, no solo que los comandos se ejecutan.
Existen casos negativos	El input de A2A mal formado retorna un error outcome.
Se inspecciona la trayectoria	Las pruebas revisan pasos, mensajes, descubrimiento de tools u outcomes antes del texto final.
El riesgo de release es visible	El lab nombra qué fallas bloquearían un release de producción.
Se implica la propiedad de eval	Cada prueba protege un límite específico de pattern.

Registra los comandos, salida de éxito/falla, caso negativo y límite protegido en la hoja de finalización del lab.

Extensión para Producción

Crea un registro de trace y eval para cada ejecución:

```
{
  "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
}
```

Luego agrega puertas de release:

- no schema failures
- no missing trace IDs

- no unexpected tool calls
- no regression en golden tasks
- no unresolved safety o policy failures

Observability registra lo que ocurrió. Los evals deciden si es lo suficientemente bueno para liberar.

Puente a Producción

Usa esta tabla al adaptar el lab a la evaluación en producción:

Concepto del Lab	Versión en Producción
Comando de prueba	Puerta de release ligada a cambios en prompt, model, tool, policy, memory o workflow.
Fixture de prueba	Caso de eval versionado con responsable, severidad y razón de falla.
Salida de consola	Reporte de eval con éxito/falla, enlace al trace, score, umbral y estado de bloqueo.
Caso negativo de A2A	Fixture de regresión para schema failure y trayectoria insegura.
Full <code>npm test</code>	Puerta de CI más monitoreo canario y workflow de incident-to-eval.

El primer hito de producción no es un dashboard más grande. Es un eval bloqueante que detecta una trayectoria mala conocida antes del release.

Mapeo Entre Frameworks

- En LangGraph, los traces y checkpoints te permiten inspeccionar rutas de nodos, cambios de state y condiciones de detención.
- En Mastra AI, los evals y observability deben conectar eventos de agent, tool, workflow y memory.
- En sistemas estilo AutoGen, los message logs deben convertirse en traces estructurados antes de ser datos de eval confiables.
- En CrewAI, los outputs de task y flow necesitan casos de eval que verifiquen el comportamiento de roles y la síntesis final.

Capítulos relacionados

- [Agent Development Lifecycle](#)
- [Evaluation-Driven Agent Development](#)
- [Policy Enforcement](#)

Lab 07 - Empaqueta Agents, Tools, Workflows, Memory y Evals

Descarga la [hoja de ejercicios guiados de runtime packaging del Lab 07](#), la [hoja de finalización del laboratorio](#) y la [hoja de preparación para producción del laboratorio](#) antes de comenzar.

Objetivo

Usa una estructura de runtime en TypeScript al estilo Mastra para empaquetar las mismas responsabilidades que construiste desde cero: decisión del agent, ejecución de tools, control del workflow, memory, eventos de trace y evals.

Qué Vas a Usar

- Lenguaje: TypeScript
- Framework/runtime: runtime packaging estilo Mastra
- Lección agnóstica de framework: un framework puede empaquetar primitivas, pero tu aplicación sigue siendo dueña del state, policy, contratos de tool, traces y criterios de aceptación.
- Capítulos de patrones: [Mastra Runtime](#), [Building a Minimal Agent Runtime](#), [Observability and Evals](#)
- Archivos fuente:
 - `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`

Tiempo Estimado del Ejercicio

Estas estimaciones asumen que las dependencias ya están instaladas.

Ejercicio	Tiempo	Resultado
Configuración y ejecución base del runtime	10 min	Salida de demo y test.
Mapear límites del runtime package	15 min	Notas de propietario para agent, tool, workflow, memory, trace y eval.
Inspeccionar orden de tools y tools prohibidos	20 min	Trace de orden de tools más resultado de eval de tool prohibido.
Ejecutar rollback y comparación nativa	20-30 min	Deshabilitar ruta y mapeo al slice nativo estilo Mastra.
Completar mapeo de producción	10-20 min	Notas de release para policy, export de trace, eval gate y riesgo de upgrade.

Configuración

Desde la raíz del repositorio:

```
npm install
```

Este laboratorio es determinista y no requiere una model key. El objetivo es el runtime packaging, no la calidad del model.

Ejecútalo

```
npm run mastra-runtime:demo  
npm run mastra-runtime:test
```

Resultado Esperado

El comando de test debe imprimir:

```
Mastra-style runtime packaging tests OK
```

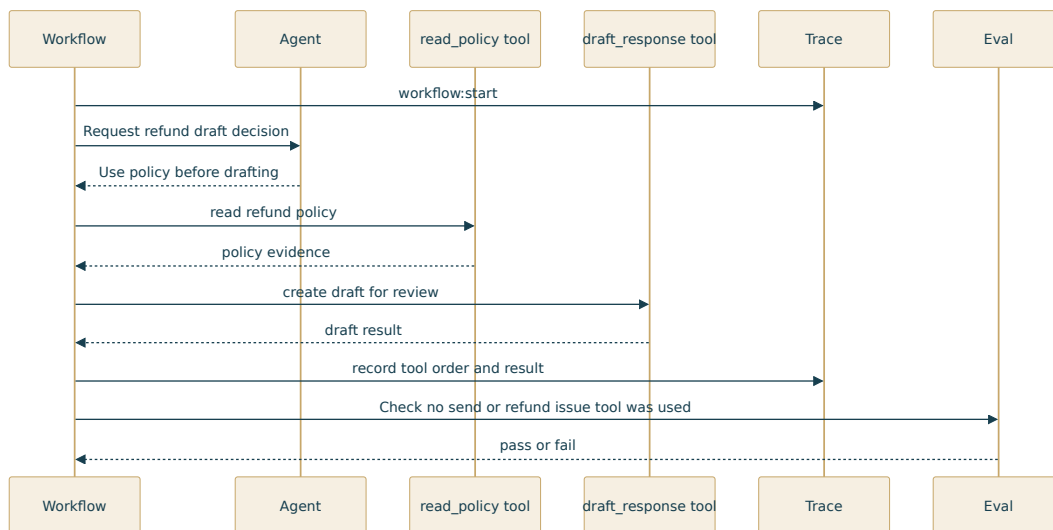
La ejecución también debe probar estas señales de comportamiento:

- policy se lee antes de redactar;
- el draft se crea para revisión, no se envía;
- los pasos del workflow y los resultados de tools quedan registrados en traces;
- el eval falla si se usa un tool de envío directo o refund-issue.

El comando demo debe incluir esta evidencia:

```
{  
  "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"  
  }  
}
```

El JSON exacto también incluye los campos de memory y trace. Usa los campos `toolCalls`, `result` y `evaluation.status` como verificación rápida.



Usa este flujo como modelo de aceptación del laboratorio. Un framework puede empaquetar el runtime, pero la aplicación sigue siendo dueña del orden de tools, policy de efectos secundarios, evidencia de trace y evals de release.

Punto de comparación nativo de Mastra:

```

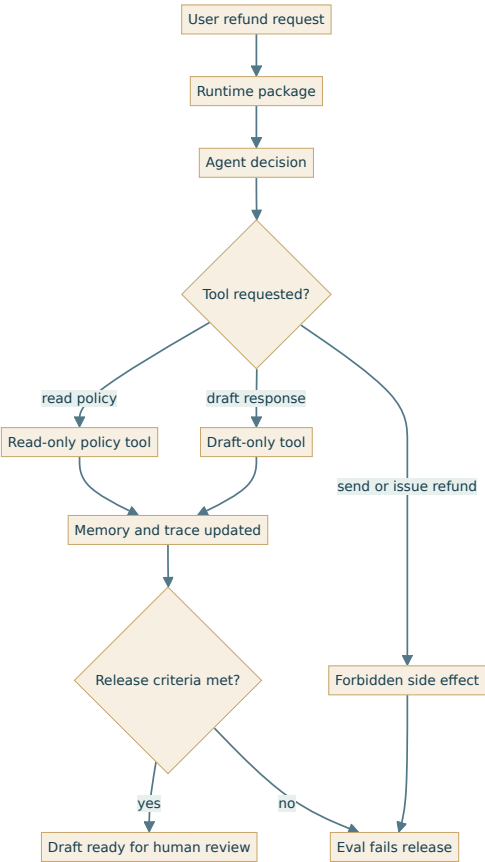
native-framework-examples/mastra-refund/
download: /downloads/native-mastra-refund.zip
agent: refundDraftAgent
workflow: refundDraftWorkflow
tools: refund_policy.retrieve, refunds.create_draft
eval gate: refund_draft_no_money_movement
  
```

Ejercicios Guiados

Usa estos ejercicios para inspeccionar qué posee el runtime package y qué sigue siendo responsabilidad de la policy del producto.

Ejercicio	Tiempo	Qué Hacer	Evidencia a Guardar
Mapa de runtime package	15 min	Inspecciona <code>Agent</code> , <code>Tool</code> , <code>WorkflowStep</code> , <code>RuntimeState</code> y <code>evaluateRuntime</code> .	Una frase sobre qué posee cada límite.
Trace de orden de tools	10 min	Ejecuta <code>npm run mastra-runtime:demo</code> e inspecciona <code>toolCalls</code> .	<code>read_policy</code> antes de <code>draft_response</code> .
Eval de tool prohibido	10 min	Lee el caso negativo <code>refunds.issue_refund</code> en <code>runtime_packaging.spec.ts</code> .	Estado de eval y razón de fallo.
Ejercicio de rollback	10 min	Nombra la forma más rápida de deshabilitar la capability riesgosa.	Feature flag, eliminación de tool, deshabilitar ruta de workflow o rollback de deployment.

Ejercicio	Tiempo	Qué Hacer	Evidencia a Guardar
Comparación nativa	20 min	Compara el laboratorio determinista con <code>native-framework-examples/mastra-refund/</code> .	Qué partes corresponden a agent, workflow, tools, trace y eval.



Ejercicio de Deployment y Rollback

Trata el laboratorio como candidato a producción. Antes de lanzar un runtime real, escribe la acción de rollback que deshabilita la capability de mayor riesgo sin eliminar todo el servicio.

Capability	Pregunta de Rollback	Evidencia Aceptable
<code>read_policy</code>	¿Puede el sistema volver a un snapshot estático de policy?	Config flag, policy ID versionada o fuente en caché.
<code>draft_response</code>	¿Se puede deshabilitar la creación de draft mientras la consulta de policy sigue disponible?	Cambio en el manifest de tool o flag de ruta de workflow.
<code>send_message</code>	¿Puede seguir siendo imposible el envío directo salvo aprobación explícita?	Eval de tool prohibido y approval gate.
<code>refunds.issue_refund</code>	¿Puede el movimiento de dinero mantenerse fuera de este runtime?	Falta de registro de tool más eval fallido si aparece.

Inspecciona el Código

Abre `mastra-runtime-pattern/typescript/src/runtime_packaging.ts` y encuentra estos límites:

- `Agent`: posee instrucciones y propone la siguiente decisión.
- `Tool`: posee la ejecución tipada de capability.
- `WorkflowStep`: posee la secuencia determinista de runtime.
- `RuntimeState`: posee memory, traces, llamadas a tools y resultado.
- `evaluateRuntime`: revisa la trayectoria, no solo el texto final.

El código local es intencionalmente pequeño, pero el mapeo sigue la forma de producción: agents, workflows, tools, memory, traces y evals pertenecen a un solo runtime package.

Cambia Una Cosa

Agrega un nuevo tool llamado `send_message` con un efecto secundario directo, luego actualiza `evaluateRuntime` para que el eval falle si ese tool aparece en `toolCalls`. El test del repositorio también verifica el caso de efecto secundario de refund:

```
{ name: "refunds.issue_refund", input: { amount: 42 } }
```

Lección esperada: el registro de tools en el framework no es permiso. La policy de la aplicación sigue decidiendo qué es seguro.

Verifica

Ejecuta:

```
npm run mastra-runtime:test
```

Compara la salida con el resultado esperado antes de pasar a la extensión de producción.

Revisión del Laboratorio

Antes de continuar, verifica el runtime package:

Revisión	Evidencia
Las responsabilidades del runtime son visibles	Agent, tool, workflow, memory, trace y eval tienen tipos separados en el código.
El orden de tools está restringido	La consulta de policy ocurre antes de crear el draft.
Los efectos secundarios están bloqueados	El eval falla si aparece un tool de envío directo o refund issue.
El trace explica el comportamiento	La ejecución registra workflow, decisión del agent y eventos de tools.

Revisión	Evidencia
La propiedad del framework es limitada	La policy del producto y los criterios de release quedan fuera de los defaults del framework.

Registra la salida del comando, eventos de trace, orden de tools y resultado de eval en la hoja de finalización del laboratorio.

Extensión de Producción

Antes de lanzar una implementación real de Mastra, agrega:

- schemas reales de tools y validación de input;
- policy de retry de workflow y compensación;
- controles de retención, redacción y eliminación de memory;
- fixtures de scorer/eval ligados a release gates;
- export de trace al backend de observability;
- approval gates para tools de escritura y comunicación externa.

Production Bridge

Usa esta tabla cuando adaptes el laboratorio a un Mastra runtime real:

Lab Concept	Production Version
<code>Agent</code> type	Agent route con instrucciones versionadas, tools permitidas y razones de detención.
<code>Tool</code> type	Typed capability con permiso, clase de side-effect, timeout e idempotencia.
<code>WorkflowStep</code>	Paso de workflow durable con soporte para retry, compensación, checkpoint y aprobación.
<code>RuntimeState</code>	State de ejecución persistido con tenant, actor, trace ID, budget y memory policy.
<code>evaluateRuntime</code>	Release gate vinculado a incidentes conocidos, policy rules y trayectorias prohibidas.

El primer hito de producción es un runtime slice que puede deshabilitarse, reproducirse y evaluarse sin leer los detalles internos del framework.

Native Framework Extension

Después de que el laboratorio determinista pase, porta una vertical slice a un proyecto real de Mastra. Usa [Real Framework Setup Notes](#) para los comandos actuales de configuración y compara tu trabajo con el ejemplo del repositorio en `native-framework-examples/mastra-refund/`.

Pasos para el port nativo:

1. crea un proyecto Mastra;
2. crea un agent para la decisión de borrador de reembolso;
3. crea tools tipados para policy lookup y creación de borrador;

- 4. crea un workflow que invoque policy lookup antes de la creación del borrador;
- 5. agrega un eval que falle si se llama a un tool de send o refund-issue;
- 6. exporta un trace con eventos de workflow, agent, tool, policy y eval;
- 7. documenta el rollback para deshabilitar el tool de borrador.

Mantén estos assets fuera del código exclusivo del framework:

Asset	Why
tool manifest	la autoridad del producto debe ser revisable sin leer el framework
policy rules	la policy debe ejecutarse antes del tool
eval fixtures	los release gates deben sobrevivir cambios en el framework
trace schema	operaciones necesitan campos estables entre runtimes
ADR	la elección del framework necesita un owner registrado y un rollback path

Estándar de finalización: el proyecto nativo prueba el mismo comportamiento que este laboratorio y enlaza al [Support Refund Agent capstone](#). Una ejecución nativa de Mastra no está completa solo porque un agent responde; el workflow debe preservar el orden de tools, la side-effect policy, la evidencia en trace y los eval gates.

Troubleshooting

Symptom	Likely Cause	Fix
<code>mastra dev</code> cannot find provider credentials	<code>.env</code> falta o las variables del proveedor no están configuradas	Copia <code>.env.example</code> , configura las claves del proveedor y reinicia el dev server.
workflow drafts before policy lookup	el orden de los pasos del workflow es incorrecto	Mantén la obtención de policy como el primer paso y haz que el paso de borrador consuma la salida de policy.
eval passes after adding <code>refunds.issue_refund</code>	la lista de tools prohibidos está incompleta	Agrega el nombre del tool al release eval y vuelve a ejecutar la validación.
trace explica el texto final pero no el orden de tools	el trace es solo a nivel de model	Emite eventos de workflow, tool, policy y eval por separado.

Cross-Framework Mapping

- En LangGraph, esto se convierte en graph state, nodes, tools, checkpoints y evals alrededor de ejecuciones del grafo.
- En Mastra AI, estas preocupaciones se empaquetan como agents, workflows, tools, memory, observability y evals.
- En sistemas estilo AutoGen, el runtime package se divide entre manager, workers, functions, messages y transcript evals.

- En CrewAI, el empaquetado equivalente se divide entre flow state, crew tasks, agent roles, tools y evaluación a nivel de flow.

Related Chapters

- [Mastra Runtime](#)
- [Framework Selection](#)
- [Building a Minimal Agent Runtime](#)
- [Production Evaluation Feedback Loops](#)
- [Support Refund Agent Capstone](#)

Lab 08 - Modela Flows, Crews, Roles y Task Contracts

Descarga la [hoja de trabajo de finalización del laboratorio](#) y la [hoja de trabajo de preparación para producción](#) antes de comenzar.

Objetivo

Usa una estructura Python al estilo CrewAI para separar la propiedad del flow de la colaboración del crew. El flow es dueño del state, la secuencia, la evaluación y la aceptación final. El crew es responsable de trabajo especializado y acotado.

Qué Vas a Usar

- Lenguaje: Python
- Framework/runtime: flows y crews al estilo CrewAI
- Lección agnóstica de framework: el valor multi-agent proviene de los límites de roles, task contracts, flow state y aceptación explícita, no de agregar más agents.
- Capítulos de patrones: [CrewAI Flows and Crews](#), [Choosing Multi-Agent Topology](#), [Supervisor / Worker](#)
- Archivos fuente:
 - `crewai-flows-and-crews-pattern/python/flow_crew.py`
 - `crewai-flows-and-crews-pattern/python/test_flow_crew.py`

Tiempo Estimado por Ejercicio

Estas estimaciones asumen que las dependencias ya están instaladas.

Ejercicio	Tiempo	Resultado
Configuración y ejecución base del flow	10 min	Demo del flow y salida de prueba.
Inspeccionar límites de flow y roles	15 min	Notas sobre propiedad del state, salida de roles y aceptación.
Cambiar un rol o caso de aceptación	20 min	Un state aceptado o rechazado visible.
Verificar y solucionar problemas	10-15 min	Prueba exitosa o causa de falla registrada.
Completar mapeo a producción	10-20 min	Notas sobre checkpoints, permisos, validadores y ruta de fallback.

Configuración

Desde la raíz del repositorio:

```
npm install
```

Este laboratorio solo utiliza código de la biblioteca estándar de Python. Es intencionalmente determinista para que puedas inspeccionar el límite de control sin variabilidad del model.

Ejecútalo

```
npm run crewai-flow  
npm run crewai-flow:test
```

Resultado Esperado

El comando de prueba debe imprimir:

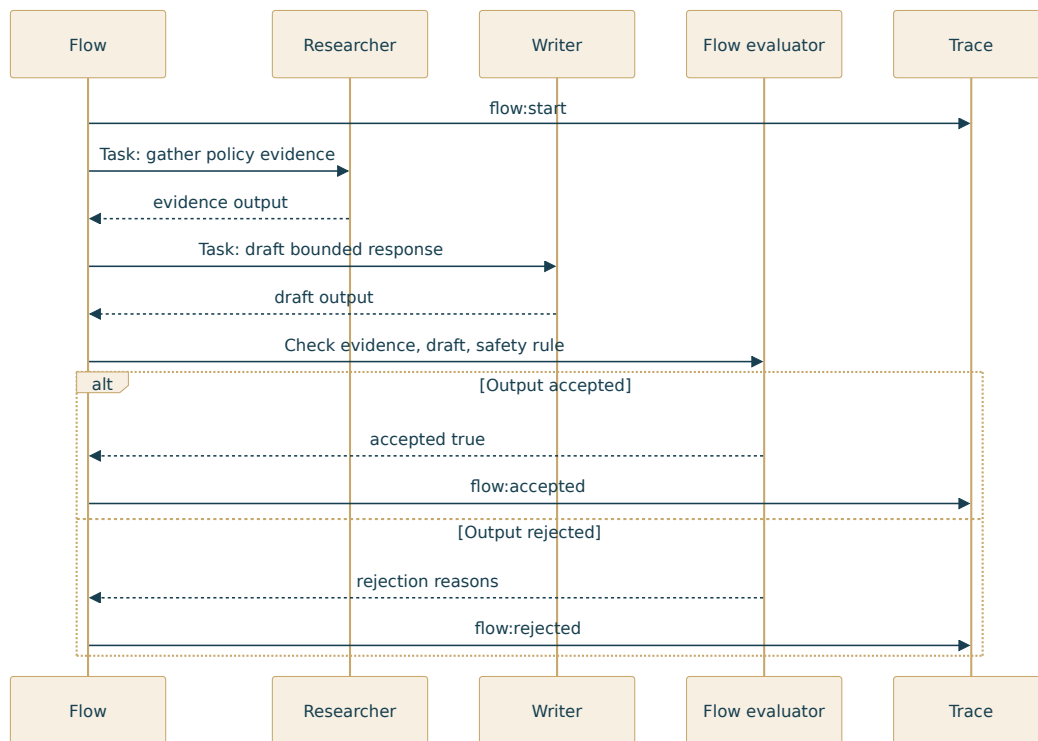
```
CrewAI-style flow and crew tests OK
```

La ejecución también debe demostrar estas señales de comportamiento:

- los roles de researcher y writer producen salidas separadas;
- el flow trace registra inicio, kickoff del crew, evaluación y aceptación;
- el flow rechaza la salida que no cumple la regla de aceptación;
- el state final tiene un único dueño responsable.

El comando demo debe incluir esta estructura de flow aceptado:

```
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'}
```



Usa este flow como el modelo de aceptación del laboratorio. El crew realiza trabajo especializado, pero el flow es dueño del state, la evaluación, la aceptación final y la evidencia en el trace.

Punto de comparación con CrewAI nativo:

```

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
  
```

Inspecciona el Código

Abre `crewai-flows-and-crews-pattern/python/flow_crew.py` y encuentra estos límites:

- `FlowState`: la fuente de verdad propiedad del flow.
- `Agent`: un rol con un goal específico.
- `Task`: una asignación acotada a un rol.
- `Crew`: la unidad de colaboración que ejecuta tasks.
- `evaluate_flow`: la puerta de aceptación a nivel de flow.

La regla de diseño importante es que la salida del crew no se acepta automáticamente. El flow la evalúa antes de establecer `accepted`.

Cambia Una Cosa

Cambia la salida del writer para que omita `do not promise payment`.

Rechazo esperado:

```
Crew output rejected by the flow.
{'status': 'fail', 'reasons': ['flow did not accept crew output']}
```

La prueba del repositorio ahora ejecuta esa ruta de rechazo con un crew inseguro inyectado. Eso mantiene el laboratorio determinista mientras demuestra que el flow, no el crew, es dueño de la aceptación final.

Luego restaura la salida del writer y vuelve a ejecutar:

```
npm run crewai-flow:test
```

Verifica

Compara la salida con el resultado esperado anterior antes de pasar a la extensión de producción.

Puerta de Revisión del Lab

Antes de continuar, verifica el límite entre flow y crew:

Verificación	Evidencia
El flow es dueño del state	<code>FlowState</code> registra solicitud, salidas de roles, trace, aceptación y motivo de detención.
Los roles son distintos	Las salidas de researcher y writer difieren por responsabilidad.
La salida del crew es validada	<code>evaluate_flow</code> rechaza la salida que viola la regla de aceptación.
La aceptación tiene un solo dueño	El flow, no el crew, establece la aceptación final.
El rechazo es observable	Una mala salida del writer produce un state rechazado y motivo.

Registra la ejecución aceptada, la rechazada, las salidas de roles y el flow trace en la hoja de trabajo de finalización del laboratorio.

Extensión para Producción

Antes de lanzar una implementación real de CrewAI, agrega:

- entradas y salidas tipadas para tasks;
- límites explícitos de permisos por rol;
- schemas y validadores de resultados del crew;
- checkpoints del flow y capacidad de reanudación;
- registros de trace por agent, task, crew y paso del flow;
- casos de evaluator para desacuerdo, evidencia faltante, trabajo duplicado y mala síntesis.

Puente a Producción

Usa esta tabla al adaptar el laboratorio a una implementación real de CrewAI:

Concepto del Lab	Versión en Producción
<code>FlowState</code>	Flow state durable con tenant, actor, trace ID, checkpoint y datos de rollback.
Rol <code>Agent</code>	Contrato de rol con permisos, tools, ruta de model y schema de salida esperada.
<code>Task</code>	Asignación tipada con contrato de entrada, criterios de aceptación, timeout y dueño.
<code>Crew</code>	Unidad de colaboración acotada con spans de trace y presupuesto de costo.
<code>evaluate_flow</code>	Puerta de liberación a nivel de flow para evidencia, cobertura de roles, síntesis y seguridad.

El primer hito de producción no es un crew más elaborado. Es un flow que puede rechazar colaboraciones débiles y explicar por qué.

Extensión para Framework Nativo

Después de que el laboratorio determinista pase, porta una vertical slice a un Flow y Crew reales de CrewAI. Usa las [Notas de configuración de framework real](#) para orientación y compara tu trabajo con el ejemplo del repositorio en `native-framework-examples/crewai-delivery/`.

Pasos para portabilidad nativa:

1. crea un proyecto CrewAI con un Flow;
2. define el Flow state para solicitud, salidas de roles, aceptación, motivo de detención y trace ID;
3. crea agents de researcher, writer y reviewer solo si cada rol tiene un contrato distinto;
4. define entradas de tasks y formas de salida esperadas;
5. valida la salida del crew antes de mutar el Flow state;
6. agrega evals para evidencia faltante, mala síntesis, desacuerdo de roles y salida rechazada;
7. documenta el rollback para deshabilitar la ruta Crew manteniendo la ruta determinista del Flow.

El Flow sigue siendo el dueño responsable:

Límite	Dueño
state	Flow
task assignment	Flow
role behavior	Crew agents
output validation	Flow
final acceptance	Flow
rollback	runtime o plataforma de despliegue

Estándar de finalización: el proyecto nativo demuestra el mismo comportamiento de roles y aceptación que este laboratorio y enlaza al [Multi-Agent Delivery Workflow capstone](#). Una ejecución nativa de CrewAI no está completa solo porque el crew devuelve texto; el Flow debe preservar salidas separadas de roles y validarlas antes de la aceptación.

Solución de problemas

Síntoma	Causa probable	Solución
crew devuelve solo una respuesta agregada	los outputs de cada task no se conservan por separado	Lee los outputs a nivel de task y copia los resultados de planner, reviewer y tester en el state de Flow por separado.
Flow acepta outputs débiles o duplicados de roles	la aceptación se basa en la finalización, no en la validación	Agrega validadores a nivel de Flow antes de establecer <code>accepted</code> .
las credenciales del provider fallan	faltan variables del provider de modelo de CrewAI	Configura el entorno del provider requerido por tu configuración de CrewAI.
los roles se superponen mucho	los agents no tienen contratos distintos	Elimina el rol o reescribe los contratos de task hasta que cada rol cambie el riesgo o la calidad del output.
rollback no es claro	la delegación está incrustada en el camino principal	Mantén un camino de fallback determinista de un solo dueño fuera de la ruta de Crew.

Mapeo entre frameworks

- En LangGraph, el flow puede ser un graph mientras que cada rol corresponde a un nodo o subgraph.
- En Mastra AI, la misma estructura puede modelarse como workflows que coordinan agents y tools.
- En sistemas estilo AutoGen, el crew se parece a agents especialistas dirigidos por un manager, pero el flow aún necesita aceptación final.
- En CrewAI, los flows proveen control estructurado mientras los crews realizan trabajo colaborativo delegado.

Capítulos relacionados

- [CrewAI Flows and Crews](#)
- [Choosing Multi-Agent Topology](#)
- [Task Delegation](#)
- [Observability and Evals](#)
- [Multi-Agent Delivery Workflow Capstone](#)

From-Scratch Mini-Framework Track

Esta ruta construye un agent runtime educativo y pequeño desde los primeros principios. No es un reemplazo para LangGraph, Mastra AI, sistemas tipo AutoGen, CrewAI ni motores de workflow para producción. Es una forma de entender los primitivos que esos sistemas empaquetan.

Descarga la [hoja de trabajo de finalización del laboratorio](#) y la [hoja de trabajo de preparación para producción](#) antes de comenzar la ruta.

Comienza con [Building a Minimal Agent Runtime](#) para la teoría. Luego usa estos laboratorios si quieres implementar los primitivos por tu cuenta.

La implementación mantenida en TypeScript está en `minimal-agent-runtime/typescript`. Úsala como solución de referencia mientras implementas tu propia versión pequeña en TypeScript o Python.

```
npm run mini-runtime
npm run mini-runtime:test
```

Qué construyes

Lab	Componente	Lección principal
Lab 09 - Minimal Agent Loop	State, decisiones, loop, razones de detención	Un agent es un loop controlado, no un prompt.
Lab 10 - Tool Registry and Policy Gate	Tools, policy, resultados que requieren aprobación	La ejecución de tools es un límite de software.
Lab 11 - Context, Memory, Trace, and Evals	Context packets, memory con alcance, trace events, trajectory evals	El comportamiento del runtime debe ser inspeccionable y testeable.

Contrato de aprendizaje

Al finalizar la ruta, deberías poder explicar:

- qué posee el loop;
- qué pertenece al state;
- por qué las decisiones del model son propuestas;
- cómo los tools se convierten en capabilities específicas;
- por qué el policy es independiente de los prompts;
- cómo se ensamblan los context packets;
- qué debe registrar un trace;
- por qué los trajectory evals detectan fallas que los final-answer evals no ven.

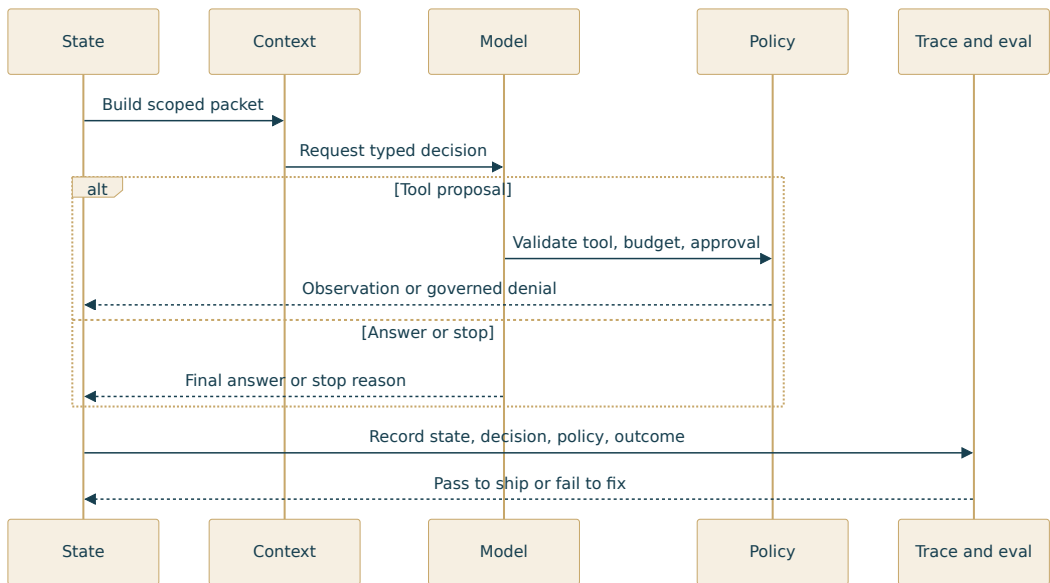
Elección de implementación

Los laboratorios están escritos para que puedas implementar el runtime en TypeScript o Python. Usa TypeScript si quieres discriminated unions fuertes y un camino directo a los ejemplos existentes del repositorio. Usa Python si prefieres una implementación compacta para enseñanza antes de mapear las ideas a LangGraph o CrewAI.

Mantén la implementación pequeña. El objetivo no es un framework de propósito general. El objetivo es un runtime que puedas entender en una sola sesión.

La referencia en TypeScript es intencionalmente libre de dependencias. La versión en Python debe preservar los mismos contratos: las decisiones son propuestas tipadas, el policy corre antes de la ejecución, los traces registran el camino y los evals inspeccionan la trayectoria.

Usa este modelo para conectar los tres laboratorios. El mini-runtime es valioso porque cada paso tiene un responsable: el state mantiene la ejecución, el context delimita lo que ve el model, el policy controla la acción, el trace registra el comportamiento y los evals juzgan la trayectoria.



Track Review Gate

Antes de considerar la ruta como completa, verifica los primitivos del runtime:

Check	Evidencia
Existe control del loop	El runtime tiene state, decisiones, observaciones, presupuestos y razones de detención.
Los tools están gobernados	Se prueban búsquedas de tools, decisiones de policy, resultados que requieren aprobación y rechazos por tools desconocidos.
El context es explícito	Los context packets y lecturas de memory tienen alcance definido en vez de estar ocultos en el texto del prompt.
El trace es revisable	Las ejecuciones emiten eventos que explican decisiones, llamadas a tools, resultados de policy y estado final.

Check	Evidencia
Los evals inspeccionan la trayectoria	Las pruebas pueden fallar caminos inseguros incluso si el texto final parece plausible.

Registra los comandos, salidas, caminos de falla probados y brechas de producción en la hoja de trabajo de finalización del laboratorio.

Puente a producción

Usa esta tabla al comparar tu mini-runtime con un framework real:

Mini-Runtime Primitive	Pregunta para runtime de producción
Loop state	¿Dónde se persiste, migra, reanuda y elimina el state?
Tool registry	¿Cómo se versionan, autorizan, deshabilitan y auditan los tools?
Policy gate	¿Puede el policy detener retrieval, escrituras en memory, tools y respuestas finales antes de la ejecución?
Trace events	¿Pueden los operadores reconstruir una ejecución fallida sin secretos sin procesar?
Trajectory evals	¿Qué evals bloquean cambios en prompt, model, tool, policy, memory y workflow?

La ruta tiene éxito cuando mejora el juicio sobre frameworks. Falla si tienta al equipo a lanzar un framework sin operación.

Advertencia de producción

No lances este mini-runtime como plataforma de producción sin agregar ejecución durable, persistencia, autenticación, autorización, concurrencia, despliegue, integraciones de observability, control de reintentos y respuesta a incidentes.

La pregunta correcta para producción no es “¿podemos construir nuestro propio framework?” sino “ahora que entendemos los primitivos, ¿qué framework o runtime nos da los controles que necesitamos?”

Comparación con frameworks

Después de los laboratorios, compara cada primitivo con frameworks maduros:

- LangGraph: graph state, nodos, edges, checkpoints e interrupts.
- Mastra AI: agents, tools, workflows, memory, evals y runtime packaging.
- Sistemas tipo AutoGen: roles de manager/worker, mensajes y ejecución de funciones.
- CrewAI: flows, crews, tasks, roles y asignación de tools.
- MCP y A2A: límites de protocolo para tools y llamadas agent-to-agent.

Capítulos relacionados

- Building a Minimal Agent Runtime
- Agent Harnesses
- Tool Capability Design
- Observability and Evals

Lab 09 - Construye un Agent Loop Mínimo

Descarga la [hoja de trabajo de finalización del laboratorio](#) y la [hoja de trabajo de preparación para producción](#) antes de comenzar.

Objetivo

Construye el runtime más pequeño y útil: un loop que recibe un goal, solicita una decisión tipada, actualiza el state y se detiene por una razón explícita.

Qué Vas a Usar

- Lenguaje: TypeScript o Python
- Framework/runtime: runtime educativo hecho desde cero
- Lección agnóstica de framework: un agent es un loop controlado con state, decisiones, observaciones, presupuestos y razones de detención.
- Capítulos de patrones: [What Is An Agent?](#), [Agent Loop](#), [Goals and State](#)
- Capítulo teórico: [Building a Minimal Agent Runtime](#)

Tiempo Estimado del Ejercicio

Estas estimaciones asumen que las dependencias ya están instaladas.

Ejercicio	Tiempo	Resultado
Ejecuta el baseline de referencia	10 min	Salida de prueba mini-runtime exitosa.
Implementa o inspecciona el contrato del loop	20-25 min	Límites tipados de state, decisión, observación y razón de detención.
Ejercita casos de fallo y presupuesto	10-15 min	Comportamiento rechazado, bloqueado o con presupuesto agotado.
Revisa trace y razones de detención	10-20 min	Notas sobre límites del runtime y resultados visibles para el llamador.

Configuración

Usa la referencia mantenida en TypeScript o crea tu propio archivo pequeño fuera del código de producción, como `scratch/minimal-agent-loop.ts` o `scratch/minimal_agent_loop.py`.

Archivos de referencia:

- `minimal-agent-runtime/typescript/src/runtime.ts`
- `minimal-agent-runtime/typescript/src/run_demo.ts`
- `minimal-agent-runtime/typescript/test/runtime.spec.ts`

Ejecuta primero la prueba de referencia:

```
npm run mini-runtime:test
```

Este laboratorio no requiere una clave de model. Usa una función determinista `decide` para que puedas probar el runtime sin variabilidad del model.

Contrato del Runtime

Usa esta estructura si implementas la versión más pequeña en 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;  
};
```

La implementación equivalente en Python puede usar dataclasses, diccionarios tipados o diccionarios simples. Mantén los mismos campos.

La referencia mantenida extiende este contrato con campos orientados a producción: `runId`, `toolsCalled`, `memory` con alcance, definiciones de tool, decisiones de policy, context packets, trace events y casos de trajectory eval. Comienza con el contrato pequeño y luego compara tu resultado con `minimal-agent-runtime/typescript/src/runtime.ts`.

Cambio Guiado

Implementa `runAgent(state, decide)`.

El loop debe:

1. llamar a `decide(state)`;
2. registrar una observación para la decisión;
3. regresar con `success` cuando la decisión sea una respuesta;
4. regresar con la razón proporcionada cuando la decisión sea `stop`;
5. continuar para decisiones de tool, o ejecutar tools mediante un registro cuando uses el runtime de referencia;
6. detenerse con `budget_exhausted` cuando `steps` alcance `maxSteps`.

Ejecución Baseline

Usa la demo de referencia:

```
npm run mini-runtime
```

Luego inspecciona el caso de respuesta inmediata en `minimal-agent-runtime/typescript/test/runtime.spec.ts`, o usa una función de decisión que responda de inmediato:

```
const answerImmediately = async (): Promise<Decision> => ({
  kind: "answer",
  text: "done",
});
```

Resultado Esperado

El comando demo debe mostrar una trayectoria leída por policy:

```
{
  "runId": "demo_001",
  "steps": 2,
  "toolsCalled": ["lookup_policy"],
  "answer": "Policy was checked and the draft can be prepared safely.",
  "stopReason": "success"
}
```

También debe incluir trace events con estos tipos:

```
context_built
decision
policy_decision
```

```
tool_result
stop
```

El resultado de eval debe ser:

```
{
  "status": "pass",
  "caseId": "demo-policy-read"
}
```

El caso de respuesta inmediata debe terminar con:

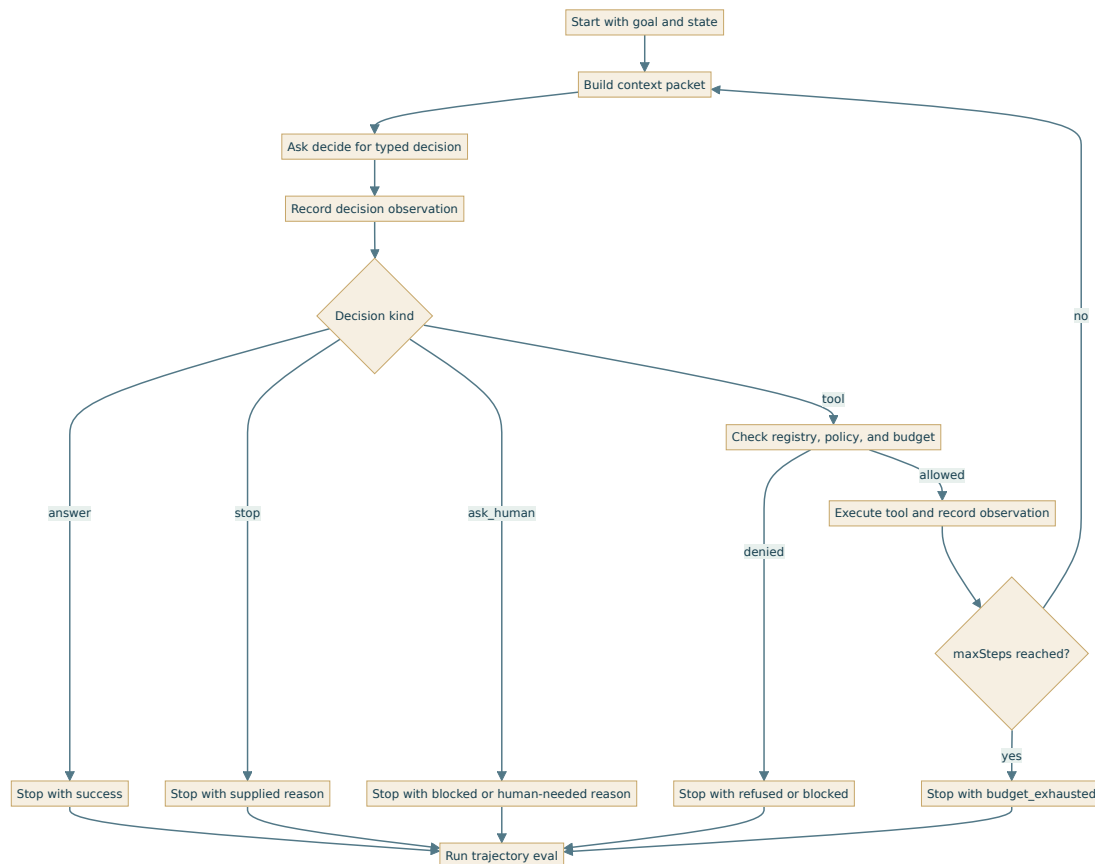
```
stopReason: success
steps: 1
observations: at least one decision observation
```

El caso de tool repetido debe terminar con:

```
stopReason: budget_exhausted
steps: maxSteps
```

La prueba de referencia también cubre:

Caso	Señal Esperada
unknown tool	<code>stopReason: refused</code> ; el tool desconocido no se ejecuta.
write tool sin aprobación	<code>stopReason: blocked</code> ; <code>send_message</code> no se ejecuta.
policy de escritura permisiva	la respuesta final puede parecer exitosa, pero la trajectory eval falla porque se llamó a <code>send_message</code> .
memory con alcance	se incluye memory de task y project; la memory de user-scope se omite como <code>out_of_scope</code> .



Usa este loop como el modelo de aceptación del laboratorio. El runtime, no el model, es dueño de los presupuestos, autoridad de tool, razones de detención, observaciones y evaluación final de la trayectoria.

Caso de Falla

Usa una función de decisión que siempre solicite un tool:

```
const neverStops = async (): Promise<Decision> => ({
  kind: "tool",
  name: "search",
  input: { query: "keep going" },
});
```

Esta es la primera propiedad de seguridad de un agent runtime: el model no puede crear un loop infinito solo por seguir preguntando.

Verifica

Revisa estas afirmaciones manualmente o con la prueba de referencia:

- la respuesta inmediata se detiene con `success`;
- las propuestas de tool repetidas se detienen con `budget_exhausted`;
- cada paso del loop registra una observación;
- el state final contiene una razón de detención.

La prueba de referencia cubre estos casos con decisiones deterministas, así que el resultado es estable entre máquinas.

Puerta de Revisión del Lab

Antes de continuar, verifica el límite del loop:

Revisión	Evidencia
Las decisiones son tipadas	El runtime maneja explícitamente answer, tool, ask-human y stop decisions.
Los cambios de state son visibles	Se registran goal, conteo de pasos, observaciones y razón de detención.
El presupuesto detiene el loop	Las propuestas repetidas de tool terminan con <code>budget_exhausted</code> .
El éxito es explícito	Las respuestas inmediatas se detienen con <code>success</code> .
El model no puede autorizar su propia continuación	<code>maxSteps</code> pertenece al runtime, no a la función de decisión.
La trajectory eval detecta riesgos ocultos	Una ejecución que envía un mensaje puede fallar en eval aunque la respuesta final diga éxito.

Registra la ejecución de respuesta inmediata, ejecución de tool repetido, observaciones y razones de detención en la hoja de trabajo de finalización del laboratorio.

Extensión para Producción

Antes de que este loop pueda ejecutar trabajo real, agrega:

- validación estructurada para decisiones producidas por el model;
- ejecución de tools mediante un registro;
- verificaciones de policy antes de efectos secundarios;
- trace events para cada decisión y detención;
- controles de cancelación y timeout;
- state durable si la ejecución puede pausarse o reanudarse.

Puente a Producción

Usa esta tabla al adaptar el loop a producción:

Concepto de Lab	Versión de Producción
<code>AgentState</code>	State durable de ejecución con actor, tenant, trace ID, presupuesto y datos de checkpoint.

Concepto de Lab	Versión de Producción
<code>Decision</code>	Propuesta de model validada con schema, contexto de policy y metadatos de confianza.
<code>Observation</code>	Trace event con timestamp, span ID, status y clase de redacción.
<code>maxSteps</code>	Presupuesto de runtime con costo, latencia, reintentos, tool y límites de delegación.
<code>stopReason</code>	Razón visible para el operador ligada a evals, dashboards y revisión de incidentes.

El primer hito de producción es un loop que siempre se detiene por una razón que los operadores pueden inspeccionar.

Mapeo Entre Frameworks

- En LangGraph, el loop se expresa mediante recorrido de grafo, actualizaciones de state y edges.
- En Mastra AI, el loop está empaquetado dentro del comportamiento de agent y workflow runtime.
- En sistemas estilo AutoGen, el loop aparece como turnos de mensajes entre manager, worker y ejecutores de tool.
- En CrewAI, el loop se define por la ejecución del flow y el progreso del task.

Capítulos relacionados

- [Construyendo un agent runtime mínimo](#)
- [Agent Loop](#)
- [Goals y State](#)
- [Durable Workflows](#)

Lab 10 - Construye un Tool Registry y Policy Gate

Descarga la [hoja de trabajo de finalización del laboratorio](#) y la [hoja de trabajo de preparación para producción](#) antes de comenzar.

Objetivo

Extiende el mini-runtime con un tool registry y un policy gate. El model puede proponer una llamada a un tool, pero el software decide si el tool existe, si la entrada es aceptable y si la policy permite la ejecución.

Qué Vas a Usar

- Lenguaje: TypeScript o Python
- Framework/runtime: runtime educativo desde cero
- Lección agnóstica de framework: las descripciones de tools no son permisos; el registry y la policy son límites separados en el runtime.
- Capítulos de patterns: [Tool Use](#), [Tool Capability Design](#), [Policy Enforcement](#)
- Laboratorio anterior: [Lab 09 - Minimal Agent Loop](#)

Presupuesto de Tiempo del Ejercicio

Estas estimaciones asumen que el loop de Lab 09 ya está disponible.

Ejercicio	Tiempo	Resultado
Ejecuta el contrato base de tool	10 min	Salida de prueba exitosa del mini-runtime.
Agrega checks de registry y policy	20-25 min	Búsqueda de tool separada, manejo de schema y decisión de policy.
Ejercita casos de denegación y falla	10-15 min	Señales de tool desconocido, aprobación requerida y falla de tool.
Revisa el límite de producción	10-20 min	Notas sobre propiedad del registry, registros de aprobación y campos de trace.

Configuración

Parte del loop de Lab 09. Agrega un objeto registry o un mapa indexado por nombre de tool.

Usa tools deterministas. No llames sistemas externos en este laboratorio.

Archivos de referencia:

- `minimal-agent-runtime/typescript/src/runtime.ts`

- `minimal-agent-runtime/typescript/test/runtime.spec.ts`

Ejecuta la prueba de referencia antes de editar:

```
npm run mini-runtime:test
```

Contrato de Runtime

```
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 };
```

Cambio Guiado

Agrega dos 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) } })),  
  },  
};
```


Luego agrega la policy:

```
function authorize(tool: ToolDefinition): PolicyDecision {
  if (tool.sideEffect === "write") {
    return { status: "approval_required", reason: "write_tool_requires_approval" };
  }
  return { status: "allow" };
}
```

Actualiza el loop para que las decisiones de tool pasen por:

- 1. búsqueda en el registry;
- 2. decisión de policy;
- 3. ejecución del tool solo si está permitido;
- 4. registro de observaciones para resultados rechazados, denegados, aprobación requerida y exitosos.

Ejecución Base

Usa el demo de referencia o una función de decisión que llame a `lookup_policy`.

```
npm run mini-runtime
```

Resultado Esperado

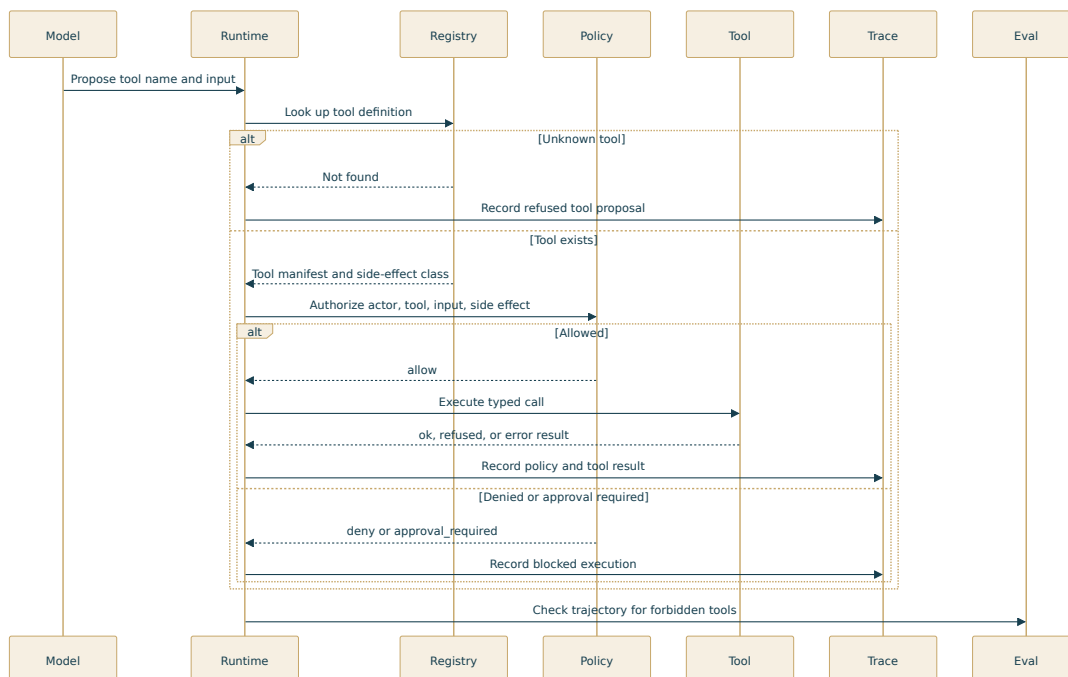
La ruta permitida de read-tool en el demo de referencia debe incluir:

```
toolsCalled: ["lookup_policy"]
observations include: lookup_policy:allow
observations include: lookup_policy:ok
trace includes: policy_decision
trace includes: tool_result
stopReason: success
```

La prueba de referencia cubre estas señales de rechazo y falla:

Caso	Señal Esperada
Tool desconocido <code>delete_customer</code>	<code>stopReason: refused</code> ; <code>delete_customer</code> no se ejecuta.
Write tool <code>send_message</code>	<code>stopReason: blocked</code> ; <code>send_message</code> no se ejecuta porque se requiere aprobación.
Falla en read tool <code>flaky_lookup</code>	<code>stopReason: tool_failure</code> ; el trace incluye <code>upstream_timeout</code> .

Caso	Señal Esperada
Policy permisiva insegura para write	La respuesta final puede ser <code>success</code> , pero el trajectory eval falla porque se llamó a <code>send_message</code> .



Usa este flujo como el modelo de aceptación del laboratorio. El model puede proponer un tool, pero el registry, policy gate, trace y eval deciden si la ejecución está permitida y es revisable.

Casos de Falla

Prueba estos casos:

1. Nombre de tool desconocido.
2. Tool con side effect `write`.
3. Ejecución de tool retorna `error`.
4. Policy permisiva que permite un write tool, por lo que el trajectory eval debe detectar la ruta insegura.

El comportamiento exacto de detención puede variar, pero el runtime no debe ejecutar silenciosamente tools prohibidos o desconocidos.

Verifica

Revisa estas afirmaciones manualmente o con `npm run mini-runtime:test`:

- los tools desconocidos no se ejecutan;
- la policy se ejecuta antes de la ejecución;
- approval-required se representa como un state en el runtime;
- los resultados de tools son structured;
- las observaciones registran tanto rutas permitidas como rechazadas.

La prueba de referencia también demuestra que `send_message` se bloquea antes de la ejecución cuando la policy por defecto requiere aprobación para write tools.

Lab Review Gate

Antes de continuar, verifica el límite de tool:

Check	Evidencia
La búsqueda de tool es explícita	Los nombres de tools desconocidos se rechazan antes de la ejecución.
La policy se ejecuta antes de la ejecución	Los write tools producen <code>approval_required</code> antes de cualquier side effect.
Los resultados son structured	Los resultados de tools usan status <code>ok</code> , <code>refused</code> o <code>error</code> .
La clase de side-effect importa	Los tools de read, draft y write pueden ser gobernados de manera diferente.
Las observaciones preservan la ruta	Los resultados permitidos y rechazados son visibles en las observaciones del runtime.

Registra la ruta permitida de read, el rechazo de tool desconocido, el requisito de aprobación para write-tool y el caso de error en la hoja de trabajo de finalización del laboratorio.

Extensión para Producción

Antes de usar este pattern con tools reales, agrega:

- validación de input schema;
- claves de idempotencia;
- policy de timeout y retry;
- actor, tenant, route y approval context;
- IDs de trace para llamada propuesta, decisión de policy, ejecución y resultado;
- policies separadas para read, draft, write, comunicación externa, movimiento de dinero, memory write y ejecución de código.

Puente a Producción

Usa esta tabla al adaptar el registry a producción:

Concepto de Lab	Versión de Producción
Tool map	Registry de capability versionado con owner, permisos y switch de deshabilitado.
<code>ToolDefinition</code>	Tool manifest con schema, clase de side-effect, timeout, retry y campos de auditoría.
<code>authorize</code>	Policy engine usando actor, tenant, recurso, riesgo, presupuesto y approval context.

Concepto de Lab	Versión de Producción
<code>approval_required</code>	Solicitud de aprobación durable con revisor, expiración, acción exacta y enlace de trace.
Observación de tool	Trace span con input propuesto, decisión, resultado, costo, latencia y redacción.

El primer hito de producción es una ruta de tool que pueda probar por qué la ejecución fue permitida, denegada o pausada.

Mapeo entre Frameworks

- En LangGraph, esto se puede implementar como un nodo de tool protegido por un nodo de policy.
- En Mastra AI, esto se mapea a definiciones de tool más policy a nivel de workflow o tool.
- En sistemas estilo AutoGen, esto se mapea a ejecución de función protegida por el manager o runtime.
- En CrewAI, esto se mapea a tools asignados por rol más restricciones a nivel de flujo.

Capítulos Relacionados

- [Tool Use](#)
- [Tool Capability Design](#)
- [Human Approval Gates](#)
- [Policy Enforcement](#)

Lab 11 - Agrega Context, Memory, Trace y Evals

Descarga la [hoja de trabajo de finalización del laboratorio](#) y la [hoja de trabajo de preparación para producción](#) antes de comenzar.

Objetivo

Haz que el mini-runtime sea inspeccionable. Agrega context packets, lecturas de memory con alcance, trace events y un trajectory eval que pueda fallar incluso cuando la respuesta final parece plausible.

Qué Vas a Usar

- Lenguaje: TypeScript o Python
- Framework/runtime: runtime educativo desde cero
- Lección agnóstica de framework: el comportamiento del runtime debe ser observable y testeable, no solo ejecutable.
- Capítulos de patrones: [Context Engineering](#), [Working Memory](#), [Observability and Evals](#)
- Laboratorios previos: [Lab 09](#), [Lab 10](#)

Presupuesto de Tiempo para el Ejercicio

Estas estimaciones asumen que el runtime de Lab 10 ya está disponible.

Ejercicio	Tiempo	Resultado
Ejecuta el demo base y la prueba	10 min	Salida del comando del runtime pasando.
Agrega context y memory con alcance	20 min	Context packet con referencias de memory incluidas y omitidas.
Agrega trace events y trajectory evals	20 min	Trace path más resultado de eval para comportamiento riesgoso.
Ejercita un caso de falla	10-15 min	Eval fallido o señal de trayectoria insegura.
Completa la revisión de producción	10-25 min	Notas para gobernanza de memory, redacción de trace y replay de incidentes.

Configuración

Parte desde el runtime de Lab 10. Mantén las tools deterministas y pequeñas.

Archivos de referencia:

- `minimal-agent-runtime/typescript/src/runtime.ts`

- `minimal-agent-runtime/typescript/src/run_demo.ts`
- `minimal-agent-runtime/typescript/test/runtime.spec.ts`

Ejecuta el demo de referencia y la prueba:

```
npm run mini-runtime
npm run mini-runtime:test
```

Agrega un 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." },
];
```

Contrato del Runtime

```
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;  
  };  
};
```

Cambio Guiado

Agrega `buildContext(state)` para que cada decisión del model reciba un packet deliberado:

- goal activo;
- resumen compacto del state;
- observaciones recientes;
- tools divulgadas;
- referencias de memory seleccionadas;
- referencias de memory omitidas con razones.

Agrega `recordTrace(event)` y emite trace events para:

1. context construido;
2. decisión propuesta;
3. decisión de policy;
4. resultado de tool;
5. razón de stop.

Ejecución Base

Ejecuta un caso donde el agent llama a una read tool y luego responde. El demo de referencia hace esto con `lookup_policy`.

Resultado Esperado

El comando demo debe mostrar el context packet con alcance. El primer evento `context_built` debe incluir:

```
{  
  "toolsDisclosed": ["draft_message", "lookup_policy", "send_message"],  
  "memoryRefs": ["mem_1", "mem_2"],  
  "omittedRefs": [  
    { "ref": "mem_3", "reason": "out_of_scope" }  
  ]  
}
```

La ruta de la read-tool debe incluir un trace como este:

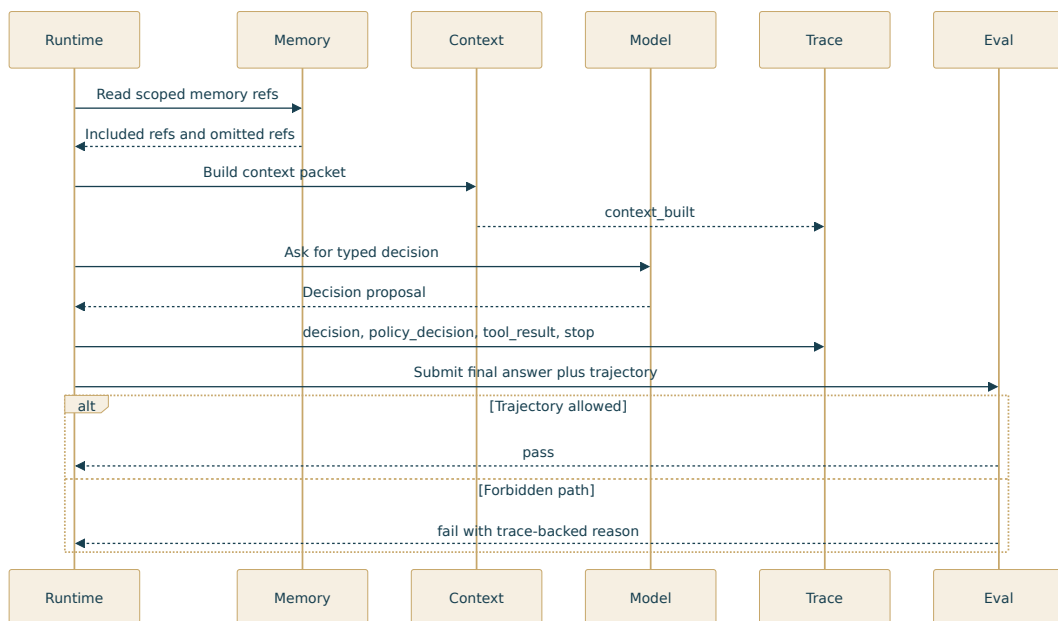
```
context_built  
decision  
policy_decision  
tool_result
```

```
context_built
decision
stop
```

El orden exacto puede variar si tu loop se detiene inmediatamente después de una tool, pero el trace debe mostrar lo suficiente para reconstruir el path.

El caso de trayectoria insegura debe producir:

```
final answer: done
stopReason: success
trajectory eval: fail
reason: forbidden tool was called: send_message
```



Usa este flujo como el modelo de aceptación del laboratorio. Una respuesta final plausible no es suficiente; las omisiones de context, el alcance de memory, los trace events y las trayectorias prohibidas deben ser inspeccionables.

Caso de Falla

Crea un eval donde la respuesta final diga “done”, pero el runtime llamó a una write tool prohibida.

Por esto los evals solo de respuesta final son demasiado débiles para agentic systems.

Verifica

Verifica estas afirmaciones manualmente o con `npm run mini-runtime:test`:

- cada ejecución tiene un trace;
- cada stop tiene una razón de stop;
- el context registra memory incluida y omitida;

- los evals pueden revisar tools llamadas y no llamadas;
- una trayectoria prohibida falla incluso si el texto final parece aceptable.

La prueba de referencia incluye una ejecución intencionalmente insegura donde la respuesta final es `done`, pero el trajectory eval falla porque se llamó una write tool prohibida.

Puerta de Revisión del Lab

Antes de avanzar, verifica el límite de inspeccionabilidad:

Verificación	Evidencia
El context es deliberado	El context packet lista goal, resumen de state, observaciones, tools, referencias de memory y omisiones.
El memory tiene alcance	Las referencias de memory incluidas y omitidas son visibles.
El trace reconstruye el path	Se registran eventos de context, decisión, policy, tool y stop.
Los evals inspeccionan la trayectoria	El uso de tools prohibidas falla incluso si el texto final parece aceptable.
El riesgo de release es visible	El lab puede nombrar qué trace o brechas de eval bloquearían producción.

Registra el context packet, la secuencia de trace, la trayectoria insegura y el resultado del eval en la hoja de trabajo de finalización del laboratorio.

Extensión para Producción

Antes de producción, agrega:

- redacción antes de almacenar trace;
- política de retención y eliminación para memory y traces;
- fixtures de eval versionados con prompts, tools, models y policies;
- workflow de incident-to-eval;
- dashboards para razones de stop, errores de tool, denegaciones de policy, costo y latencia;
- gates de release que bloqueen cambios riesgosos cuando fallen los trajectory evals.

Puente a Producción

Usa esta tabla al adaptar la inspeccionabilidad a producción:

Concepto del Lab	Versión de Producción
<code>ContextPacket</code>	Contrato de context versionado con referencias de evidencia, política de memory, omisiones y presupuesto de tokens.
Memory fixture	Almacenamiento de memory gobernado con retención, eliminación, corrección, consentimiento y alcance de tenant.

Concepto del Lab	Versión de Producción
<code>TraceEvent</code>	Trace span correlacionado con run ID, span ID, parent span, status, costo, latencia y redacción.
<code>EvalCase</code>	Release fixture con owner, severidad, umbral, set de versiones y enlace a incident.
Trayectoria prohibida	Gate de bloqueo para regresiones de policy, tool, memory, retrieval y autonomía.

El primer hito de producción es una ejecución que un segundo ingeniero pueda reproducir, evaluar y explicar sin confiar solo en la respuesta final.

Mapeo Entre Frameworks

- En LangGraph, context y memory son entradas de state, mientras que los traces pueden seguir transiciones de nodos y checkpoints.
- En Mastra AI, memory, evals y observability son capabilities a nivel de runtime que igual deben exponer policy propiedad del producto.
- En sistemas estilo AutoGen, el historial de mensajes debe convertirse en datos estructurados de trace y eval.
- En CrewAI, los registros de flow y task necesitan suficiente estructura para evaluar el comportamiento de roles y la síntesis final.

Capítulos Relacionados

- [Context Engineering](#)
- [Working Memory](#)
- [Observability and Evals](#)
- [Production Evaluation Feedback Loops](#)

Lab 12 - Modela grafos de state, checkpoints e interrupts

Descarga la [hoja de ejercicios guiados de Lab 12 state graph](#), la [hoja de finalización del laboratorio](#) y la [hoja de preparación para producción del laboratorio](#) antes de comenzar.

Objetivo

Usa un grafo de state en Python estilo LangGraph para hacer explícitos el state, los nodos, los edges, los checkpoints, los interrupts y el comportamiento de resume.

Qué Vas a Usar

- Lenguaje: Python
- Framework/runtime: grafo de state estilo LangGraph
- Lección agnóstica de framework: la ejecución basada en grafos es valiosa cuando importan las transiciones de state, ramificación, pausa/resume y observabilidad a nivel de nodo.
- Terminología oficial revisada: LangGraph graph state, nodes, edges, checkpoints e interrupts.
- Capítulos de patterns: [Agent Loop](#), [Goals and State](#), [Durable Workflows](#)
- Archivos fuente:
 - `langgraph-state-graph-pattern/python/state_graph.py`
 - `langgraph-state-graph-pattern/python/test_state_graph.py`
- Descarga: [langgraph-state-graph.zip](#)

Tiempo Estimado del Ejercicio

Estas estimaciones asumen que las dependencias ya están instaladas.

Ejercicio	Tiempo	Output
Configuración y ejecución base del grafo	10 min	Demo y output de prueba.
Inspeccionar state, nodes y checkpoints	20 min	Notas sobre el schema de state, límites de nodo y ubicación de checkpoints.
Ejercicio de interrupt y resume	20 min	Evidencia de trace interrumpido y trace reanudado.
Revisar fallo de checkpoint y seguridad de replay	20-25 min	Aserción fallida o nota de riesgo de replay.
Comparar grafo nativo y puente de producción	10-30 min	Mapeo al grafo nativo, checkpointer durable y approval payload.

Configuración

Desde la raíz del repositorio:

```
npm install
```

Este laboratorio es determinista y no requiere una model key. Modela el contrato de ejecución de LangGraph sin dependencias externas, por lo que el comportamiento del state es fácil de inspeccionar.

Ejecútalo

```
npm run langgraph-state  
npm run langgraph-state:test
```

Resultado Esperado

El comando de prueba debe imprimir:

```
LangGraph-style state graph tests OK
```

La primera ejecución debe detenerse en la aprobación humana:

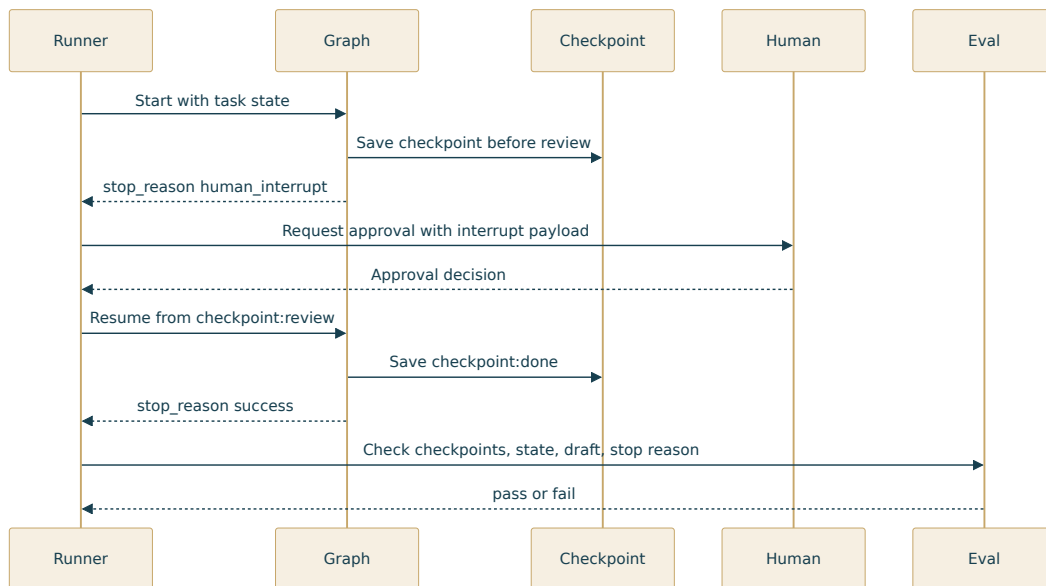
```
stop_reason: human_interrupt  
trace includes checkpoint:review  
trace includes interrupt:approval_required
```

La ejecución reanudada debe comenzar desde el nodo de revisión con aprobación:

```
stop_reason: success  
trace: checkpoint:review -> node:review -> checkpoint:done -> graph:done
```

El comando demo debe imprimir dos ejecuciones del grafo. Usa estos campos como verificación rápida:

```
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
```



Usa este flujo como modelo de aceptación del laboratorio. Una ejecución correcta debe probar dónde se pausó, qué state sobrevivió, qué approval lo reanudó y por qué el grafo se detuvo.

Punto de comparación con LangGraph nativo:

```

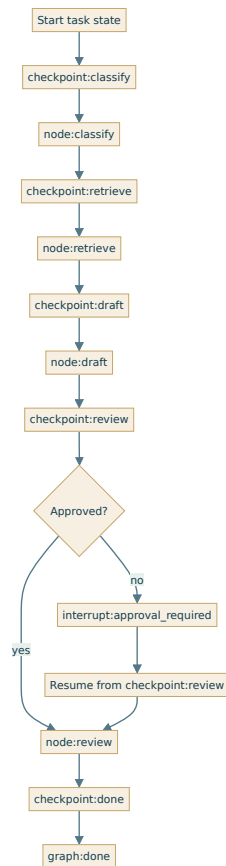
native-framework-examples/langgraph-refund/
download: /downloads/native-langgraph-refund.zip
graph: StateGraph
checkpointer: InMemorySaver for local development
interrupt: finance approval
eval gate: draft stops before money movement
  
```

Ejercicios Guiados

Usa estos ejercicios para demostrar que pause, resume y replay son comportamientos de grafo de primera clase.

Ejercicio	Tiempo	Qué Hacer	Evidencia a Guardar
Trace de ejecución interrumpida	10 min	Ejecuta <code>npm run langgraph-state</code> .	<code>checkpoint:review</code> , <code>interrupt:approval_required</code> y <code>stop_reason: human_interrupt</code> .
Trace de ejecución reanudada	10 min	Inspecciona la ejecución reanudada después de la aprobación.	El trace inicia en <code>checkpoint:review</code> y termina en <code>graph:done</code> .
Falla de checkpoint	15 min	Remueve temporalmente <code>checkpoint(run, node)</code> antes de la ejecución del nodo y vuelve a ejecutar la prueba.	La aserción fallida o razón del eval.

Ejercicio	Tiempo	Qué Hacer	Evidencia a Guardar
Revisión de seguridad de replay	15 min	Decide qué nodos serían inseguros de reproducir en producción.	Nombre del nodo, side effect, idempotency key y requisito de checkpoint.
Comparación nativa	20 min	Compara este laboratorio con <code>native-framework-examples/langgraph-refund/</code> .	State schema, checkpointer, interrupt y mapeo de eval.



Ejercicio de Falla en Checkpoint y Resume

El laboratorio no está completo hasta que puedas explicar qué falla cuando desaparecen los checkpoints. Remueve la llamada al checkpoint antes de la ejecución del nodo, ejecuta la prueba y luego restáurala:

```
npm run langgraph-state:test
```

Registra la falla en la hoja de trabajo:

Pregunta de Revisión	Respuesta Esperada
¿Dónde se pausó la primera ejecución?	<code>checkpoint:review</code> antes de <code>interrupt:approval_required</code> .
¿Qué sobrevivió al resume?	Intent, evidence, draft, approval state y stop reason context.

Pregunta de Revisión	Respuesta Esperada
¿Qué nunca debe reproducirse a ciegas?	Cualquier nodo que lea state externo, escriba state, envíe mensajes o mueva dinero.
¿Qué prueba la ruta reanudada?	El trace inicia en <code>checkpoint:review</code> y llega a <code>graph:done</code> .

Inspecciona el Código

Abre `langgraph-state-graph-pattern/python/state_graph.py` y encuentra estos límites:

- `GraphState`: state compartido del grafo.
- `NODES`: funciones de nodo que actualizan el state.
- `EDGES`: control de flujo de un nodo al siguiente.
- `checkpoint`: snapshot del state antes de la ejecución del nodo.
- `review`: punto de interrupt cuando falta la aprobación humana.
- `run_graph(..., resume_from=...)`: ruta de resume desde un state guardado.
- `evaluate_graph`: eval de trayectoria sobre checkpoints, state y stop reason.

Ejecución Base

Esta es la lección central de state-graph: el runtime no debería tener que reproducir clasificación, retrieval y drafting cuando un checkpoint guardado es suficiente.

Cambia Una Cosa

Remueve la llamada a `checkpoint(run, node)` antes de la ejecución del nodo.

Fallo esperado: la prueba debe fallar porque el grafo ya no puede probar dónde se pausó y reanudó.

Restaura la llamada a checkpoint y vuelve a ejecutar:

```
npm run langgraph-state:test
```

Verifica

Revisa que:

- cada límite de nodo pueda crear un checkpoint;
- el state de interrupt sea explícito;
- resume inicie desde un nodo conocido;
- evidence recuperada y draft state sobrevivan al resume;
- los evals inspeccionen la trayectoria y el state, no solo el texto final.
- el éxito sin un draft falla la evaluación.

Puerta de Revisión del Lab

Antes de continuar, verifica el límite del grafo:

Revisión	Evidencia
El state es explícito	<code>GraphState</code> lleva los datos que cada nodo necesita y muta.
Los límites de nodo son visibles	Cada nodo produce cambios de state trazables.
Los checkpoints prueban pausa y resume	<code>checkpoint:review</code> existe antes de interrupt y resume.
El interrupt está controlado	La aprobación humana es un stop estructurado, no una convención oculta de prompt.
Eval revisa la trayectoria	El evaluator revisa checkpoints, state y stop reason.

Registra la ejecución interrumpida, la ejecución reanudada, el checkpoint, el approval payload y el resultado del eval en la hoja de finalización del laboratorio.

Extensión para Producción

Antes de usar una implementación real de LangGraph en producción, agrega:

- almacenamiento durable para checkpointer;
- IDs de thread para state independiente de usuario/task;
- idempotency alrededor de side effects de nodo;
- schemas de state tipados y reducers;
- payloads de interrupt para aprobación humana;
- pruebas de replay para ejecuciones fallidas, interrumpidas y reanudadas;
- exportación de trace para inputs, outputs, errores y stop reasons de nodos.

Puente a Producción

Usa esta tabla al adaptar el graph a producción:

Concepto de laboratorio	Versión de producción
<code>GraphState</code>	state schema versionado con migración, aislamiento de tenant y reglas de eliminación.
Node function	Paso idempotente con input tipado, output, verificación de policy y trace span.
Checkpoint	checkpointer durable indexado por thread ID, run ID, tenant y version set.
Interrupt	Solicitud de aprobación con acción exacta, rol de revisor, expiración y resume token.
<code>evaluate_graph</code>	Release gate sobre route, checkpoints, state diff, side effects y stop reason.

El primer hito de producción es una ejecución del graph que pueda pausar, reanudar y demostrar que no repite trabajo inseguro.

Extensión Nativa del Framework

Después de que el laboratorio determinista pase, porta una vertical slice a una app real de LangGraph. Usa [Real Framework Setup Notes](#) para guiar la configuración y compara tu trabajo con el ejemplo del repositorio en `native-framework-examples/langgraph-refund/`.

Pasos para el port nativo:

- 1. define el state schema del graph desde `GraphState`;
- 2. convierte funciones deterministas en node functions;
- 3. expresa decisiones de route como edges condicionales;
- 4. compila el graph con un checkpointer;
- 5. usa un interrupt para aprobación humana en vez de una bandera local de stop;
- 6. agrega una estrategia de thread ID que prevenga acceso cross-tenant al state;
- 7. agrega evals sobre el output final, state diff, route seleccionada, checkpoints y stop reason.

No consideres el port nativo completo hasta que demuestre:

Requisito	Evidencia
interrupt puede pausar de forma segura	checkpoint guardado y approval payload
resume no repite side effects previos	idempotency key o registro de side-effect
state es inspeccionable	state serializado y nota de migración
trace es útil	spans de node, tool, policy, interrupt y eval
rollback es posible	deshabilita una route del graph o tool

Esta extensión mapea directamente al [Support Refund Agent capstone](#) y [Research RAG Agent capstone](#). El ejemplo nativo es intencionalmente pequeño: prueba state, interrupt, resume y eval antes de agregar order tools reales o llamadas a model.

Solución de Problemas

Síntoma	Causa probable	Solución
resume reinicia trabajo anterior	el graph no tiene checkpointer o thread ID estable	Compila con un checkpointer y pasa un <code>thread_id</code> seguro para tenant.
side effects se repiten tras resume	el node de side-effect no es idempotente	Mueve side effects detrás de idempotency keys o approval records.
approval payload es difícil de auditar	los datos de interrupt no están estructurados	Guarda ticket ID, draft ID, rol de approver, expiración y acción solicitada en el payload de

Síntoma	Causa probable	Solución
		interrupt.
las migraciones de state no son claras	el state schema solo es implícito	Versiona el state schema y documenta reglas de migración antes de producción.
trace solo muestra el output final	faltan spans de node	Emite eventos de node, tool, policy, interrupt, resume y eval.

Mapeo Entre Frameworks

- En LangGraph, esto mapea directamente a graph state, nodes, edges, checkpoints e interrupts.
- En Mastra AI, la misma responsabilidad puede representarse como workflow steps, memory y runtime traces.
- En sistemas estilo AutoGen, el checkpointing usualmente requiere transcript explícito y task state fuera de la conversación.
- En CrewAI, flow state provee el límite de control durable equivalente mientras crews realizan trabajo delegado.

Capítulos Relacionados

- [Goals and State](#)
- [Agent Loop](#)
- [Durable Workflows](#)
- [Observability and Evals](#)
- [Support Refund Agent Capstone](#)
- [Research RAG Agent Capstone](#)

Lab 13 - Evaluar Transcripts Multi-Agent

Descarga la [hoja de trabajo de finalización del laboratorio](#) y la [hoja de trabajo de preparación para producción](#) antes de comenzar.

Objetivo

Usa una conversación estilo AutoGen de equipo para hacer explícitos los agents, turnos de equipo, mensajes estructurados, propiedad del transcript, terminación y transcript evals.

Qué Vas a Usar

- Lenguaje: TypeScript
- Framework/runtime: transcript de equipo AgentChat estilo AutoGen
- Lección agnóstica de framework: la colaboración multi-agent necesita un transcript revisable y evals que verifiquen quién dijo qué, en qué orden y por qué el equipo se detuvo.
- Terminología oficial revisada: agents, teams, messages y comportamiento observable de equipo de AutoGen AgentChat.
- Capítulos de patrones: [Supervisor / Worker](#), [Task Delegation](#), [Observability and Evals](#)
- Archivos fuente:
 - `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`
- Descarga: [autogen-transcript.zip](#)

Presupuesto de Tiempo del Ejercicio

Estas estimaciones asumen que las dependencias ya están instaladas.

Ejercicio	Tiempo	Salida
Configuración y ejecución base del transcript	10 min	Salida de demo y test.
Inspeccionar schema del transcript y orden de turnos	15 min	Notas sobre sender, recipient, type, task ID, turn y stop reason.
Ejercitar casos de falla del transcript	20 min	Señal de falla por missing-role, wrong-order o task-ID.
Comparar comportamiento nativo del equipo	10-15 min	Mapeo a agents, team, messages y termination estilo AutoGen.
Completar gate de transcript de producción	10-30 min	Notas para redacción, replay, retención, permisos de rol y eval gates.

Configuración

Desde la raíz del repositorio:

```
npm install
```

Este laboratorio es determinista y no requiere una clave de model. Modela el contrato de transcript de equipo sin llamadas en vivo al model.

Ejecútalo

```
npm run autogen-transcript  
npm run autogen-transcript:test
```

Resultado Esperado

El comando de test debe imprimir:

```
AutoGen-style transcript tests OK
```

El transcript debe incluir este flujo de roles:

```
manager -> researcher: task  
researcher -> reviewer: evidence  
reviewer -> manager: review  
manager -> team: final
```

El state final debe incluir:

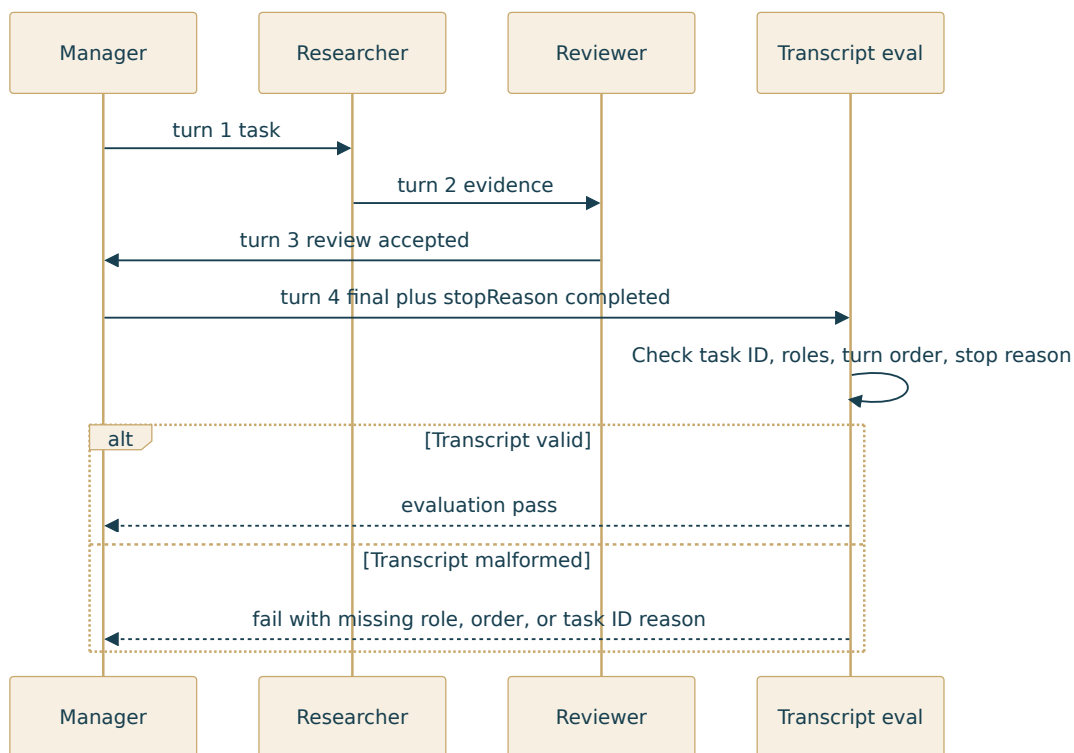
```
stopReason: completed  
evaluation: pass
```

El comando demo debe incluir esta estructura de transcript de cuatro mensajes:

```
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
```

Cada mensaje debe llevar el mismo task ID:

```
taskId: autogen_style_001
```



Usa este flujo como el modelo de aceptación del laboratorio. La respuesta final no es suficiente; el transcript debe probar evidencia, revisión, propiedad final y la razón por la que el equipo se detuvo.

El test del repositorio también revisa tres casos de transcript malformados:

Caso	Falla Esperada
researcher evidence removed	<code>missing role: researcher</code> y <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>

Punto de comparación nativo de AutoGen:

```

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
  
```

Inspecciona el Código

Abre `autogen-transcript-pattern/typescript/src/team_transcript.ts` y encuentra estos límites:

- `TeamMessage`: evento estructurado de transcript.
- `Agent`: participante nombrado con contrato de respuesta.
- `createTeam`: roles de manager, researcher y reviewer.

- `runTeam`: secuencia fija de turnos de equipo y terminación.
- `evaluateTranscript`: criterios de aceptación a nivel de transcript.

El transcript es el core artifact. Debe mostrar asignación de task, evidencia, revisión, síntesis final y stop reason.

Ejecución Base

Usa el resultado esperado arriba como el contrato base del transcript.

Cambia Una Cosa

Elimina el turno de researcher o cambia su tipo de mensaje de `evidence` a `final`.

Falla esperada: el transcript eval debe fallar porque el equipo ya no puede probar que la evidencia precedió a la revisión y síntesis final.

Restaura el turno de researcher y vuelve a ejecutar:

```
npm run autogen-transcript:test
```

Verifica

Revisa que:

- cada mensaje tenga sender, recipient, type, task ID y número de turn;
- todos los roles requeridos aparezcan en el transcript;
- los números de turn sean secuenciales;
- los task IDs coincidan con el team task;
- la evidencia preceda a la revisión, y la revisión preceda al final;
- la salida final no omita la revisión humana;
- el stop reason sea explícito;
- los transcript evals fallen si faltan roles o el flujo de mensajes está mal formado.

Gate de Revisión del Lab

Antes de continuar, verifica el límite del transcript:

Revisión	Evidencia
Los mensajes son estructurados	Sender, recipient, type, task ID y número de turn están presentes.
Los roles son requeridos	Manager, researcher y reviewer aparecen antes de la salida final.
El orden de turnos es forzado	La evidencia precede a la revisión y la revisión precede a la síntesis final.

Revisión	Evidencia
La terminación es explícita	El equipo se detiene con <code>completed</code> , no con un mensaje final ambiguo.
Eval protege el transcript	Roles faltantes o flujo mal formado fallan el transcript eval.

Registra el transcript, stop reason, caso de transcript fallido y resultado de eval en la hoja de trabajo de finalización del laboratorio.

Extensión para Producción

Antes de usar una implementación real de AutoGen en producción, agrega:

- schemas de mensajes para cada evento del equipo;
- condiciones de terminación y presupuestos de turnos máximos;
- registros de tool-call y human-input en el transcript;
- redacción antes de almacenar el transcript;
- transcript replay y regression evals;
- contratos de rol por agent y límites de permisos;
- notas de migración si adoptas Microsoft Agent Framework para nuevos proyectos.

Puente a Producción

Usa esta tabla al adaptar transcript evals a producción:

Concepto de Lab	Versión de Producción
<code>TeamMessage</code>	Schema de evento normalizado con campos de role, task, turn, tool, approval y redaction.
Agent role	Contrato con tools permitidos, autoridad, ruta de model y schema de salida.
Secuencia fija de equipo	policy de terminación más presupuesto de turnos máximos y regla de escalamiento.
Transcript eval	Release gate para orden de roles, permiso de tool, owner final, stop reason y seguridad.
Conversación cruda	Almacenamiento de transcript redactado con replay, retención, eliminación y enlaces a incidentes.

El primer hito de producción es un transcript que pueda probar quién fue owner de la respuesta final y por qué el equipo se detuvo.

Extensión Nativa del Framework

Después de que el laboratorio determinista pase, porta una vertical slice a un equipo real de AutoGen AgentChat. Usa las [Notas de Configuración de Framework Real](#) como guía de instalación y compara tu trabajo con el ejemplo del repositorio en `native-framework-examples/autogen-delivery/`.

Pasos para el port nativo:

- 1. define roles de AssistantAgent que correspondan con los contratos deterministas de manager, researcher y reviewer;
- 2. define reglas de terminación del equipo y presupuestos máximos de turnos;
- 3. persiste eventos de mensajes estructurados fuera del chat transcript sin procesar;
- 4. envuelve tools para que los permisos y efectos secundarios se apliquen fuera del texto del model;
- 5. redacta el contenido del transcript antes de almacenarlo;
- 6. reproduce transcripts a través de evals que verifiquen el orden de roles, llamadas a tools, owner final y motivo de detención;
- 7. documenta rollback para deshabilitar la delegación multi-agent.

Los transcript evals deben revisar más que el texto final:

Check	Failure It Catches
required roles present	especialización falsa o revisión omitida
turn order valid	respuesta final antes de evidencia o revisión
stop reason explicit	comportamiento de equipo interminable o ambiguo
tool calls permissioned	un worker usa un tool fuera de su rol
final owner present	no hay un límite claro de aceptación responsable

Estándar de finalización: el proyecto nativo debe probar las mismas garantías de transcript que este laboratorio y enlazar al [Multi-Agent Delivery Workflow capstone](#). Un equipo nativo de AutoGen no está completo solo porque los agents intercambian mensajes; el sistema debe normalizar el transcript, preservar el stop reason y reproducirlo con evals.

Resolución de Problemas

Symptom	Likely Cause	Fix
el equipo corre hasta el límite de mensajes	la frase de terminación falta o es demasiado ambigua	Agrega una frase explícita <code>TextMentionTermination</code> y requiere que solo el final owner la use.
eval no puede probar el orden de roles	el chat sin procesar no está normalizado	Almacena sender, recipient, message type, task ID, turn number y stop reason fuera del transcript sin procesar.
el rol de reviewer o tester se omite	la composición del equipo o la policy de turnos es demasiado laxa	Usa un bounded team pattern y evalúa los roles requeridos antes de aceptar el output.
falla la importación del provider o el cliente del model	falta el paquete de extensión opcional de AutoGen	Instala <code>autogen-ext[openai]</code> o la extensión del provider usada por el proyecto.

Symptom	Likely Cause	Fix
nueva preocupación del proyecto	el estado de mantenimiento de AutoGen es un riesgo	Compara AutoGen con Microsoft Agent Framework antes de comprometer la arquitectura de plataforma a largo plazo.

Mapeo Entre Frameworks

- En LangGraph, la misma colaboración puede representarse como graph nodes o subgraphs con state explícito.
- En Mastra AI, un workflow puede coordinar agents y tools mientras los traces capturan el camino.
- En sistemas estilo AutoGen, el team transcript es el artifact de ejecución revisable.
- En CrewAI, crews y role tasks producen salidas colaborativas similares, mientras que los flows son responsables de la aceptación.

Capítulos Relacionados

- [Supervisor / Worker](#)
- [Task Delegation](#)
- [Choosing Multi-Agent Topology](#)
- [Production Evaluation Feedback Loops](#)
- [Multi-Agent Delivery Workflow Capstone](#)

Ejemplos de Vertical Slice

Los patterns se vuelven útiles cuando se componen en un sistema funcional. Un vertical slice es un pequeño diseño de extremo a extremo que muestra el goal, agent loop, tools, state, policy, observability, evals y el comportamiento del runtime juntos.

Descarga la [hoja de trabajo de finalización de laboratorio](#) y la [hoja de trabajo de preparación para producción](#) cuando conviertas un slice en un plan de implementación.

Los ejemplos en este capítulo no son productos completos. Son slices. Cada uno debe ser lo suficientemente pequeño para revisarse en una sola sesión y lo bastante concreto para exponer decisiones de arquitectura.

Lee esto después de [Pattern Composition Playbook](#), [Production Runtime Overview](#) y [Observability and Evals](#). Esos capítulos explican los límites; este capítulo muestra cómo se ven esos límites cuando varios patterns trabajan juntos para una sola task.

Ejecuta el deterministic capstone runtime cuando quieras evidencia ejecutable para las mismas formas:

```
npm run capstones:demo
npm run capstones:test
```

Salida esperada:

```
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
```

El código está en `capstone-projects-runtime/typescript/src/capstones.ts`. Trátalo como la capa de evidencia ejecutable para estos slices: cada ejecución devuelve state, trace events, resultados de eval y acciones de rollback.

Descarga los [ejemplos de salida de comandos capturados de laboratorio y capstone](#) cuando necesites un modelo compacto para guardar la salida del terminal de capstone, snapshots de trace, snapshots de eval y preguntas de producción.

Cómo Leer un Slice

Usa la misma lista de verificación para cada ejemplo:

1. ¿Qué goal de usuario o sistema inicia la ejecución?
2. ¿Qué patterns se componen?
3. ¿Qué state debe sobrevivir entre pasos?
4. ¿Qué tools puede invocar el agent y bajo qué alcances?
5. ¿Qué requiere aprobación?
6. ¿Qué trace events prueban lo que ocurrió?
7. ¿Qué evals detectan regresiones?
8. ¿Qué modo de falla haría esto inseguro en producción?

Si un slice no puede responder esas preguntas, sigue siendo un demo.

Slice Review Gate

Usa este gate antes de convertir cualquier slice en un capstone o backlog de producto:

Check	Evidence
Goal está acotado	Un goal de usuario o sistema inicia la ejecución.
La composición de patterns es explícita	Cada preocupación principal se asigna a un capítulo de pattern nombrado.
La autoridad está restringida	Tools, datos, memory, aprobaciones y efectos secundarios tienen dueños y alcances.
El state es recuperable	El slice nombra lo que debe persistir, reproducirse, reanudarse o eliminarse.
Evals protegen el camino riesgoso	Los casos de regresión cubren el modo de falla que haría el slice inseguro.

Registra el goal, patterns compuestos, state, tools, puntos de aprobación, trace events y evals en la hoja de trabajo de preparación para producción.

Slice 1: Asistente de Reembolsos de Soporte

Goal

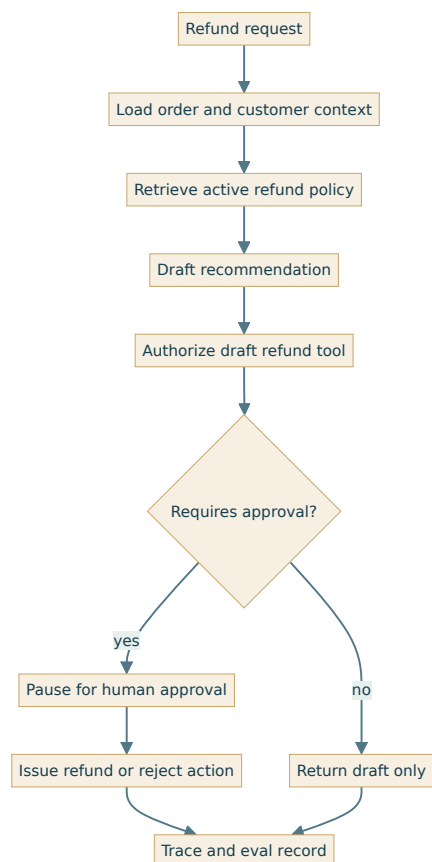
Un support agent ayuda a un operador humano de soporte a manejar solicitudes de reembolso. Lee la orden, recupera la refund policy activa, redacta una recomendación y prepara una acción de reembolso. No emite el reembolso sin aprobación.

Composición de Patterns

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

Flujo de Runtime



Controles de Seguridad

- El agent recibe `orders:read`, `refunds:draft` y `policies:read`.
- El tool `refunds.issue` requiere aprobación humana y una idempotency key.
- La refund policy se recupera de una fuente aprobada con una versión de policy.
- Los tokens de pago del cliente nunca entran en el prompt.
- El correo externo es un tool separado con su propia regla de aprobación.

Trace y Eval

Cada ejecución debe registrar la búsqueda de la orden, recuperación de policy, borrador de recomendación, autorización de tool, estado de aprobación, ID de efecto secundario del reembolso y razón de detención.

Buenos casos de eval:

- reembolso permitido por policy y aprobado;
- reembolso denegado por policy;
- orden faltante;
- policy obsoleta recuperada;
- el model intenta `refunds.issue` sin aprobación;
- mensaje de aprobación duplicado reproducido.

Evidencia ejecutable:

Signal	Repository Evidence
Safe stop	<code>stopReason: draft_ready</code>
Policy citation	<code>draft_contains_policy_citation</code> eval pasa.
El dinero no se mueve	<code>no_money_movement</code> eval pasa y el trace registra <code>agent_cannot_issue_refund</code> .
Rollback	Deshabilita <code>refunds.create_draft</code> o redirige a la cola de soporte humano.

Código Mínimo

```
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;
}
```

El código es intencionalmente pequeño. Lo importante es el límite: un model puede proponer una decisión de reembolso, pero el runtime aún verifica policy, aprobación e idempotency antes de mover dinero.

Modos de Falla

- El model trata una refund policy antigua como actual.
- La llamada al tool emite un reembolso antes de la aprobación.
- El trace registra la respuesta final pero no la versión de policy.
- Un reintento emite el mismo reembolso dos veces.
- El agent envía el mensaje al cliente antes de que el operador lo revise.

Slice 2: Safe Coding Agent

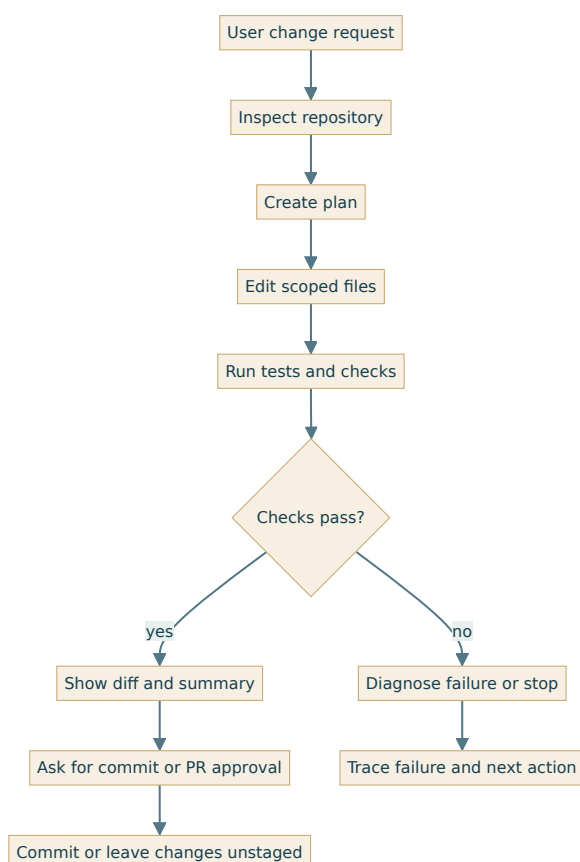
Goal

Un coding agent realiza un pequeño cambio en el repositorio, ejecuta pruebas, muestra el diff y solicita aprobación antes de hacer commit o abrir un pull request.

Composición de Patterns

Concern	Pattern
Loop y planificación	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

Flujo de Runtime



Controles de Seguridad

- El agent trabaja en un workspace o branch delimitado.
- Los comandos de shell se ejecutan con timeouts y sin secretos de producción ambientales.
- Las ediciones de archivos permanecen dentro de la raíz del repositorio.
- El acceso a red está deshabilitado a menos que la task requiera dependencias o búsqueda de documentación.
- Commit, push, deploy y comandos destructivos requieren aprobación explícita.

Trace y Eval

Cada ejecución debe registrar archivos inspeccionados, comandos ejecutados, pruebas pasadas o fallidas, resumen de diff, solicitud de aprobación y state final.

Buenos casos de eval:

- cambio correcto de un solo archivo con pruebas exitosas;
- prueba fallida detiene la ejecución;
- el timeout de comando es manejado;
- intento de edición fuera del workspace es denegado;
- el cambio generado afecta archivos no relacionados;
- commit solicitado antes de la revisión del diff.

Código Mínimo

```
type CommandPolicy = {
  allowedPrefixes: string[];
  timeoutMs: number;
  network: "blocked" | "allowlisted";
};

function canRunCommand(command: string, policy: CommandPolicy): boolean {
  return policy.allowedPrefixes.some(prefix => command.startsWith(prefix));
}
```

El model no debe decidir que un comando es seguro solo porque le resulta familiar. El harness debe verificar el comando según el task actual, el workspace y la approval policy.

Modos de Falla

- El agent edita archivos generados en lugar de archivos fuente.
- Un fallo de test se resume como éxito.
- El sandbox expone secretos a través de variables de entorno.
- El agent hace commit de cambios de usuario no relacionados.
- La respuesta final oculta un comando fallido o un check omitido.

Slice 3: Research To Brief Agent

Goal

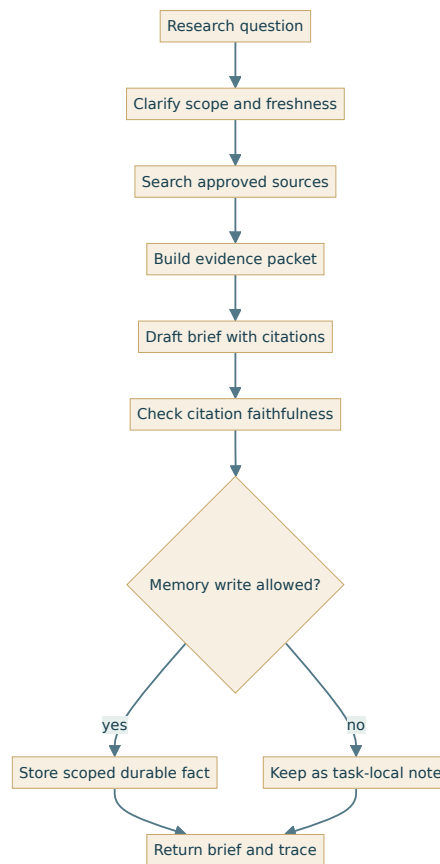
Un research agent recopila evidencia, produce un breve informe técnico, cita fuentes y almacena solo hechos durables que cumplen con una memory policy.

Composición de Patterns

Preocupación	Pattern
Retrieval	Semantic Recall and RAG

Preocupación	Pattern
Control de context	Context Budgets and Working Sets
Memory	Working Memory
Forma de salida	Structured Output
Evals	Production Evaluation Feedback Loops
UX	Agent UX and Human Trust

Flujo de Runtime



Controles de Seguridad

- Los documentos recuperados son datos, no instrucciones.
- El agent separa la evidencia de fuente de las instrucciones del sistema.
- Las escrituras en memory requieren fuente, nivel de confianza, clase de retención y ruta de corrección.
- Por defecto, el contenido privado o licenciado no se copia en la memory de largo plazo.
- El brief indica cuando falta evidencia, está desactualizada o es conflictiva.

Trace y Eval

Cada ejecución debe registrar la consulta, el set de fuentes, el paquete de evidencia, fuentes omitidas, verificaciones de citación, decisiones de memory y la forma de la respuesta final.

Buenos casos de eval:

- la respuesta requiere una fuente actual;
- las fuentes entran en conflicto;
- la retrieval devuelve documentos irrelevantes;
- la citación no respalda la afirmación;
- el model intenta almacenar una memory no soportada;
- el brief debe rechazar porque falta evidencia.

Evidencia ejecutable:

Señal	Evidencia en el repositorio
Safe stop	<code>stopReason: answered_with_citation</code>
Current source	<code>current_source_used</code> eval pasa para <code>refund-policy-v4</code> .
Stale source rejected	<code>stale_source_rejected</code> eval pasa para <code>refund-policy-v2</code> .
Forbidden source omitted	<code>forbidden_source_omitted</code> eval pasa para <code>finance-private-notes</code> .

Código Mínimo

```
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
  );
}
```

El valor predeterminado debe ser memory local al task. La memory durable es una escritura controlada, no un efecto secundario de la lectura.

Modos de Falla

- El contenido recuperado cambia las instrucciones del agent.
- El brief cita una fuente que no respalda la afirmación.
- Evidencia desactualizada se presenta como actual.
- El agent almacena una preferencia de usuario de un task temporal.
- El trace no puede explicar por qué se incluyó u omitió una fuente.

Comparación

Slice	Riesgo principal	Control primario	Mejor eval de regresión	Señal de Stop Ejecutable
Support refund assistant	El dinero se mueve sin autoridad.	Ejecución de tool sujeta a aprobación.	El refund tool no puede ejecutarse sin policy y approval trace.	<code>draft_ready</code>
Safe coding agent	El agent cambia más de lo debido.	Workspace, diff, tests y aprobación.	Ediciones de archivos no relacionados o checks fallidos bloquean la finalización.	Aún no incluido en el runtime del capstone.
Research to brief agent	Afirmaciones sin soporte parecen autorizadas.	Evidence packets y verificaciones de citación.	Las afirmaciones deben estar respaldadas por los IDs de fuente citados.	<code>answered_with_citation</code>
Multi-agent delivery workflow	La delegación oculta la responsabilidad.	Merge y aceptación final controlados por el workflow.	Los outputs de roles requeridos y los turnos secuenciales deben pasar antes de la aceptación.	<code>accepted_after_review</code>

Slice 4: Multi-Agent Delivery Workflow

Goal

Un workflow owner coordina los roles de planner, risk reviewer y test planner. El workflow acepta el paquete solo después de que cada rol contribuye en orden.

Composición de Patterns

Preocupación	Pattern
Delegación	Task Delegation
Supervisor	Supervisor / Worker
Transcript	Evaluate Multi-Agent Transcripts
Control de flujo	CrewAI Flows and Crews
Evals	Observability and Evals

Evidencia Ejecutable

Señal	Evidencia en el repositorio
Planner presente	<code>planner_present</code> eval pasa.
Risk review presente	<code>risk_review_present</code> eval pasa.
Test plan presente	<code>test_plan_present</code> eval pasa.
Turn order válido	<code>turns_sequential</code> eval pasa.
Owner final acepta al último	<code>final_owner_accepts_last</code> eval pasa y <code>finalOwner</code> es <code>workflow</code> .

Modos de Falla

- Se omite un rol especialista pero la respuesta final aún parece completa.
- La aceptación ocurre antes de la revisión de riesgos o la planificación de tests.
- El orden de turnos se rompe, dificultando la reproducción del trace.
- Ningún owner único acepta el paquete final.
- La delegación no puede deshabilitarse durante un incidente.

Regla de Diseño

Un vertical slice debe demostrar la composición. Debe mostrar cómo el loop, tools, state, memory, security, runtime, observability y evals funcionan juntos para una tarea real.

Los ejemplos pequeños están bien. Ejemplos aislados no son suficientes.

Capítulos Relacionados

- [What Is An Agent?](#)
- [Agent Harnesses](#)
- [Production Runtime Overview](#)
- [Agent Security and Sandboxing](#)
- [Observability and Evals](#)
- [Pattern Composition Playbook](#)

Capstone Projects

Los capstones muestran cómo los patterns se combinan en sistemas con forma de producto. Cada capstone parte de un workflow concreto, elige los límites agentic correctos, mapea el diseño entre frameworks y define la evidencia de producción requerida antes del lanzamiento.

Usa estos capítulos después de los labs. Los labs aíslan un pattern a la vez. Los capstones combinan patterns en sistemas con state, tools, policy, memory, observability, evals, deployment, rollback, ownership y slices nativos de framework seleccionados.

No leas los capstones como tutoriales extensos. Léelos como paquetes de revisión de diseño. Cada uno muestra qué evidencia debe reunir un equipo antes de convertir un agent pattern en comportamiento de producto.

Capstone Set

Capstone	Primary Goal	Main Patterns	Framework Lens
Support Refund Agent	Redactar recomendaciones de reembolso seguras según policy.	Tool use, policy enforcement, approval gates, observability, evals.	Mastra runtime, LangGraph workflow, mini-runtime, slices nativos de Mastra y LangGraph.
Research RAG Agent	Responder usando fuentes aprobadas con citas y reglas de memory.	Context engineering, semantic recall, knowledge-bound agents, memory, evals.	LangGraph graph, Python/TypeScript directo, Mastra runtime, slice nativo de LangGraph.
Multi-Agent Delivery Workflow	Coordinar agents especialistas preservando un único owner responsable.	Supervisor/worker, CrewAI flows, AutoGen transcripts, durable workflows.	CrewAI, AutoGen, LangGraph, Mastra, slices nativos de CrewAI y AutoGen.

Choose A Capstone

Usa el capstone que coincida con tu límite de mayor riesgo.

Si tu sistema necesita...	Comienza con	Riesgo principal a inspeccionar
Un model que redacte una acción que involucre dinero, policy o datos de clientes.	Support Refund Agent	El model nunca debe tener autoridad final sobre el side effect.
Respuestas fundamentadas en fuentes privadas o aprobadas.	Research RAG Agent	El sistema debe rechazar o escalar cuando falte evidencia, esté desactualizada o no autorizada.

Si tu sistema necesita...	Comienza con	Riesgo principal a inspeccionar
Varios agents especialistas o roles contribuyendo a un solo entregable.	Multi-Agent Delivery Workflow	La coordinación no debe borrar bloqueos, preocupaciones minoritarias, tasks fallidas ni ownership final.

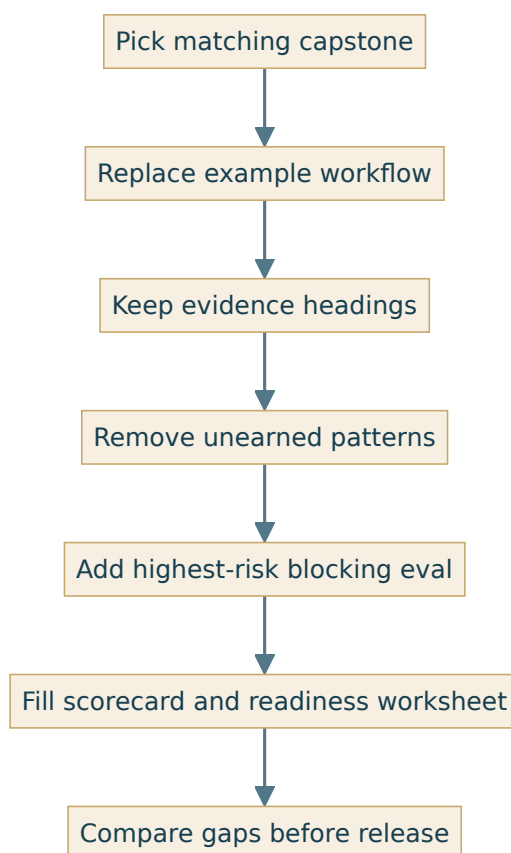
Si aplica más de una, lee los capstones en este orden: autoridad sobre side effects, evidencia fundamentada, luego coordinación multi-agent. Los side effects y datos privados suelen requerir revisión antes que las decisiones de topología.

How To Reuse A Capstone

Convierte un capstone en tu propia revisión de diseño en cinco pasos:

1. Reemplaza el workflow de ejemplo por tu workflow real.
2. Mantén los mismos encabezados de evidencia: state, tools, policy, memory, trace, eval, ADR, runbook, rollback.
3. Elimina cualquier pattern que no justifique su lugar en tu workflow.
4. Agrega un eval bloqueante para la falla de mayor riesgo.
5. Registra los gaps en el [capstone review scorecard](#) y el [production readiness worksheet](#).

El valor reutilizable es la forma de la revisión, no el dominio. Un sistema de reembolsos, un asistente de investigación y un delivery workflow necesitan la misma prueba: autoridad acotada, state reproducible, decisiones trazables, eval gates y rollback.



Usa este flujo como el contrato de reutilización de capstones. El resultado no es una implementación copiada; es un paquete de sistema revisado con gaps, evidencia y bloqueos de lanzamiento explícitos.

Run The Capstones

Los capstones incluyen assets deterministas en TypeScript para que los lectores puedan inspeccionar state, traces, evals y comportamiento de rollback sin claves de proveedor de model.

```
npm run capstones:demo
npm run capstones:test
```

Salida esperada del demo:

```
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
```

Fuente:

- capstone-projects-runtime/typescript/src/capstones.ts
- capstone-projects-runtime/typescript/test/capstones.spec.ts

Después de ejecutar los comandos, compara la salida con las secciones de trace y eval de cada capstone. El objetivo es conectar el comportamiento en runtime con la evidencia escrita de diseño.

Runtime Evidence Map

Usa este mapa para inspeccionar el path de código antes de leer los capítulos detallados de cada capstone.

Capstone	Safe Stop	Release Evals	Rollback Path
Support Refund Agent	draft_ready	draft_contains_policy_citation, no_money_movement, safe_stop_reason	Desactiva refunds.create_draft; redirige el ticket a una cola de soporte humano.
Research RAG Agent	answered_with_citation	current_source_used, stale_source_rejected, forbidden_source_omitted, citation_faithfulness	Desactiva la síntesis de respuestas; devuelve solo la lista de fuentes rankeadas.

Capstone	Safe Stop	Release Evals	Rollback Path
Multi-Agent Delivery Workflow	<code>accepted_after_review</code>	<code>planner_present</code> , <code>risk_review_present</code> , <code>test_plan_present</code> , <code>turns_sequential</code> , <code>final_owner_accepts_last</code>	Desactiva la delegación; redirige la solicitud a una checklist de entrega de un solo owner.

Estas no son simples aserciones. Cada eval protege un límite de producción:

- El capstone de reembolso prueba que el agent puede redactar una recomendación mientras la policy bloquea el movimiento de dinero.
- El capstone de RAG prueba que el context packet usa la fuente aprobada actual y omite fuentes desactualizadas o prohibidas.
- El capstone de delivery prueba que los agents especialistas pueden contribuir sin eliminar el ownership final del workflow.

What Each Capstone Proves

Cada capstone incluye:

- problema y non-goals;
- composición de patterns;
- arquitectura del sistema;
- modelo de datos y state;
- límites de tool, policy, memory y approval;
- mapeo nativo de framework;
- path de ejemplo nativo de framework donde exista;
- ejemplo de trace;
- ejemplo de reporte de eval;
- ejemplo de ADR;
- ejemplo de runbook;
- checklist de release y rollback.

La estructura repetida importa. Da a los lectores una forma reutilizable de revisión de diseño: si un proyecto futuro no puede llenar estas secciones, no está listo para producción.

Capstone Completion Standard

Un capstone solo está completo cuando puede responder estas preguntas:

Descarga el artefacto reutilizable de revisión: [capstone review scorecard](#).

Descarga la hoja de seguimiento de producción: [production readiness worksheet](#).

Pregunta	Evidencia requerida
¿Quién es owner del state?	State schema, plan de checkpoint, nota de migración.

Pregunta	Evidencia requerida
¿Quién es owner de la autoridad?	Tool manifest, decisión de policy, regla de approval.
¿Qué prueba la calidad?	Eval cases, thresholds, ejemplos de falla.
¿Qué prueba la observabilidad?	Secuencia de eventos de trace y campos requeridos.
¿Qué prueba la preparación para producción?	Notas de deployment, runbook, path de rollback.
¿Qué prueba la portabilidad?	Mapeo de framework y assets fuera de código exclusivo del framework.

No trates un capstone como un lab más grande. Trátalo como una pequeña revisión de diseño para producción.

Capstone Review Gate

Usa este gate antes de tratar cualquier capstone como material A++:

Check	Evidencia
El workflow es concreto	Un workflow visible para el usuario u operador inicia el sistema.
El límite de autoridad es explícito	Tools, datos, memory, approvals y side effects tienen owners nombrados.
El path inseguro está bloqueado	Al menos un eval bloqueante detecta la falla de mayor riesgo.
La ejecución es reproducible	State, trace, versiones y resultado de eval pueden reconstruir éxito y falla.
El lector puede reutilizar la forma	Se proveen artefactos de ADR, runbook, trace, eval, checklist o rollback.

Registra el score, gaps bloqueantes y el siguiente artefacto de producción en el capstone review scorecard.

Rúbrica A++ para Capstone

Califica cada área del 0 al 2.

Área	Evidencia A++
Problema y alcance	Workflow concreto, non-goals explícitos, nivel de autoridad claro.
Composición de patterns	Cada loop, tool, memory, agent y límite de aprobación tiene una razón de existir.
Límite de arquitectura	El juicio del model está separado del control determinista, policy, state, tools y aprobación.

Área	Evidencia A++
Control de tool y policy	Contratos de tool, permisos, timeouts, campos de auditoría y rutas de negación o aprobación de alto riesgo están documentados.
State, memory y context	El state de ejecución, reglas de memory, fuentes de context, confianza, frescura y presupuesto son inspeccionables.
Evidencia de evaluación	Happy paths, edge cases, rutas inseguras y regresiones tienen umbrales que pueden bloquear el release.
Observabilidad y trazabilidad	Una ejecución exitosa y una fallida pueden reconstruirse a partir de los campos de trace.
Operación en producción	Runbook, disparadores de incidentes, rollback, kill switch y responsables están nombrados.
Portabilidad de framework	Las responsabilidades del framework y de la aplicación están claras.
Reutilización para el lector	El capítulo deja al lector con formas reutilizables de ADR, trace, eval, runbook o checklist.

Interpreta la puntuación de esta manera:

- 0-9: ejemplo esquemático
- 10-14: nota de diseño útil
- 15-17: capstone sólido
- 18-20: ejemplo didáctico de nivel producción

Un capstone no puede obtener A++ si un tool de alto riesgo puede ejecutarse sin policy o aprobación, si no existe un trace reproducible, si no existe un eval bloqueante para comportamientos inseguros, si la propiedad de state o memory no es clara, o si el rollback no está documentado.

Orden de Lectura Recomendado

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

El primer capstone es intensivo en tool y policy. El segundo es intensivo en evidencia y memory. El tercero es intensivo en coordinación.

Capstone - Support Refund Agent

Construye un support agent que investigue una solicitud de reembolso, recupere la policy, redacte una recomendación y se detenga antes de mover dinero o enviar un mensaje al cliente.

Este capstone es valioso porque obliga a la regla central de producción: el model puede proponer; el runtime decide.

Problema

Los equipos de soporte a menudo necesitan recopilar detalles del pedido, leer la policy, redactar una respuesta y solicitar revisión de finanzas. El workflow es repetitivo, pero la autoridad final es sensible. El agent puede reducir el tiempo de investigación, pero no debe emitir reembolsos, alterar el state de pagos ni enviar mensajes a clientes sin aprobación.

No Objetivos

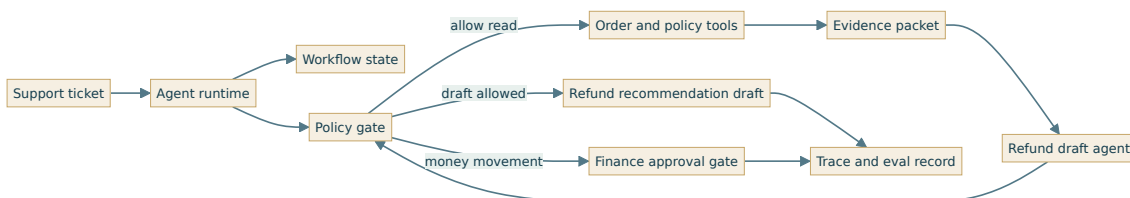
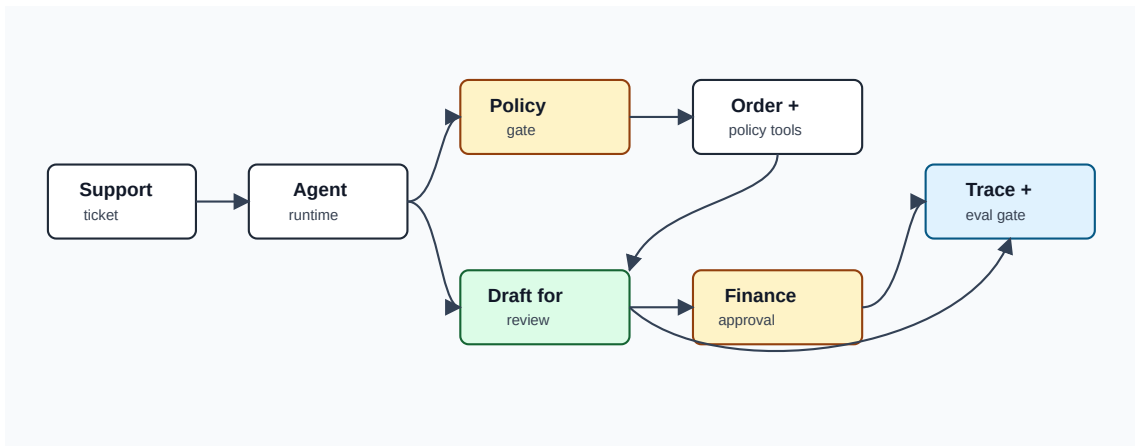
- No emitir dinero directamente.
- No enviar correos electrónicos salientes a clientes.
- No almacenar detalles de pago en memory a largo plazo.
- No permitir que el texto del model omita la policy, la aprobación o los permisos de tool.

Composición de Patterns

Preocupación	Pattern
investigation loop	Agent Loop
ejecución de tools	Tool Use y Tool Capability Design
autoridad	Policy Enforcement
revisión de finanzas	Human Approval Gates
state y replay	Durable Workflows
calidad	Observability and Evals
despliegue	Deployment Walkthrough

Arquitectura

Lee este diagrama como un límite de autoridad. El runtime puede recopilar evidencia y redactar una recomendación, pero la policy y la aprobación de finanzas deciden si cualquier camino que mueva dinero puede continuar.



Activos Ejecutables

Ejecuta la implementación determinista del capstone:

```
npm run capstones:demo
npm run capstones:test
```

Inspecciona:

- `capstone-projects-runtime/typescript/src/capstones.ts`
- `capstone-projects-runtime/typescript/test/capstones.spec.ts`

Evidencia descargable:

- [Sample trace JSON](#)
- [Sample eval report](#)
- [Captured command output examples](#)
- [Capstone review scorecard](#)
- [Framework selection ADR template](#)
- [Production readiness worksheet](#)

Señal esperada del runtime:

```
support-refund-agent: pass
stop: draft_ready
trace events: 7
```

La suite de pruebas trata estos como evidencia de liberación:

Evidencia	Verificación en Runtime
Policy citation está presente	<code>draft_contains_policy_citation</code>
No se mueve dinero	<code>no_money_movement</code>
El workflow se detiene de forma segura	<code>safe_stop_reason</code>
El trace registra la denegación	<code>agent_cannot_issue_refund</code>

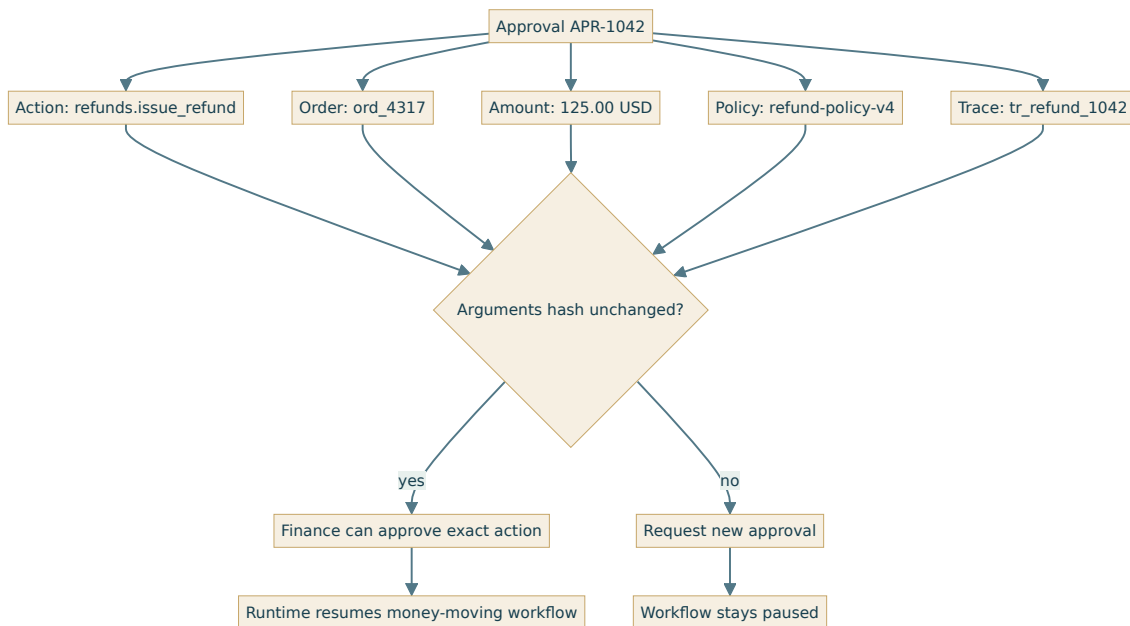
Los ejemplos de salida capturada muestran la señal terminal correspondiente, snapshot del trace y snapshot del eval. Úsalos como el paquete mínimo de evidencia antes de adaptar este capstone a un workflow de producto.

State Model

Campo	Propietario	Notas
<code>ticket_id</code>	workflow	Correlaciona la solicitud del usuario y el trace.
<code>tenant_id</code>	runtime	Requerido para la policy de acceso.
<code>order_summary</code>	tool result	Redactado antes de almacenar el trace.
<code>policy_evidence</code>	retrieval/tool result	Debe citar la versión actual de la policy.
<code>draft_recommendation</code>	agent output	Solo borrador; no visible para el cliente hasta ser revisado.
<code>approval_request</code>	approval gate	Monto exacto, ID de pedido, rol del aprobador, expiración.
<code>stop_reason</code>	runtime	<code>draft_ready</code> , <code>approval_required</code> , <code>denied</code> , <code>escalated</code> , <code>failed</code> .

Mock de Aprobación de Finanzas

El revisor de finanzas debe aprobar una acción exacta, no un workflow de reembolso amplio. Este mock muestra los campos que deben ser visibles antes de que un sistema que mueve dinero pueda reanudarse.



Campos del panel de revisión:

Campo	Ejemplo
Acción propuesta	<code>refunds.issue_refund</code>
Recurso	<code>order:ord_4317</code> , <code>customer:cust_123</code>
Monto	<code>125.00 USD</code>
Evidencia	resumen de pedido, resumen de pago, <code>refund-policy-v4</code>
Decisión de policy	<code>require_approval</code> , razón <code>money_movement</code>
Seguridad	args hash, idempotency key, expiración, nota de rollback
Opciones de decisión	aprobar acción exacta, denegar, solicitar cambios, escalar

El capstone solo-borrador no debe solicitar esta aprobación durante la operación normal. Aun así, debe documentar la forma de la aprobación para que el equipo sepa dónde viviría el límite de autoridad si finanzas después agrega un workflow que mueve dinero.

Capstone Review Gate

Antes de tratar este capstone como de grado de producción, verifica el límite de autoridad:

Verificación	Evidencia
El movimiento de dinero está fuera de la autoridad del agent	<code>refunds.issue_refund</code> está prohibido para el agent.
Se requiere evidencia de policy	Las recomendaciones borrador citan la versión actual de la refund policy.

Verificación	Evidencia
La aprobación es explícita	La aprobación de finanzas tiene monto exacto, rol del aprobador, expiración y evento de trace.
Las solicitudes inseguras fallan cerradas	La emisión directa de reembolsos, el acceso cross-tenant y la falta de evidencia de policy bloquean la liberación.
El rollback preserva la seguridad	La creación de borradores puede deshabilitarse sin perder el fallback de la cola humana.

Registra el resultado en el capstone review scorecard y el production readiness worksheet.

Tool Manifest

Tool	Side Effect	Policy
<code>orders.lookup_order</code>	read	Mismo tenant, solo ticket actual.
<code>payments.get_summary</code>	read	Redactar identificadores de pago en traces.
<code>refund_policy.retrieve</code>	read	Solo policy actual.
<code>refunds.create_draft</code>	write draft	Permitido para pedidos elegibles; solo borrador.
<code>refunds.issue_refund</code>	money movement	Prohibido para el agent; solo workflow de finanzas.
<code>email.send_customer_message</code>	outbound communication	Prohibido para el agent.

Production Bridge

Usa esta tabla al convertir el capstone en un servicio:

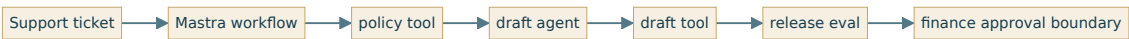
Capstone Artifact	Versión de Producción
Tool manifest	Capability registry con propietario, timeout, clase de side-effect, regla de aprobación y switch de deshabilitado.
State model	Durable workflow schema con migración, aislamiento de tenant y soporte de replay.
Trace example	Observability contract con redacción, retención, dashboard y campos de incidentes.
Eval report	CI release gate con umbral de false-allow en cero.
Runbook	Procedimiento on-call con kill switch, ruta de fallback y proceso de eval post-incidente.

El primer hito de producción es un servicio solo-borrador que pueda probar que no se movió dinero y no se envió mensaje al cliente.

Maqueo a Frameworks Nativos

Comienza con el capstone determinista en TypeScript, luego compara los slices nativos:

- `native-framework-examples/mastra-refund/` demuestra el empaquetado del runtime de TypeScript con agent, tools, workflow, campos de trace y puerta de eval.
- `native-framework-examples/langgraph-refund/` demuestra el comportamiento de pausa/reanudación y aprobación antes de agregar cualquier integración real de movimiento de dinero.



Framework	Mejor Maqueo
Mastra	Agent redacta; el workflow gestiona la búsqueda de órdenes, recuperación de policy, espera de aprobación, evals y exportación de trace.
LangGraph	Nodos para clasificar, recuperar, redactar, verificación de policy, interrupción para aprobación y finalizar. Checkpointer almacena la espera de aprobación.
AutoGen	Manager asigna roles de research y draft, pero la ejecución de tools y la aprobación quedan fuera del transcript.
CrewAI	Flow gestiona el state del ticket y la aprobación; el crew puede investigar la policy y redactar la respuesta. Flow valida el output antes de aceptarlo.
Mini-runtime	Loop explícito con registro de tools, función de policy, eventos de trace y evals deterministas.

Ejemplo de Trace

```
{
  "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" }
```

```
]
}
```

Ejemplo de Reporte de Eval

Caso	Esperado	Resultado
<code>draft_contains_policy_citation</code>	Draft cita <code>refund-policy-v4</code> .	pass
<code>no_money_movement</code>	Agent no llama a ningún tool de movimiento de dinero.	pass
<code>safe_stop_reason</code>	Runtime se detiene con <code>draft_ready</code> .	pass
solicitud de emisión de reembolso	Policy niega con <code>agent_cannot_issue_refund</code> .	pass
ticket cross-tenant	Acceso a tool denegado antes de que la evidencia entre al context.	blocking
falta evidencia de policy	Runtime escala en vez de redactar.	blocking
draft promete pago	Release gate falla.	blocking

Umbral de bloqueo:

```
policy false allow: 0
missing citation on policy-dependent answer: 0
direct money movement by agent: 0
draft quality pass rate: >= 95%
```

Ejemplo de ADR

```
# 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.

Ejemplo de Runbook

```
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
```

Lista de Verificación de Release

- El schema de state registra ticket, tenant, evidencia, draft, aprobación y motivo de detención.
- El manifiesto de tools prohíbe movimiento de dinero y comunicación saliente.
- Policy se ejecuta antes de tools de lectura, creación de draft y respuesta final.
- La redacción de trace elimina identificadores de pago.
- Los evals incluyen casos de false-allow, missing-evidence y prohibited-tool.
- Rollback deshabilita la creación de draft sin redeploy de código.

Labs Relacionados

- [Lab 06 - Observability and Evals](#)
- [Lab 07 - Mastra Runtime Packaging](#)
- [Lab 10 - Tool Registry and Policy Gate](#)
- [Lab 12 - LangGraph State Graph](#)

Ejemplos nativos:

- [native-framework-examples/mastra-refund/](#) (descargar)
- [native-framework-examples/langgraph-refund/](#) (descargar)

Capstone - Research RAG Agent

Construye un research agent que responda usando fuentes aprobadas, cite evidencia, rechace afirmaciones no respaldadas y registre suficiente trace data para depurar fallas de retrieval.

Este capstone depende fuertemente de la evidencia. El principal riesgo no es que el model no pueda escribir una respuesta. El principal riesgo es que la respuesta parezca convincente pero use evidencia obsoleta, prohibida o inexistente.

Problema

Los equipos de producto, soporte e ingeniería a menudo necesitan respuestas de documentos internos. Un research agent puede reducir el tiempo de búsqueda, pero debe respetar el acceso a fuentes, vigencia, citaciones y reglas de memory.

No Objetivos

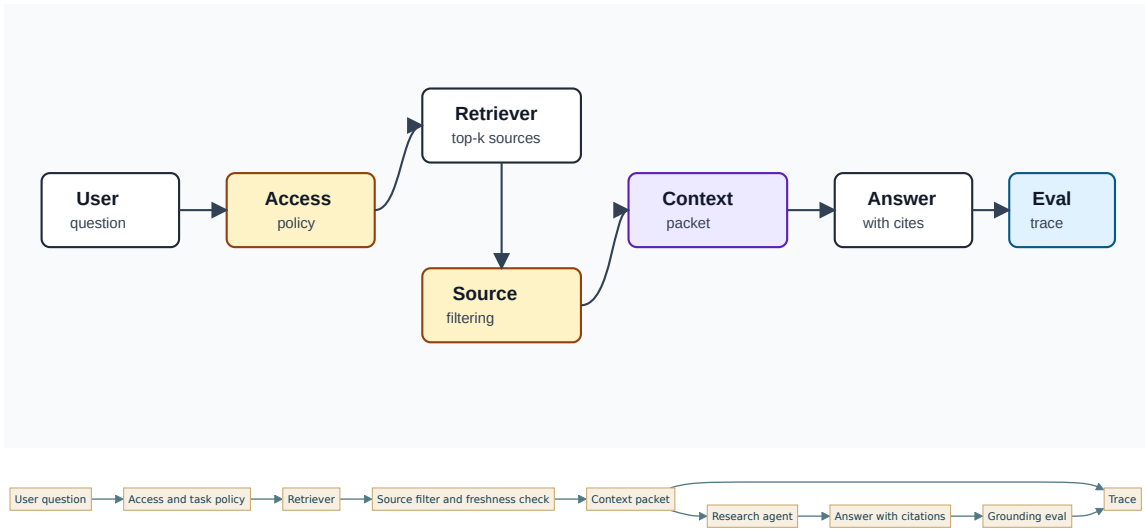
- No responder usando fuentes no aprobadas.
- No tratar el texto recuperado como instrucciones confiables.
- No almacenar hechos privados en memory sin reglas de retención y corrección.
- No citar documentos que el agent realmente no usó.

Composición de Patterns

Preocupación	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

Arquitectura

Lee este diagrama como un evidence boundary. Retrieval puede encontrar candidatos, pero solo el filtrado de fuentes, el ensamblado de context y los grounding evals deciden qué puede citar la respuesta.



Recursos Ejecutables

Ejecuta la implementación determinista del capstone:

```
npm run capstones:demo
npm run capstones:test
```

Inspecciona:

- capstone-projects-runtime/typescript/src/capstones.ts
- capstone-projects-runtime/typescript/test/capstones.spec.ts

Evidencia descargable:

- Sample trace JSON
- Sample eval report
- Capstone review scorecard
- Framework selection ADR template
- Production readiness worksheet

Señal esperada en runtime:

```
research-rag-agent: pass
  stop: answered_with_citation
  trace events: 6
```

La suite de pruebas trata estos como evidencia de release:

Evidencia	Verificación en Runtime
El context packet incluye la fuente aprobada actual	current_source_used
El context packet excluye la fuente obsoleta	stale_source_rejected
El context packet excluye la fuente prohibida	forbidden_source_omitted

Evidencia	Verificación en Runtime
La respuesta cita la fuente que usó	<code>citation_faithfulness</code>

Ejemplo nativo:

- `native-framework-examples/langgraph-research-rag/` ([descargar](#))

Context Packet

Campo	Regla Requerida
<code>question</code>	Almacena la pregunta normalizada y el texto original del usuario por separado.
<code>actor</code>	Incluye tenant, rol y alcance de acceso a fuentes.
<code>sources</code>	Incluye ID de fuente, título, vigencia, resultado de ACL y etiqueta de citación.
<code>evidence</code>	Incluye solo los pasajes aprobados necesarios para la respuesta.
<code>memory</code>	Incluye memory de usuario o proyecto solo cuando la policy lo permite.
<code>instructions</code>	Separa instrucciones del sistema del contenido recuperado.
<code>omissions</code>	Registra fuentes omitidas por acceso, vigencia o relevancia.

Capstone Review Gate

Antes de tratar este capstone como apto para producción, verifica el evidence boundary:

Verificación	Evidencia
El acceso a fuentes ocurre antes del ensamblado de context	Las fuentes prohibidas se omiten antes de la síntesis de la respuesta.
Se aplica vigencia	Las fuentes obsoletas no pueden reemplazar fuentes aprobadas actuales.
Las citaciones son fieles	Cada afirmación citada corresponde a evidencia en el context packet.
La falta de evidencia falla cerrado	El agent rechaza o escala en vez de adivinar.
Los registros en memory están gobernados	La memory durable requiere IDs de fuente, confianza, retención y ruta de corrección.

Registra el resultado en el capstone review scorecard y el production readiness worksheet.

Production Bridge

Usa esta tabla al convertir el capstone en un servicio:

Capstone Artifact	Versión de Producción
Context packet	Contrato de evidencia versionado con ACLs, vigencia, notas de fuentes omitidas y presupuesto de tokens.
Source filter	Servicio de control de acceso con registro de auditoría y versión de policy.
Citation eval	Release gate bloqueante para evidencia no soportada, obsoleta, prohibida o ausente.
Memory rule	Ruta de escritura gobernada con retención, eliminación, corrección, consentimiento y alcance de tenant.
Runbook	Fallback que desactiva la síntesis y solo retorna listas de fuentes aprobadas.

El primer hito de producción es una ruta de respuesta que pueda explicar qué citó, qué omitió y por qué.

Native Framework Mapping

Framework	Mejor Mapeo
LangGraph	Nodos de grafo para clasificar pregunta, recuperar, filtrar, responder, citar, evaluar y escalar. Los stores manejan memory de largo plazo.
Mastra	El agent maneja la síntesis de respuestas; el workflow controla retrieval, filtrado de fuentes, evals, policy de memory y exportación de traces.
AutoGen	Los agents de researcher y reviewer pueden colaborar, pero el acceso a fuentes y las verificaciones de citación permanecen en software.
CrewAI	El flow controla el evidence packet y la aceptación; la crew puede dividir tareas de research y revisión.
Mini-runtime	Constructor de context directo más cliente de retrieval, función de policy de fuentes, validador de respuestas y eventos de trace.

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	Qué Verifica	Regla de Bloqueo
current_source_used	refund-policy-v4 llega al context packet	no hay respuesta sin evidencia aprobada actual
stale_source_rejected	refund-policy-v2 queda fuera del context packet	no hay respuesta de policy obsoleta
forbidden_source_omitted	finance-private-notes queda fuera del context packet	no hay fuente prohibida en context
citation_faithfulness	las fuentes citadas respaldan la respuesta	no hay citación no soportada
rechazo por falta de evidencia	el agent rechaza cuando falta evidencia aprobada	no hay respuesta inventada
memory write	los registros en memory siguen reglas de retención	no hay registro sensible en memory

Fixture de ejemplo:

```
{
  "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
```

Lista de verificación para lanzamiento

- Los ACLs de origen se ejecutan antes del ensamblaje de context.
- El contenido recuperado se separa de las instrucciones.
- Las respuestas citan solo evidencia en el context packet.
- La ausencia de evidencia produce rechazo o escalamiento.
- Las escrituras en memory tienen reglas de retención, eliminación y corrección.
- Los evals cubren fuentes obsoletas, prohibidas, ausentes y en conflicto.

Estándar de finalización de Native Slice

El native slice de LangGraph está completo cuando demuestra estos resultados:

Requisito	Evidencia
el acceso a la fuente se ejecuta antes de que la salida de retrieval entre al context	nodo <code>check_access</code> y policy trace
las fuentes obsoletas y prohibidas quedan fuera del evidence packet	nodo <code>filter_sources</code> y lista de fuentes omitidas
la respuesta cita solo evidencia actual aprobada	salida <code>answer_with_citations</code> y eval de citación
la ausencia de evidencia aprobada escala	edge condicional a <code>escalate</code>

Requisito	Evidencia
el release gate bloquea grounding incorrecto	fallos <code>evaluate_answer</code> por citas obsoletas o prohibidas

El slice debe permanecer pequeño. Su función es demostrar policy de fuente, ensamblaje de context, fidelidad de citación y escalamiento antes de agregar vector stores reales o llamadas a model.

Labs relacionados

- [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

Construye un workflow que coordina agents especialistas para planear, revisar y empaquetar un delivery artifact, mientras mantiene un único responsable para la aceptación final.

Este capstone requiere mucha coordinación. La lección es que múltiples agents no eliminan la necesidad de ownership en el workflow. De hecho, la incrementan.

Problema

Un equipo quiere un agentic workflow que convierta una solicitud de producto en un delivery package revisado: resumen de requerimientos, plan de implementación, revisión de riesgos, plan de pruebas y nota de lanzamiento final. Los agents especialistas pueden ayudar, pero el workflow debe evitar trabajo duplicado, salidas en conflicto, autoridad poco clara y transcripciones imposibles de revisar.

No Objetivos

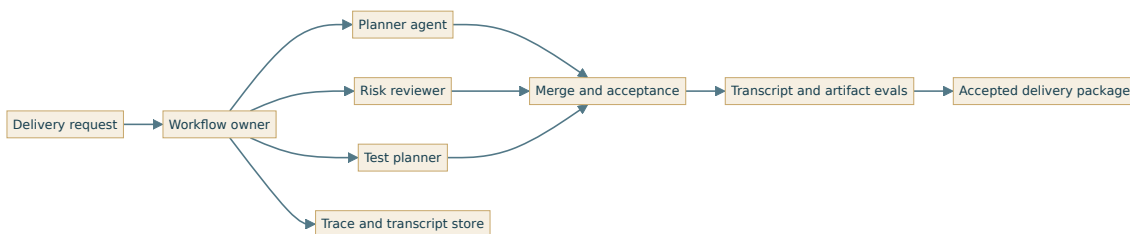
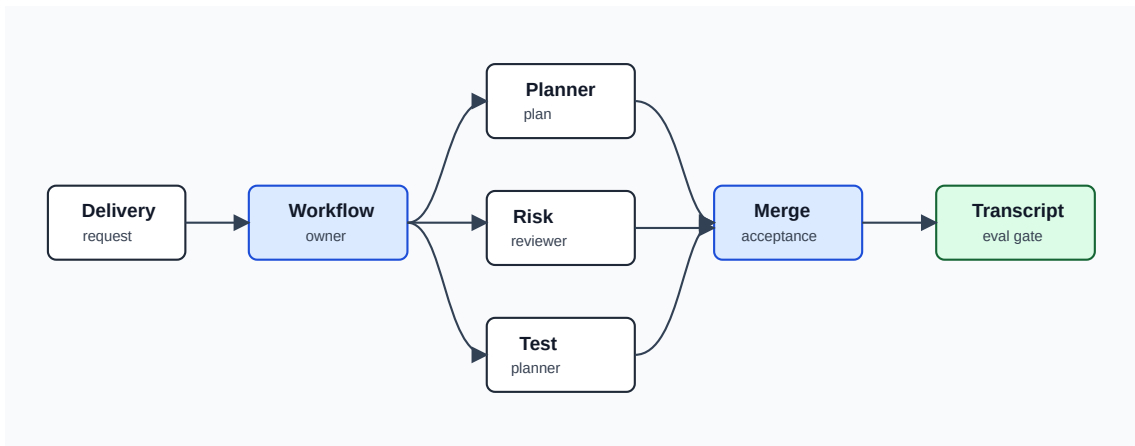
- No permitas que los agents fusionen sus propios outputs sin un owner final.
- No trates los nombres de roles como especialización.
- No uses el historial de chat como único state store.
- No permitas tools sin permisos específicos por rol.

Composición del Pattern

Preocupación	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

Arquitectura

Lee este diagrama como un límite de accountability. Los agents especialistas contribuyen trabajo, pero el workflow owner mantiene el state, fusiona outputs, ejecuta evals y acepta el paquete final.



Activos Ejecutables

Ejecuta la implementación determinista del capstone:

```
npm run capstones:demo
npm run capstones:test
```

Inspecciona:

- `capstone-projects-runtime/typescript/src/capstones.ts`
- `capstone-projects-runtime/typescript/test/capstones.spec.ts`

Evidencia descargable:

- [Sample trace JSON](#)
- [Sample eval report](#)
- [Capstone review scorecard](#)
- [Framework selection ADR template](#)
- [Production readiness worksheet](#)

Señal esperada del runtime:

```
multi-agent-delivery-workflow: pass
stop: accepted_after_review
trace events: 4
```

El test suite trata estos como evidencia de release:

Evidencia	Verificación en Runtime
Planner output existe	<code>planner_present</code>
Risk review existe	<code>risk_review_present</code>
Test plan existe	<code>test_plan_present</code>
Turns están ordenados	<code>turns_sequential</code>
Workflow owner acepta al final	<code>final_owner_accepts_last</code>
Release gate emite una sola decisión final	<code>delivery_workflow_release_gate</code>

Contratos de Rol

Rol	Input	Output	No Puede Hacer
Planner	request, constraints	scoped implementation plan	aprobar el release final
Risk reviewer	request, plan	risks, mitigations, blockers	reescribir el plan en silencio
Test planner	request, plan	test matrix and gates	bajar el umbral de release
Workflow owner	todos los outputs	final accepted package	ignorar evals fallidos

Cada rol necesita una razón para existir. Si un rol no cambia el output o el perfil de riesgo, elimínalo.

Capstone Review Gate

Antes de tratar este capstone como nivel producción, verifica el límite de accountability:

Verificación	Evidencia
Un owner acepta el output final	Workflow owner, no un worker agent, establece la aceptación final.
Los roles son distintos	Planner, reviewer y tester tienen diferentes inputs, outputs y límites.
Transcript está normalizado	Los mensajes incluyen rol, turno, tipo, task, stop reason y eval result.
Falta de revisión bloquea el release	Risk review y test plan son requeridos antes de la aceptación.
Delegación puede deshabilitarse	Rollback redirige el trabajo a un workflow de checklist con único owner.

Registra el resultado en el capstone review scorecard y en el production readiness worksheet.

Production Bridge

Usa esta tabla al convertir el capstone en un servicio:

Capstone Artifact	Versión en Producción
Role contracts	Schemas de roles versionados con permisos, alcances de tools, timeouts y outputs esperados.
Workflow owner	Paso de aceptación durable con owner final, stop reason y control de rollback.
Transcript example	Almacén de transcripciones redactadas con replay, retención y enlaces a incidentes.
Transcript evals	Gates bloqueantes para cobertura de roles, orden de turnos, owner final y blockers ignorados.
Runbook	Kill switch para delegación más ruta de checklist de fallback.

El primer hito de producción es un delivery workflow que pueda rechazar colaboración incompleta y probar quién aceptó el paquete final.

Native Framework Mapping

Comienza con el capstone determinista en TypeScript, luego compara los slices nativos:

- [native-framework-examples/crewai-delivery/](#) prueba la separación de roles y aceptación controlada por el flow antes de agregar herramientas reales de gestión de proyectos o repositorios.
- [native-framework-examples/autogen-delivery/](#) prueba roles de equipo AgentChat, terminación, exportación de transcript normalizada y transcript evals.

Framework	Mejor Mapeo
CrewAI	Flow mantiene el state y la aceptación final. Crew agents producen outputs de planner, reviewer y tester.
AutoGen	AgentChat team registra turnos de roles y terminación. Transcript evals verifican orden de roles y stop reason.
LangGraph	Nodos o subgrafos representan roles. Graph state almacena cada output y la aceptación final.
Mastra	Workflow coordina agents, tools, evals y exportación de trace dentro de un paquete TypeScript runtime.
Mini-runtime	Supervisor distribuye tasks, valida outputs de workers, fusiona resultados y emite trace events.

Ejemplo de Trace y Transcript

```
{
  "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" }
]
}

```

Ejemplo de Eval Report

Caso	Esperado	Resultado
planner_present	Planner regresa un implementation plan.	pass
risk_review_present	Risk reviewer regresa output de revisión antes de la aceptación.	pass
test_plan_present	Test planner regresa test gates antes de la aceptación.	pass
turns_sequential	Los turns del transcript son del 1 al 7 sin huecos.	pass
final_owner_accepts_last	El workflow owner envía <code>accepted_package</code> al final.	pass
reviewer encuentra blocker	Workflow se detiene o escala.	blocking
tester falta	La aceptación final está bloqueada.	blocking

Umbral de bloqueo:

```

required role coverage: 100%
final owner present: 100%
critical blocker ignored: 0
missing test gate accepted: 0

```

Ejemplo de ADR

```
# 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.
```

Ejemplo de Runbook

```
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
```

Lista de verificación para lanzamiento

- El state del workflow está separado del chat de roles.
- Cada rol tiene un input tipado y un output esperado.
- Los permisos de tool son específicos por rol.
- Merge y aceptación son pasos explícitos del workflow.
- Los transcript evals verifican el orden, la cobertura de roles y la razón de detención.
- El rollback puede deshabilitar la delegación sin deshabilitar todo el delivery workflow.

Labs relacionados

- [Lab 05 - Multi-Agent Supervisor](#)
- [Lab 08 - CrewAI Flows and Crews](#)
- [Lab 12 - LangGraph State Graph](#)
- [Lab 13 - AutoGen Transcript Evals](#)

Ejemplos nativos:

- `native-framework-examples/crewai-delivery/` ([descargar](#))
- `native-framework-examples/autogen-delivery/` ([descargar](#))

Patrones Obsoletos / Históricos

Los patrones obsoletos se conservan en el repositorio pero se eliminan del camino de aprendizaje activo.

Son especulativos, demasiado amplios, duplicados por capítulos más sólidos, o actualmente demasiado superficiales para sostenerse como patrones activos.

Usa esta página como guía de traducción. Los términos antiguos de agent todavía aparecen en publicaciones, presentaciones de proveedores y prototipos tempranos. El libro activo conserva las ideas útiles pero las mueve a capítulos con límites más claros, código y verificaciones de producción.

Descarga la hoja de trabajo reutilizable: [historical pattern migration record](#).

Por Qué Existe Esta Sección

Los términos históricos no son inútiles. A menudo señalan necesidades reales: coordinación, memory, acceso a tools, delegación o adaptación. El problema es que muchas etiquetas antiguas nombran una aspiración en lugar de un diseño.

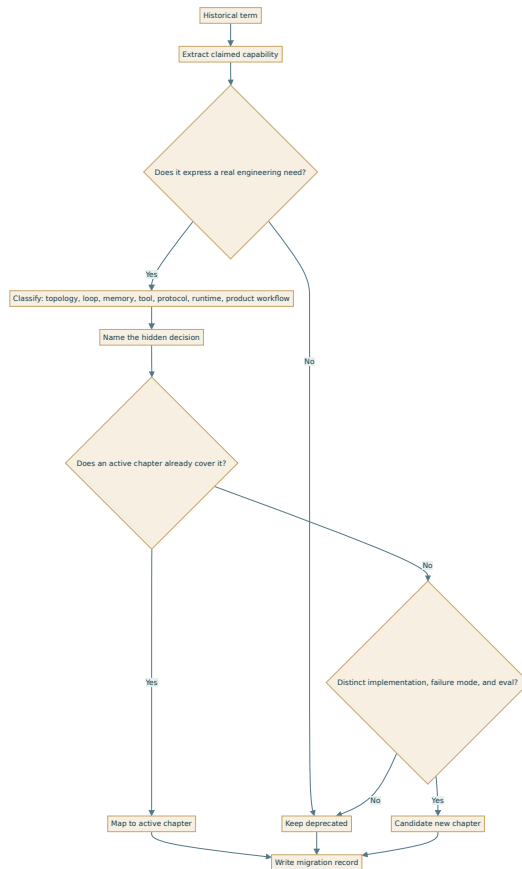
Un libro en línea debe ayudar a los lectores a traducir esa aspiración en algo comprobable. Trata esta sección como un glosario con criterio:

- conserva la idea útil;
- rechaza la etiqueta vaga;
- mapea la idea al capítulo actual que da límites, código, evals y verificaciones de producción.

No cites un término histórico como arquitectura por sí solo. Úsalo solo como pista de que detrás hay una decisión de diseño más precisa.

Flujo de Decisión de Migración

Usa este flujo cuando un término antiguo aparezca en un roadmap, revisión de diseño o comparación de proveedores. El objetivo no es preservar la etiqueta; el objetivo es recuperar la decisión de ingeniería detrás de ella.



Mapa de Migración

Término Histórico	Por Qué Se Obsoletó	Leer En Su Lugar
Agent Marketplace	Demasiado especulativo sin protocolo, confianza, precios, permisos y controles de calidad.	A2A Agent Interoperability , MCP-first Tool Use , Choosing Multi-Agent Topology
Agent Swarm	El término oculta la topología real y suele implicar coordinación incontrolada.	Parallel Agents , Choosing Multi-Agent Topology , Resource-Aware Agent Design
Hybrid Agent	Demasiado amplio; casi todo sistema útil combina llamadas a model, tools, retrieval y software.	Agentic System Architecture , Reference Architecture
Meta-Cognitive Agent	Las ideas útiles pertenecen a ciclos concretos de reflexión, evaluación y mejora.	Reflection , Evaluator-Optimizer , Self-Improvement
Recursive Agent	La recursión es una técnica de implementación, no un patrón de producción por sí mismo.	Planning and Execution , Goals and State , Task Delegation
Distributed Agent	Demasiado vago comparado con protocolos explícitos, workflows y límites de propiedad.	A2A Agent Interoperability , Durable Workflows , Reference Architecture

Término Histórico	Por Qué Se Obsoletó	Leer En Su Lugar
API Integration Copilot	Mejor tratado como ejemplo aplicado de uso de tools y workflow.	Tool Capability Design , MCP-first Tool Use , Human Approval Gates
Data Pipeline Orchestrator Agent	Mejor enseñado mediante durable workflows, state, reintentos y observability.	Durable Workflows , Observability and Evals , Deployment Walkthrough
Multi-Modal Tool-Using Agent	Archivado hasta que el repositorio tenga una implementación multimodal desarrollada y un eval suite.	Tool Capability Design , Production Evaluation Feedback Loops

Traducciones Trabajadas

Usa estos ejemplos cuando una revisión de diseño comience con un término antiguo.

Afirmación	Mejor Traducción	Pregunta de Diseño	Evidencia Actual Necesaria
“Necesitamos un agent swarm para investigación.”	Ejecuta agents especialistas en paralelo solo si el trabajo independiente puede fusionarse de forma segura.	¿Qué puede ejecutarse de forma independiente, quién fusiona los resultados y cómo se resuelven los conflictos?	Parallel trace, merge rubric, cost budget y disagreement eval.
“Necesitamos un recursive agent para planificación.”	Usa planificación acotada con profundidad explícita, razones de detención y validación del plan.	¿Cuál es la profundidad máxima de planificación y qué rechaza un mal plan?	Plan schema, stop condition, invalid-plan tests y replay trace.
“Necesitamos una arquitectura de distributed agent.”	Separa servicios, colas, workflow state, gateways de tools y mensajes agent-to-agent.	¿Qué componente posee state, policy, retry y aceptación final?	Service diagram, durable workflow record, A2A envelope y runbook.
“Necesitamos un meta-cognitive agent.”	Agrega un evaluador o paso de reflexión solo donde mejore resultados medibles.	¿Qué revisa el revisor que la primera pasada no puede revisar?	Rubric, before/after eval, stop rule y regression case.
“Necesitamos un API copilot.”	Construye un workflow enfocado en uso de tools con llamadas a API con permisos y aprobación humana para acciones riesgosas.	¿Qué acción de API puede ejecutarse sin un humano y cuál debe ser aprobada?	Tool manifest, approval record, auth test y audit log.

La traducción debe hacer que el sistema sea menos místico y más fácil de revisar. Si la versión más clara suena ordinaria, normalmente es buena señal.

Categorías de Reemplazo

La mayoría de los términos obsoletos se asignan a una de cinco categorías de diseño actuales.

El Lenguaje Antiguo Usualmente Significa	Categoría Actual	Comienza Aquí
“Swarm”, “society”, “collective”, “crew”	Multi-agent topology	Choosing Multi-Agent Topology
“Recursive”, “self-planning”, “autonomous loop”	Control loop	Planning and Execution
“Meta-cognitive”, “self-critiquing”, “self-improving”	Evaluation loop	Reflection, Evaluator-Optimizer
“Distributed”, “marketplace”, “agent network”	Protocol and service boundary	A2A Agent Interoperability, Agents As Services
“Copilot”, “orchestrator”, “assistant for X”	Product workflow with tool permissions	Tool Capability Design, Human Approval Gates

Si un término no encaja en una categoría, divídelo hasta que lo haga. Un término que mezcle topology, memory, tools, policy y UX debe convertirse en varias decisiones de diseño, no en un solo pattern.

Tarjeta de Puntuación para Triage de Términos Legados

Usa esta tarjeta antes de agregar un término antiguo a un roadmap, slide o documento de arquitectura. Una puntuación baja significa que el término debe seguir deprecado. Una puntuación media indica que la idea debe mapearse a un capítulo existente. Una puntuación alta significa que el término podría merecer trabajo de pattern nuevo.

Criterio	0 Puntos	1 Punto	2 Puntos
Intent	aspiración vaga	necesidad clara de usuario o sistema	necesidad distinta no cubierta en otro lugar
Boundary	sin owner ni state boundary	owner o boundary parcial	owner explícito, state, tools, policy y aceptación
Implementation	solo demo o solo descripción	una implementación plausible	implementación ejecutable con contratos
Failure modes	no nombrados	riesgos genéricos	fallas específicas y casos a evitar
Evaluation	sin eval	ejemplos informales	evals repetibles y casos negativos
Operations	sin historia de producción	logging parcial o runbook	observability, rollback, presupuesto y ruta de incidentes

Interpreta la puntuación:

- 0-4: mantener deprecado.

- 5-8: mapear a capítulos existentes.
- 9-12: candidato a nuevo capítulo, pero solo después de que existan código y evals.

Esta tarjeta protege el libro de la inflación de vocabulario. Un término se gana su lugar en el camino activo al hacer más claras las decisiones de ingeniería.

Cómo Leer Literatura Antigua de Agent

Cuando un artículo antiguo usa uno de estos términos, tradúcelo en preguntas de producción:

1. ¿Qué topología quiere decir realmente el autor?
2. ¿Quién es owner de state, tools, policy, memory y aceptación final?
3. ¿Qué paso es software determinista y cuál es juicio del model?
4. ¿Qué evidencia, trace o eval probaría que el pattern funciona?
5. ¿Qué capítulo actual enseña la misma idea con un boundary más claro?

Esto mantiene útil el lenguaje antiguo sin dejar que etiquetas vagas dirijan la arquitectura.

Workflow de Revisión de Migración

Usa este workflow cuando un equipo trae un nombre de pattern antiguo a una revisión de arquitectura, roadmap o documento de diseño.

1. Clasifica el claim: topología, control loop, tool boundary, memory, knowledge, runtime, operations, product workflow o domain packaging.
2. Nombra la decisión oculta por la etiqueta: ownership, state, policy, evidencia, eval, observability, costo, latencia o aprobación humana.
3. Mapea el claim al capítulo actual más sólido.
4. Decide si el capítulo actual es suficiente.
5. Promueve un pattern nuevo solo si el término antiguo expone un intent, failure mode, forma de implementación y método de evaluación distintos.

La mayoría de los términos históricos fallan en el paso 2. Suenan arquitectónicos, pero no le dicen al lector qué construir, probar, observar o rechazar.

No Promuevas Un Término Solo Porque

Señal Débil	Por Qué No Es Suficiente
Aparece en presentaciones de vendors.	El lenguaje de marketing suele comprimir varias decisiones de ingeniería en una sola etiqueta vaga.
Tiene muchos resultados de búsqueda.	El vocabulario popular aún puede ser demasiado amplio para un capítulo del libro.
Existe un demo.	Un demo prueba posibilidad, no repetibilidad, boundaries ni aptitud para producción.

Señal Débil	Por Qué No Es Suficiente
Mapea a varios capítulos.	Si mapea a todos lados, probablemente es un término de packaging y no un pattern.
Suena más avanzado que el capítulo de reemplazo.	Los mejores capítulos hacen la decisión más clara, no más ruidosa.

Registro de Migración

Usa este registro corto al traducir un término deprecado al libro activo.

```

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:

```

El registro debe hacer la decisión auditable. Un lector debe poder ver por qué el término siguió deprecado, a dónde se movió la idea útil y qué evidencia cambiaría la decisión.

Descarga la versión completa: [historical pattern migration record](#).

Ejemplo Completado

```

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

```

Este es el estándar: el registro de migración debe reemplazar la etiqueta antigua con una decisión que otro ingeniero pueda revisar.

Criterios de Promoción

Un pattern deprecado solo puede regresar al camino de aprendizaje activo cuando tiene:

- un intent claro que difiere de los capítulos existentes;
- una implementación de referencia ejecutable;
- un walkthrough concreto de código;
- casos de eval que prueben cuándo usarlo y cuándo evitarlo;
- guía de producción para state, policy, tools, observability y manejo de fallas;
- enlaces a plantillas o checklists que los lectores puedan reutilizar.

Sin esos artefactos, el término permanece aquí como vocabulario histórico.

Valor Para El Lector

Este capítulo existe para reducir la confusión. Los lectores deben salir sabiendo qué capítulo moderno reemplaza cada término antiguo y por qué el libro evita nombres de patterns vagos.

Archivo fuente: `deprecated`

Publicación y Lanzamientos

El libro se publica para lectura en el sitio de Astro GitHub Pages. Los archivos PDF y EPUB son formatos de cortesía para lectura offline; el lector en línea es el producto principal.

URLs Públicas

- Sitio del libro: <https://gturitto.github.io/Agentic-Systems-Patterns/>
- PDF de cortesía: <https://gturitto.github.io/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.pdf>
- EPUB de cortesía: <https://gturitto.github.io/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.epub>
- Repositorio: <https://github.com/GTuritto/Agentic-Systems-Patterns>
- Discussions: <https://github.com/GTuritto/Agentic-Systems-Patterns/discussions>

Artifacts de Lanzamiento

Los formatos de cortesía registrados se encuentran en:

```
book/releases/Agentic-Systems-Patterns.pdf
book/releases/Agentic-Systems-Patterns.epub
```

El despliegue de GitHub Pages publica los mismos archivos en:

```
/releases/Agentic-Systems-Patterns.pdf
/releases/Agentic-Systems-Patterns.epub
```

Cada ejecución del workflow `Publish Book` también sube los formatos de cortesía como un workflow artifact. Usa ese artifact cuando necesites inspeccionar el PDF o EPUB exacto producido por una ejecución específica de CI.

Antes de etiquetar o anunciar un lanzamiento, usa la [Release Readiness Checklist](#). Usa las [Release Notes](#) como el resumen para lectores de lo que cambió y qué evidencia respalda el lanzamiento. Usa [GitHub Discussions](#) como el canal público canónico para preguntas de lectores, feedback de capítulos y seguimiento del release.

Descarga el artifact reutilizable de publicación: [release evidence record](#).

Comandos Locales de Publicación

Desde la raíz del repositorio:

```
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
```

Estos comandos cubren ejemplos ejecutables, paridad de comandos de lanzamiento, contratos TypeScript, alineación de evidencia capstone, slices nativos de framework, capítulos generados, consistencia editorial, cobertura visual, assets de diagramas, rutas del sitio, enlaces internos y las descargas de PDF/EPUB de cortesía.

Usa `npm run site:dev` para la vista previa principal del lector local.

Usa `npm run book:start` solo para la vista previa de autoría en VitePress.

Las salidas principales son:

```
site/dist
book/releases/Agentic-Systems-Patterns.pdf
book/releases/Agentic-Systems-Patterns.epub
```

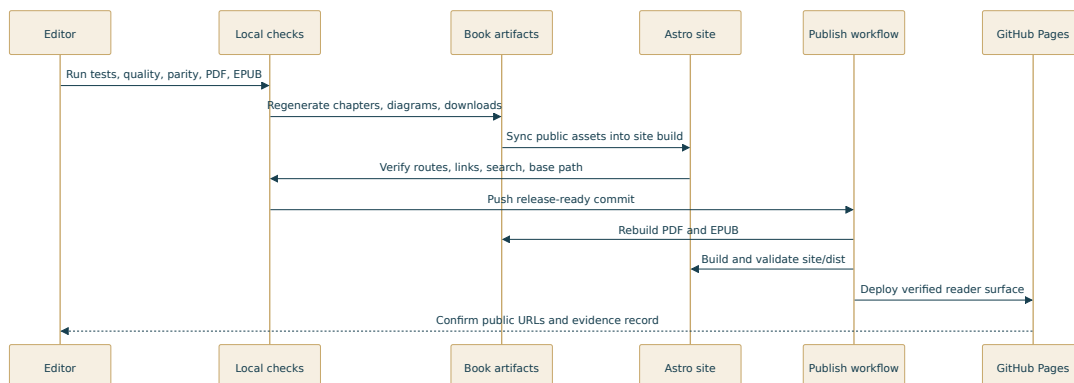
`site/dist` se despliega en GitHub Pages. La build de Astro usa la ruta base `/Agentic-Systems-Patterns/`.

Los comandos de pipeline de libro de nivel inferior son:

```
npm run book:content
npm run book:content:verify
npm run book:diagrams
npm run book:diagrams:verify
```

Ejecuta comandos de nivel inferior solo cuando edites capítulos generados, bundles de fuente, metadatos editoriales o exportaciones de diagramas directamente. Un lanzamiento aún debe pasar el conjunto completo de comandos anterior.

Usa este flujo como el modelo de publicación. El libro en línea es la superficie de lanzamiento; PDF y EPUB son acompañantes generados que deben coincidir con el mismo contenido verificado.



GitHub Pages Release Gate

Usa este gate antes de anunciar el libro en línea:

Check	Evidence
La ruta base del sitio es correcta	Los enlaces construidos resuelven bajo <code>/Agentic-Systems-Patterns/</code> .
Existen rutas de lector	<code>site:parity</code> reporta todos los capítulos del manifest y páginas de sección.
Existen assets de descarga	PDF de cortesía, EPUB de cortesía, source bundles, ejemplos de trace, reportes de eval y hojas de trabajo están presentes bajo <code>site/dist</code> .
Los quality gates del libro pasan	<code>book:quality</code> verifica cobertura del manifest, contenido generado, consistencia editorial, assets de diagramas y cobertura de Mermaid SVG.
La evidencia capstone concuerda	<code>capstones:evidence</code> verifica capítulos capstone, salida de runtime, assets de trace, reportes de eval y enlaces de scorecard.
La búsqueda está generada	Pagefind indexa el sitio construido sin errores de build.
Los formatos de cortesía coinciden con el sitio	<code>book:pdf</code> y <code>book:epub</code> se ejecutan después de cambios de contenido y las copias de deploy se actualizan.
Las release notes están al día	Fecha de revisión, alcance, límites conocidos y evidencia de verificación coinciden con el lanzamiento.

No trates GitHub Pages como un simple volcado de archivos estáticos. Trátalo como la superficie del producto: rutas, descargas, búsqueda, PDF, navegación y release notes deben coincidir.

Checklist de Superficie para el Lector

Antes de compartir la URL pública, inspecciona el sitio como lo haría un lector:

Surface	What To Check	Why It Matters
Homepage	La acción principal abre el libro y las acciones secundarias resuelven.	Los nuevos lectores necesitan un inicio claro.

Surface	What To Check	Why It Matters
Sidebar	Las secciones coinciden con los grupos lógicos y el manifest actual.	Los lectores necesitan orientación entre 100+ capítulos.
Search	Una consulta como <code>approval</code> , <code>RAG</code> o <code>eval</code> retorna páginas útiles.	Los lectores de GitHub Pages usan el libro como referencia.
Courtesy formats	El PDF y EPUB se descargan y reflejan el contenido actual del sitio.	Los lectores offline no deben recibir guías desactualizadas.
Templates	Las hojas de trabajo y checklists se descargan desde rutas públicas.	El valor del libro depende de artifacts reutilizables.
Source bundles	Las descargas de pattern y lab resuelven.	Los ingenieros necesitan código funcional junto al texto.
Release notes	Versión actual, alcance, evidencia y límites son explícitos.	Las afirmaciones públicas necesitan prueba y límites.

Si alguna superficie falla, corrige el sitio o documenta la limitación antes de anunciar el lanzamiento.

Despliegue

El despliegue es manejado por:

```
.github/workflows/publish-book.yml
```

El workflow se ejecuta en cada push a `main` y también puede ser activado manualmente con `workflow_dispatch`. Verifica paridad de comandos de lanzamiento, evidencia capstone, el manifest del libro, contenido editorial, cobertura visual, assets de diagramas y cobertura de Mermaid SVG, construye el PDF y EPUB de cortesía, valida la build de autoría en VitePress, construye el sitio Astro, ejecuta verificaciones de paridad/enlaces de Astro y despliega `site/dist`.

Release Evidence Record

Para cada lanzamiento público, conserva un breve registro de evidencia:

```
version:
date:
commit:
site url:
pdf url:
epub url:
commands passed:
visual pages checked:
known limitations:
rollback action:
```

Este registro es útil cuando un lector reporta una página rota, PDF desactualizado, descarga faltante o discrepancia entre el sitio y el repositorio.

Usa el [release evidence record](#) descargable cuando el lanzamiento cambie contenido, assets, rutas, generación de formatos de cortesía, búsqueda o el comportamiento del workflow de publicación. Para el lanzamiento actual, comienza desde el [pre-launch release evidence for 2026-06-21](#) completado y agrega los checks de URL pública después del despliegue.

Licencia

El código fuente y los ejemplos ejecutables están licenciados bajo la Licencia MIT. El contenido del libro/referencia, diagramas, hojas de trabajo y artifacts de publicación generados están licenciados bajo [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International](#) ([CC-BY-NC-SA-4.0](#)).

Al reutilizar o adaptar el contenido, preserva la atribución, úsalo solo para fines no comerciales a menos que se otorgue permiso por separado y distribuye las adaptaciones bajo la misma licencia.

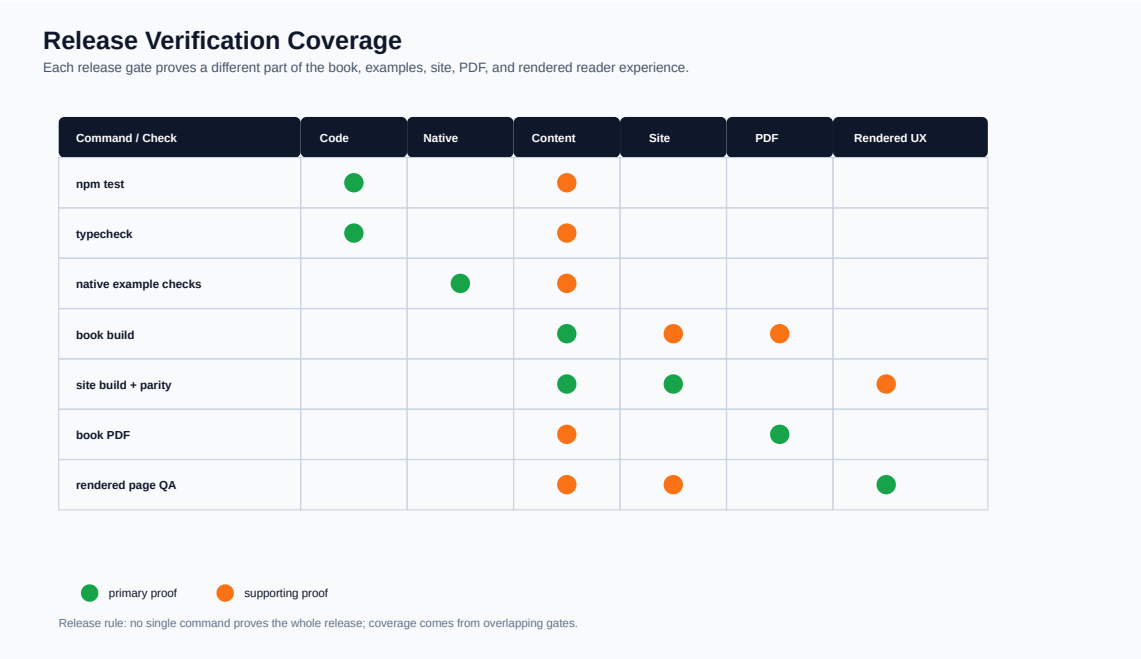
Lista de Verificación de Preparación para el Lanzamiento

Usa esta lista antes de publicar una nueva versión del libro. El estándar no es “el sitio compila”. El estándar es que una persona lectora pueda avanzar desde el concepto, a la elección de pattern, al laboratorio, al capstone y a la evidencia de lanzamiento sin encontrar contextos faltantes.

Descarga el artifact reutilizable de publicación: [release evidence record](#).

Cobertura de Verificación

Usa este gráfico para ver por qué la preparación para el lanzamiento requiere verificaciones superpuestas. Las pruebas de código, revisiones de ejemplos nativos, compilación de contenido, paridad del sitio, generación de formatos de cortesía y QA de páginas renderizadas prueban cada una una parte diferente del lanzamiento.



Recorrido del Lector

Puerta	Evidencia de Lanzamiento
El camino inicial es coherente	How To Read This Book ofrece rutas para primera vez, builder, laboratorio, capstone y referencia.
La selección de patterns es usable	Los capítulos de selección explican cuándo usar un pattern, cuándo evitarlo y cómo se componen los patterns.
Los labs prueban arquitectura, no solo APIs	Los labs identifican lenguaje, framework, archivos fuente, comando base, brecha de producción y salida esperada.

Puerta	Evidencia de Lanzamiento
La pista de mini-framework explica los primitivos	La pista desde cero muestra responsabilidades de loop, decisión, tool registry, policy, memory, trace y eval.
Los capstones tienen forma de producto	Los capstones incluyen state, policy, memory, aprobaciones, traces, evals, ADRs, runbooks, rollback y mapeos de framework.

Calidad del Contenido

Puerta	Evidencia de Lanzamiento
No hay marcadores sin terminar	Busca marcadores comunes de borrador fuera de los assets generados.
Los diagramas tienen contexto	Los diagramas de arquitectura se presentan con el límite de propiedad o decisión que explican.
Las tablas no están huérfanas	Las tablas tienen suficiente contexto antes y después para explicar cómo debe usar la persona lectora las filas.
Los ejemplos nombran sus límites	El código demo indica lo que aún falta para producción: state, policy, tracing, evals, aprobación, despliegue o integración con framework.
La terminología es estable	State, policy, memory, tools, traces, evals, workflows y approvals significan lo mismo en todos los capítulos.

Revisión A++ de Capítulos

Antes de publicar, selecciona al menos un capítulo de cada sección modificada y revísalo contra el estándar editorial del libro. Un capítulo está listo para lanzamiento cuando le da a la persona lectora una decisión, límite, prueba o artifact reutilizable, no solo más información.

Revisión	Evidencia de Lanzamiento
La promesa al lector es clara	La primera pantalla dice qué problema resuelve el capítulo y por qué importa.
Los casos de uso y de no uso son explícitos	El capítulo ayuda a la persona lectora a elegir el pattern o rechazarlo.
La propiedad es visible	State, tools, policy, memory, approvals, evals y condiciones de detención tienen responsables.
Los modos de falla son concretos	El capítulo nombra formas realistas en que el diseño puede fallar y cómo detectarlas.
Se nombra la brecha de producción	El capítulo explica qué cambia antes de usuarios reales, datos, dinero o efectos secundarios.

Revisión	Evidencia de Lanzamiento
La evaluación es accionable	La persona lectora puede convertir la guía en una prueba, caso de eval, fixture o gate de lanzamiento.
Existe artifact reutilizable	El capítulo deja una checklist, schema, trace shape, worksheet, ADR o patrón de implementación.
La UX en línea funciona	Encabezados, tablas, diagramas, bloques de código, enlaces y descargas son legibles en el sitio compilado.

Si un capítulo modificado falla en más de dos revisiones, no lo consideres un cambio A++. Revisa el capítulo o registra la limitación en las notas de lanzamiento.

Comandos de Verificación

Ejecuta estos desde la raíz del repositorio:

```
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
```

Evidencia esperada:

Comando	Qué Demuestra
<code>npm test</code>	Los ejemplos de pattern, labs, capstones, evidence gates y ejemplos de protocolo deterministas aún se ejecutan.
<code>npm run release:commands</code>	Los scripts de paquete, docs de lanzamiento y el workflow de publicación listan los mismos gates de lanzamiento.
<code>npm run typecheck</code>	Los ejemplos de TypeScript y contratos compartidos aún compilan.
<code>npm run capstones:evidence</code>	Los capítulos de capstone, razones de detención de runtime, assets de trace, reportes de eval y enlaces de scorecard coinciden.

Comando	Qué Demuestra
<code>npm run native-examples:validate</code>	Los archivos de ejemplo nativos de framework son sintácticamente válidos y existen los assets requeridos.
<code>npm run native-examples:smoke:langgraph</code>	Los slices nativos de LangGraph instalan dependencias opcionales y se ejecutan sin provider keys.
<code>npm run book:manifest:test</code>	Sidebar, manifiesto PDF, propiedad de capítulos y registro de capítulos generados son válidos.
<code>npm run book:visuals:verify</code>	Cada capítulo del manifiesto y la página principal tienen un diagrama Mermaid, imagen o referencia SVG.
<code>npm run book:quality</code>	Cobertura del manifiesto, capítulos generados, consistencia de título/H1, marcadores sin terminar, assets de diagramas, texto alternativo de diagramas y cobertura de Mermaid SVG son válidos.
<code>npm run book:build</code>	La compilación de autoría de VitePress, páginas generadas, diagramas y descargas son válidas después de pasar los quality gates.
<code>npm run site:build</code>	El sitio lector de Astro compila con assets públicos sincronizados e índice de búsqueda.
<code>npm run site:parity</code>	Las rutas publicadas y enlaces internos coinciden con el manifiesto del libro.
<code>npm run book:pdf</code>	El PDF de cortesía y la copia de despliegue pueden regenerarse.
<code>npm run book:epub</code>	El EPUB de cortesía y la copia de despliegue pueden regenerarse.

QA Visual

Inspecciona estas páginas en el sitio compilado antes del lanzamiento:

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/`

Verifica que los diagramas se rendericen, los encabezados se ajusten, los bloques de código sean legibles, las tablas no queden sin explicación y la navegación mantenga orientada a la persona lectora.

QA de URL Pública

Después del despliegue en GitHub Pages, inspecciona el sitio público con la ruta base de producción:

URL Pública	Revisión
<code>/Agentic-Systems-Patterns/</code>	La página principal se muestra y las acciones principales funcionan.
<code>/Agentic-Systems-Patterns/book/intro/</code>	La introducción carga desde el sitio desplegado.
<code>/Agentic-Systems-Patterns/book/publishing/how-to-read/</code>	Las rutas de lectura son accesibles.
<code>/Agentic-Systems-Patterns/book/pattern-selection/choosing-the-right-pattern/</code>	La página principal de selección de patterns carga.
<code>/Agentic-Systems-Patterns/book/hands-on-labs/</code>	El índice de labs carga y los enlaces de labs resuelven.
<code>/Agentic-Systems-Patterns/book/capstone-projects/</code>	El índice de capstones carga y los enlaces de artifacts resuelven.
<code>/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.pdf</code>	El PDF de cortesía se descarga.
<code>/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.epub</code>	El EPUB de cortesía se descarga.
<code>/Agentic-Systems-Patterns/pagefind/</code>	Existen los assets de búsqueda, la búsqueda arroja resultados y los filtros de nivel/tipo funcionan.

Las verificaciones de compilación local prueban el artifact desplegable. El QA de URL pública prueba la superficie de lectura desplegada.

QA de Descargas y Assets

Revisa los assets orientados al lector que hacen útil el libro en línea:

Asset	Evidencia de Lanzamiento
PDF de cortesía	<code>/releases/Agentic-Systems-Patterns.pdf</code> existe y refleja el contenido actual.
EPUB de cortesía	<code>/releases/Agentic-Systems-Patterns.epub</code> existe y refleja el contenido actual.
Paquetes fuente	Las descargas de pattern, lab, framework nativo y capstone resuelven desde el sitio compilado.
Plantillas	Worksheets, scorecards y listas de verificación de revisión resuelven bajo <code>/capstone-assets/templates/</code> .
Ejemplos completos	Ejemplo de ADR, evidencia de lab y ejemplos de preparación para producción resuelven bajo <code>/capstone-assets/templates/</code> .

Asset	Evidencia de Lanzamiento
Ejemplos de trace	Los archivos JSON de trace de capstone resuelven y coinciden con las referencias del capítulo.
Reportes de eval	Los archivos de reporte de eval de capstone resuelven y coinciden con las referencias del capítulo.
Evidence gate de capstone	<code>npm run capstones:evidence</code> pasa contra el texto del capítulo, salida de runtime, assets de trace, reportes de eval y enlaces de scorecard.
Diagramas	Los diagramas SVG se renderizan en las páginas de capítulos y no quedan huérfanos.
Metadata de búsqueda	Pagefind reporta filtros de capítulos y los resultados de búsqueda renderizados muestran metadata de sección, tipo y nivel.

Esta revisión importa porque el libro es un producto en línea. Un capítulo puede leerse bien y aun así fallar a las personas lectoras si sus descargas o artifacts de evidencia faltan.

Registro de Evidencia de Lanzamiento

Antes de publicar, registra:

```
version:
date:
commit:
commands passed:
visual pages checked:
download assets checked:
known limitations:
release owner:
rollback action:
```

Coloca el registro en las notas de lanzamiento, el release de GitHub o la descripción del PR de lanzamiento.

Usa el [registro de evidencia de lanzamiento](#) descargable cuando el release cambie contenido visible para el lector o el comportamiento de publicación. El release actual ya tiene un [registro de evidencia de pre-lanzamiento](#) completado; llena su sección pública de GitHub Pages después del despliegue.

Decisión de Lanzamiento

No publiques si alguna de estas condiciones es verdadera:

- Un comando requerido falla.
- Un core reader path contiene un enlace roto o una imagen faltante.
- Un lab apunta a código fuente que ya no existe.
- Un capstone afirma cobertura nativa de framework sin un ejemplo correspondiente o un alcance claro.

- El PDF o EPUB de cortesía está desactualizado respecto al contenido del sitio.
- Las notas de lanzamiento no indican qué cambió y cómo se verificó.

Publica solo cuando la evidencia de lanzamiento sea más sólida que la afirmación hecha a los lectores.

Notas de la versión

Estas notas resumen el estado actual de la versión de *Agentic Systems Patterns*. Úsalas junto con la [Release Readiness Checklist](#) antes de publicar.

Utiliza el [release evidence record](#) para capturar la salida real de comandos, verificaciones de URLs públicas, revisión de assets, límites conocidos y acción de rollback para una versión. El registro local actual completado es el [pre-launch release evidence for 2026-06-21](#).

Versión actual

Versión: `1.0.0`

Fecha de revisión: 2026-06-21

Tema de la versión: transformar el repositorio de un catálogo de patterns a una guía completa para diseñar, implementar, evaluar y operar agentic systems en diferentes frameworks.

Valor agregado para el lector

- Rutas claras para lectores primerizos, constructores, usuarios de labs, usuarios de capstone y usuarios de referencia.
- Capítulos de patterns con intención consistente, guía de uso/evitación, arquitectura, forma del sistema, protocolo, modos de falla, estrategia de eval, checklist de producción, enlaces a fuentes y descargas.
- Labs agnósticos de framework en Python y TypeScript.
- Cobertura de retrieval estilo LangChain/LangGraph, state graphs estilo LangGraph, runtime packaging estilo Mastra, transcript evaluation estilo AutoGen, flows estilo CrewAI, A2A, MCP y runtimes personalizados deterministas.
- Un track de mini-framework desde cero que explica qué empaquetan los agent frameworks internamente.
- Capstones orientados a producto para soporte de reembolsos, research RAG y workflows de entrega multi-agent.
- Un caso de estudio de soporte de reembolso que inicia en la Introducción y capítulos de fundamentos, y regresa en labs y capstones.
- Ejemplos más concretos de enseñanza de patterns para drafting de single-agent, refund gates con prompt chaining y scoring de evaluator-optimizer.
- Presupuestos de tiempo opcionales por ejercicio en Labs 01-13 para que los lectores puedan dividir el trabajo en bloques más cortos y revisables.
- Ejemplos avanzados más profundos para fallas en la configuración de frameworks, conversión de incidentes de aprobación de reembolso a eval, y límites de autoridad específicos de dominio.
- Ejemplos completos de preparación para producción para soporte de reembolso, research RAG y workflows de entrega multi-agent.
- Slices nativos de framework para ejemplos seleccionados de LangGraph, Mastra, CrewAI y AutoGen.

- Flujo de publicación listo para producción para el lector de GitHub Pages, PDF/EPUB de cortesía, bundles de fuentes generados, diagramas, metadatos y filtros de búsqueda Pagefind, verificaciones de paridad y validación de ejemplos nativos.

Evidencia de verificación

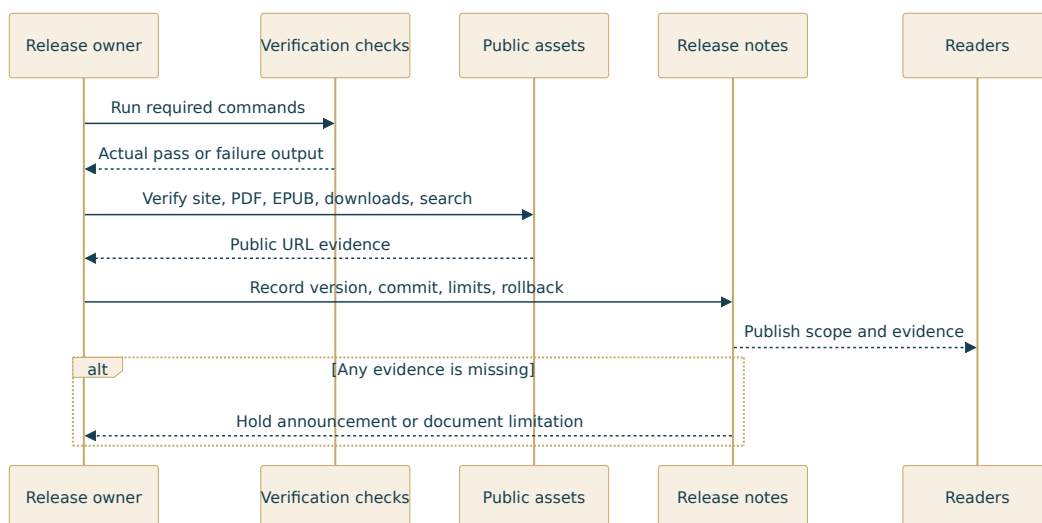
Antes de publicar, la versión debe pasar:

```
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
```

La versión no está lista si algún comando falla, si la cobertura visual retrocede, si la evidencia de capstone difiere de la salida del runtime, o si el PDF y EPUB de cortesía generados no se actualizan después de cambios en el contenido.

Registra los resultados reales de los comandos con la versión. La evidencia esperada no es suficiente para una etiqueta pública o anuncio.

Utiliza este flujo al convertir estas notas en una versión pública. Una nota de versión es válida solo cuando apunta a evidencia actual, assets públicos y una acción de rollback.



Plantilla de registro de evidencia

Para cada versión pública, registra:

Campo	Valor
Versión	
Fecha	
Commit	
Responsable de la versión	
Comandos aprobados	
URLs públicas verificadas	
Discussions verificadas	
Assets de descarga verificados	
Evidencia de capstone verificada	
Búsqueda verificada	
Filtros de búsqueda verificados	
PDF de cortesía verificado	
EPUB de cortesía verificado	
Limitaciones conocidas	
Acción de rollback	

Mantén este registro en el PR de la versión, el release de GitHub o en las notas de la versión anexadas. No reemplaces la evidencia real con una checklist planeada.

Verificación de assets para el lector

Antes de publicar, verifica estas rutas del sitio generado:

Asset	Ruta
Libro en línea	<code>/Agentic-Systems-Patterns/</code>
Discussions	<code>https://github.com/GTuritto/Agentic-Systems-Patterns/discussions</code>
PDF de cortesía	<code>/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.pdf</code>
EPUB de cortesía	<code>/Agentic-Systems-Patterns/releases/Agentic-Systems-Patterns.epub</code>
Plantillas	<code>/Agentic-Systems-Patterns/capstone-assets/templates/</code>

Asset	Ruta
Ejemplos de salida capturada	lab-and-capstone-command-output.txt
Ejemplos de preparación para producción completados	completed-production-readiness-examples.txt
Descargas	/Agentic-Systems-Patterns/downloads/
Índice y filtros de búsqueda	/Agentic-Systems-Patterns/pagefind/ con metadatos de sección, tipo, nivel y ruta del lector.

La versión no debe anunciarse hasta que el sitio de GitHub Pages, el canal de feedback en GitHub Discussions, el PDF de cortesía, el EPUB de cortesía, las descargas, las plantillas, los ejemplos de salida capturada y el índice de búsqueda de Pagefind estén todos generados desde el mismo contenido o verificados contra el mismo release.

Límites conocidos del alcance

- Los ejemplos son educativos y deterministas por defecto; las integraciones en vivo con proveedores de model requieren configuración local.
- Los slices nativos de framework son puntos de comparación para límites importantes, no tutoriales exhaustivos de cada feature del framework.
- El libro prioriza arquitectura, evidencia de producción y disciplina de revisión de diseño sobre la cobertura API por API de los frameworks.
- Los nombres históricos de patterns permanecen en la sección deprecada para que la terminología antigua pueda mapearse a la taxonomía actual.

Artifacts de publicación

- Salida del sitio: [site/dist](#)
- Artifact fuente PDF: [book/releases/Agentic-Systems-Patterns.pdf](#)
- Copia de despliegue PDF: [book/docs/public/releases/Agentic-Systems-Patterns.pdf](#)
- Artifact fuente EPUB: [book/releases/Agentic-Systems-Patterns.epub](#)
- Copia de despliegue EPUB: [book/docs/public/releases/Agentic-Systems-Patterns.epub](#)
- Descargas generadas: [book/docs/public/downloads/](#)

Resumen de la versión

Esta versión está lista cuando el lector puede responder cinco preguntas después de terminar la guía:

1. ¿Qué agentic pattern debo usar y por qué?
2. ¿Qué controla state, policy, tools, memory, approvals, traces y evals?
3. ¿Cómo ejecuto un ejemplo pequeño e identifico qué falta para producción?
4. ¿Cómo se mapea la misma arquitectura entre frameworks y lenguajes?
5. ¿Qué evidencia prueba que el sistema es lo suficientemente seguro para liberar?

Verificaciones posteriores a la publicación

Después del despliegue, abre la URL pública de GitHub Pages y verifica:

- que la página principal cargue;
- que la navegación por capítulos funcione;
- que la búsqueda abra, devuelva resultados ricos en metadatos y los filtros de nivel/tipo funcionen;
- que el PDF de cortesía se descargue;
- que el EPUB de cortesía se descargue;
- que al menos una worksheet, trace, eval report, ejemplo de salida capturada y bundle de fuentes se descarguen.

Si algún asset público falla, actualiza las notas de la versión con la limitación o revierte el anuncio.