

Facsímil 06

Construir y operar

Coleccionable completo · 10 capítulos

Índice

01	De prototipo a sistema operable
02	Arquitectura de runtime: API, colas, estado y contratos
03	Serving de modelos: workers, batching y capacidad
04	Observabilidad: logs, métricas, trazas y costes
05	Routing, fallback y presupuestos por tarea
06	EvalOps y gates de release
07	Cambios progresivos: shadow, canary y rollback
08	Handoffs operativos y revisión humana
09	Incidencias, postmortems y mejora continua
10	Runbooks y continuidad de operación

Capítulo 01: De prototipo a sistema operable

Qué deberías poder hacer al terminar

En una asignatura de ingeniería, este capítulo no debería evaluarse preguntando “¿qué es LLMOps?”. Eso sería demasiado fácil y demasiado poco útil. Lo que importa es si puedes transformar una capacidad de IA en un sistema revisable.

Al terminar, deberías poder hacer estas cinco cosas:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Distinguir demo, prototipo y sistema operable.	Explicas qué falta para pasar de una respuesta bonita a una capacidad mantenible.
Diseñar un manifest mínimo.	Nombras versiones de modelo, prompt, política, dataset, runtime y owner.
Modelar una run como máquina de estados.	Dices en qué estado está, qué transiciones son válidas y qué eventos la mueven.
Definir SLI, SLO y presupuesto de error.	Conviertes “que vaya bien” en números: calidad, latencia, coste y errores.
Escribir un release gate.	Bloqueas una versión que mejora una métrica pero rompe contrato, latencia, coste o rollback.

Este encuadre cambia el tono del facsímil: no basta con entender los nombres. Hay que saber diseñar la evidencia.

La pregunta que abre este facsímil

Hasta ahora hemos aprendido a reconocer piezas: modelos, APIs, RAG, embeddings, herramientas, agentes, memoria, permisos, trazas y evaluaciones. Eso ya es mucho. Pero todavía falta una pregunta incómoda: **¿qué hace que todo eso pueda vivir fuera de una demo?**

Una demo puede responder bien tres veces delante de una persona paciente. Un sistema operable debe responder muchas veces, con usuarios distintos, datos cambiantes, coste medido, versiones controladas, límites claros, trazas revisables y una forma de volver atrás si

una mejora aparente empeora el conjunto.

Este facsímil va de ese salto. No vamos a tratar producción como “subir algo a un servidor”. La vamos a tratar como una disciplina: construir sistemas de IA que puedan observarse, evaluarse, mantenerse, auditarse y mejorar sin que cada cambio sea una apuesta.

Qué no significa construir y operar

Construir y operar no significa envolver un prompt en una API y ponerle una URL bonita. Eso puede ser el primer envoltorio, pero no resuelve las preguntas importantes: qué versión contestó, qué contexto recibió, cuánto costó, qué ocurrió si falló, qué datos se guardaron, qué usuario quedó afectado y cómo sabemos si la versión nueva es mejor.

Tampoco significa montar una plataforma enorme desde el primer día. Un equipo pequeño puede operar bien con pocos artefactos: un manifest, un dataset de evaluación, trazas mínimas, configuración versionada, un gate de release y una decisión escrita. El problema no es empezar pequeño. El problema es empezar sin forma de medir ni revertir.

Y no significa que todo sea automático. Operar bien incluye decidir qué no debe automatizarse todavía, qué requiere revisión humana, qué se ejecuta en modo lectura, qué se prueba en sombra y qué se detiene cuando falta evidencia.

Qué sí significa operar un sistema de IA

Un sistema de IA está operado cuando podemos responder preguntas técnicas sin reconstruir la historia a mano:

PREGUNTA	RESPUESTA QUE DEBERÍA EXISTIR
¿Qué versión contestó?	Modelo, prompt, herramienta, política, dataset y código versionados.
¿Qué vio el modelo?	Contexto, documentos recuperados, mensajes relevantes y filtros aplicados.
¿Qué hizo el sistema?	Llamadas a modelo, tools, validadores, colas, gates y salidas.
¿Cuánto costó?	Tokens, tiempo, llamadas externas, coste total y coste por tarea aceptada.
¿Cómo falló?	Error clasificado, traza con spans, estado final y causa probable.
¿Cómo se corrige?	Cambio propuesto, eval de regresión, canary, rollback y owner.

Sculley y colaboradores llamaron la atención sobre una deuda técnica propia de los sistemas de aprendizaje automático: datos, configuraciones, dependencias y comportamiento pueden entrelazarse hasta hacer difícil entender qué cambió realmente.¹ Amershi y su equipo mostraron que construir sistemas de ML exige prácticas específicas de ingeniería porque el comportamiento no depende solo del código: también depende de datos, experimentos, métricas, pipelines y evaluación continua.²

En sistemas con LLMs añadimos otra capa: prompts, contexto, herramientas, proveedores, modelos cambiantes, costes por token, respuestas variables y decisiones de producto. Por eso necesitamos una forma de pensar más parecida a ingeniería de sistemas que a “probar prompts”.

Requisitos antes de hablar de arquitectura

Una costumbre sana en ingeniería es no dibujar arquitectura antes de saber qué problema debe aguantar. En IA esto se olvida con facilidad porque el modelo produce algo visible muy pronto. Pero una respuesta visible no equivale a requisitos entendidos.

Para un sistema de IA, separaría los requisitos en tres grupos:

TIPO DE REQUISITO	PREGUNTA	EJEMPLO EN UN ASISTENTE DE SOPORTE
Funcional	¿Qué debe hacer el sistema?	Responder dudas de matrícula con citas o abrir revisión si falta evidencia.
No funcional	¿Con qué calidad operativa debe hacerlo?	p95 menor de 2,5 s, coste menor de 0,04 euros por respuesta aceptada, salida JSON válida.
De cambio	¿Cómo se modifica sin perder control?	Prompt, modelo, retrieval y política versionados con eval previa y rollback probado.

En software clásico solemos distinguir requisitos funcionales y no funcionales. En IA añadiría siempre los requisitos de cambio, porque el sistema vive de ajustes: cambia el prompt, cambia el modelo, cambia el corpus, cambia el proveedor, cambia la política de abstención y cambia la evaluación. Si no modelas el cambio, el sistema parece estable hasta que alguien pregunta qué versión produjo una respuesta concreta.

Un requisito bien escrito no dice “que responda bien”. Dice algo como:

Para preguntas cubiertas por normativa vigente, el asistente debe devolver una respuesta con al menos una cita recuperable, schema válido, p95 menor de 2,5 s y coste por tarea aceptada menor de 0,04 euros. Si no hay evidencia suficiente, debe abstenerse y proponer siguiente paso.

Esta frase ya contiene producto, datos, calidad, latencia, coste, contrato y fallback. Es mucho más aburrida que una demo, y por eso mismo es más útil.

Las piezas de un sistema operable

Antes de cualquier fórmula conviene un inventario. Operar un sistema de IA no es una sola cosa: es sostener seis piezas a la vez, y si falta una, la operación cojea justo por ahí. Esto no es una ecuación de la literatura (no se «suman» un control plane y un runtime), así que lo presentamos como lo que es, una lista de ingeniería:

PIEZA	QUÉ APORTA	EJEMPLO
Control plane	Dónde se decide y se versiona el comportamiento.	Registro de modelo, prompt, flags, políticas y límites.
Runtime	Dónde se ejecuta cada petición.	API, cola, llamada al modelo, tools, timeouts y reintentos.
Observabilidad	Cómo se ve lo que pasó.	Trazas, métricas, logs, coste y errores por ejecución.
Evaluación	Cómo se sabe si funciona.	Dataset de regresión, métricas y revisión de casos difíciles.
Gates de release	Cuándo se permite publicar.	Reglas para pasar de local a sombra, canary y producción.
Versionado y cambio	Cómo se vuelve atrás con calma.	SemVer, manifest, rollback, owner y decisión escrita. ³

En palabras: la lista no mide nada físicamente; sirve para comprobar qué pieza falta. Si tienes runtime sin observabilidad, no puedes depurar. Si tienes evaluación sin versionado, no puedes comparar. Si tienes control plane sin rollback, puedes cambiar rápido, pero no volver con calma.

La decisión que sí se convierte en código es el **gate de release**: una puerta de calidad clásica, una conjunción de criterios de aceptación que solo deja publicar si se cumplen todos a la vez.⁴

$$release\_ok = calidad\_ok \wedge latencia\_ok \wedge coste\_ok \wedge contrato\_ok \wedge rollback\_ok$$

TÉRMINO	QUÉ COMPRUEBA	EJEMPLO
<i>calidad_ok</i>	La salida cumple la rúbrica o métrica mínima.	Al menos 44 de 50 casos pasan la eval privada.
<i>latencia_ok</i>	La respuesta cabe en el tiempo acordado.	p95 menor o igual a 2,5 segundos.
<i>coste_ok</i>	El coste por tarea aceptada está dentro del presupuesto.	Menos de 0,04 euros por caso válido.
<i>contrato_ok</i>	La salida respeta schema, citas, permisos y abstención.	JSON válido con fuente verificable o respuesta de no evidencia.
<i>rollback_ok</i>	Existe forma probada de volver a la versión anterior.	Feature flag que restaura <code>prompt_v12</code> y <code>model_a</code> .

En palabras: un cambio solo se publica si la calidad, la latencia, el coste, el contrato y la recuperación están todos en verde. Basta que uno falle para no publicar, y por eso este gate sí debe convertirse en código, aunque empiece siendo pequeño.

SLI, SLO y presupuesto de error

Estas tres siglas vienen del mundo de la fiabilidad de servicios. Conviene traducirlas despacio porque, si solo damos el acrónimo, parecen burocracia. En realidad son una forma muy práctica de convertir “que vaya bien” en números que el equipo pueda discutir.

SIGLA	NOMBRE COMPLETO	TRADUCCIÓN ÚTIL	PREGUNTA QUE RESPONDE
SLI	<i>Service Level Indicator</i>	Indicador de nivel de servicio.	¿Qué estamos midiendo realmente?
SLO	<i>Service Level Objective</i>	Objetivo de nivel de servicio.	¿Qué valor consideramos aceptable?
Presupuesto de error	<i>Error budget</i>	Margen de fallo permitido.	¿Cuánto podemos fallar antes de frenar cambios?

El orden importa:

1. Primero defines el **SLI**, porque sin indicador no sabes qué medir.
2. Después defines el **SLO**, porque necesitas decidir qué valor mínimo aceptas.
3. Por último calculas el **presupuesto de error**, porque quieres saber cuánto margen tienes antes de investigar, pausar cambios o volver a una versión anterior.

Un SLO no es una frase de marketing. Es un objetivo interno medible. Para escribirlo necesitamos primero un SLI, que es el indicador que medimos.

Ejemplo en lenguaje llano:

CONCEPTO	VERSIÓN HUMANA	VERSIÓN MEDIBLE
SLI	"De todas las runs, ¿cuántas salen aceptables?"	<code>runs_aceptadas / runs_totales</code>
SLO	"Queremos que casi todas salgan aceptables."	<code>SLI_calidad >= 0.97</code>
Presupuesto de error	"Aceptamos un margen pequeño antes de parar."	<code>1 - 0.97 = 0.03</code>

Por ejemplo:

$$SLI_{calidad} = \frac{runs\_aceptadas}{runs\_totales}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$SLI_{calidad}$	Proporción de ejecuciones aceptadas.	9.650 de 10.000 runs pasan calidad y contrato.
$runs_{aceptadas}$	Runs que cumplen rúbrica, contrato y límites.	9.650
$runs_{totales}$	Runs evaluadas en una ventana temporal.	10.000

Si el SLO de calidad es 97%, entonces:

$$presupuesto\_de\_error = 1 - SLO$$

TÉRMINO	SIGNIFICADO	EJEMPLO
SLO	Objetivo que prometemos internamente.	0,97
$1 - SLO$	Margen de error tolerado.	0,03
Presupuesto en 10.000 runs	Runs que pueden fallar antes de parar cambios.	300

Esto no significa que 300 fallos “den igual”. Significa que el equipo decide de antemano cuánto margen tiene antes de congelar cambios, investigar o volver a una versión anterior. Sin presupuesto de error, cada fallo parece una anécdota o una crisis. Con presupuesto de error, las conversaciones se vuelven operativas.

En una ventana de 10.000 runs, el cálculo sería:

PASO	CÁLCULO	RESULTADO
Objetivo	$SLO = 0,97$	97% de runs aceptables
Margen	$1 - 0,97$	0,03
Presupuesto	$10.000 \times 0,03$	300 runs no aceptadas

Ese presupuesto se gasta. Si en dos días ya llevas 250 runs no aceptadas, quizá no conviene desplegar un prompt nuevo aunque la demo parezca mejor. Si en toda la semana llevas 20, tienes más margen para experimentar. Esa es la gracia: el presupuesto de error conecta calidad, operación y velocidad de cambio.

Para sistemas de IA suelo definir varios SLI a la vez:

SLI	FÓRMULA O MEDICIÓN	QUÉ CAPTURA
Calidad aceptada	<code>runs_aceptadas / runs_totales</code>	Si la salida cumple rúbrica, contrato y evidencia.
Latencia p95	Percentil 95 de duración total.	Si la cola de usuarios reales espera demasiado.
Coste por tarea aceptada	<code>coste_total / runs_aceptadas</code>	Si el sistema es sostenible, incluyendo reintentos.
Abstención correcta	<code>abstenciones_correctas / casos_sin_evidencia</code>	Si el sistema sabe no responder cuando falta base.
Error de contrato	<code>salidas_invalidas / runs_totales</code>	Si el JSON, las citas o el formato rompen integraciones.

El punto docente es importante: una eval offline mide comportamiento con casos preparados; un SLO mide operación en una ventana concreta. Los dos se necesitan. La eval evita publicar una mala versión. El SLO detecta que el mundo cambió después.

Qué significa, en números, estar disponible. «El sistema está disponible» suena absoluto, pero la disponibilidad es una proporción de tiempo, y se calcula con dos cantidades que cualquier equipo de operación debería conocer: cuánto aguanta el sistema entre fallos y cuánto tarda en recuperarse.⁵

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
A	Disponibilidad: fracción de tiempo en que el sistema sirve.	0,999 (tres nueves).
MTTF	<i>Mean Time To Failure</i> : tiempo medio en funcionamiento entre fallos.	720 horas.
MTTR	<i>Mean Time To Repair</i> : tiempo medio que tardas en recuperarte.	0,72 horas (43 min).

En palabras: disponibilidad no es «que casi nunca se cae»; es cuánto aguanta dividido entre cuánto aguanta más lo que tardas en levantarlo. La consecuencia operativa es contundente: recuperar el doble de rápido sube la disponibilidad igual que fallar la mitad de veces, y casi siempre es más barato invertir en rollback y runbooks (bajar MTTR) que en evitar todo fallo (subir MTTF).

Por qué la disponibilidad de extremo a extremo es peor que la de cada pieza. Un sistema de IA no se sostiene solo: depende del proveedor del modelo, de la base vectorial, de tu API, de la cola y del almacenamiento. Si una petición necesita que todas esas piezas funcionen a la vez, su disponibilidad combinada es el producto de las individuales, nunca el mejor ni la media.⁶

$$A_{\text{sistema}} = \prod_{i=1}^n A_i$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
A_{sistema}	Disponibilidad de la petición completa.	$0,999^5 \approx 0,995$.
A_i	Disponibilidad de cada dependencia en la cadena.	API, modelo, base vectorial, cola, BBDD: 0,999 cada una.
n	Número de dependencias que deben funcionar a la vez.	5

En palabras: cinco piezas con tres nueves cada una no dan tres nueves al usuario, dan dos y medio: pasas de 8,7 horas de caída al año a más de 40. Por eso operar bien no es solo cuidar cada servicio, sino reducir cuántos tienen que estar vivos a la vez (cachés, modos degradados, fallback) para acortar la cadena en serie.

Del objetivo de disponibilidad al tiempo de caída que te permites. Un SLO de disponibilidad parece abstracto hasta que lo traduces a minutos. La cuenta es directa y conviene tenerla memorizada, porque desmonta promesas: prometer «cinco nueves» en un sistema que depende de terceros suele ser prometer lo imposible.

$$D = (1 - A) \cdot T$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
D	Tiempo de caída permitido en la ventana.	8,77 horas al año.
$1 - A$	Indisponibilidad: el complemento del objetivo.	$1 - 0,999 = 0,001$.
T	Duración de la ventana.	8.760 horas (un año).

En palabras: este es el presupuesto de error del capítulo, pero contado en tiempo en lugar de en runs. El 99,9% son 8,77 horas al año; el 99,99%, 52,6 minutos; el 99,999%, 5,26 minutos. Saber esto antes de firmar un SLO evita comprometerse con cifras que ninguna

cadena de proveedores externos puede sostener.

Cuánta confianza merece un SLI medido con pocas runs. Cuando dices «el SLI de calidad es 0,96», esa cifra es una estimación con incertidumbre, no una verdad. Con 50 runs, 48 aciertos también dan 0,96, pero el intervalo real es ancho. El intervalo de puntuación de Wilson da el rango creíble de una proporción y se comporta bien incluso con pocas muestras o proporciones cercanas a 1, donde la fórmula ingenua falla.⁷

$$\tilde{p} = \frac{\hat{p} + \frac{z^2}{2n}}{1 + \frac{z^2}{n}} \pm \frac{z}{1 + \frac{z^2}{n}} \sqrt{\frac{\hat{p}(1 - \hat{p})}{n} + \frac{z^2}{4n^2}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\hat{p}$	Proporción observada de runs aceptadas.	48/50 = 0,96.
n	Número de runs evaluadas.	50
z	Cuantil normal según la confianza buscada.	1,96 para el 95%.
$\tilde{p}$	Centro y semiancho del intervalo creíble.	Aproximadamente [0,865; 0,990].

En palabras: con 50 runs, un SLI «de 0,96» es en realidad «entre 0,87 y 0,99 con un 95% de confianza». No puedes declarar que cumples un SLO de 0,97 con esa muestra: cabe dentro del intervalo, pero también cabe 0,88. La lección operativa es que un SLI necesita una ventana con suficientes runs antes de servir para bloquear o promover; con pocas, el número engaña.

Fecha de corte y alcance

Fecha de corte: 27 de mayo de 2026.

Fuentes consultadas en este facsímil hasta este punto: prácticas de ingeniería de ML, TFX, OpenTelemetry, W3C Trace Context, Harness Engineering, AGENTS.md, vocabulario normativo de RFC 2119/RFC 8174 y versionado semántico.

Lo estable es el método: versionar artefactos, ejecutar evals, observar runs, controlar coste, diseñar límites, introducir cambios gradualmente y preparar rollback. Lo cambiante son productos concretos, nombres de runtimes, modelos, precios, dashboards y SDKs.

Baylor y colaboradores describieron TFX como una plataforma de producción donde no basta entrenar: hay que validar datos, validar modelos, servirlos y monitorizarlos dentro de un pipeline.⁸ Esa idea se mantiene aunque trabajemos con LLMs y no con un clasificador clásico: la capacidad del modelo es solo una pieza de un sistema mayor.

El contrato operativo de una run

Aquí “contrato” no significa contrato legal. Significa **acuerdo técnico explícito**: qué entra, qué sale, qué estados existen, qué errores son posibles y qué garantías mínimas ofrece el sistema. Es la diferencia entre “esto suele responder así” y “esto debe cumplir estas condiciones para considerarse válido”.

Un contrato en software funciona como una promesa verificable entre partes:

PARTE	QUÉ PROMETE	QUÉ PUEDE COMPROBARSE
Quien llama	Enviar datos con una forma válida.	Campos obligatorios, tipos, tamaño y permisos.
El sistema	Procesar bajo reglas conocidas.	Estado, límites, trazas, presupuesto y errores tipados.
Quien consume la salida	Recibir algo estable.	Schema de respuesta, campos esperados y significado de cada estado.

Ejemplo sencillo: si una API promete devolver siempre `answer` , `sources` , `confidence` y `needs_review` , eso es parte del contrato. Si a veces devuelve texto libre, a veces JSON y a veces un campo llamado `respuesta` , no hay contrato estable; hay una costumbre frágil.

En una run de IA el contrato es todavía más importante porque el modelo puede generar salidas variables. El contrato no elimina toda incertidumbre, pero pone una frontera: si la salida no cumple estructura, citas, límites o política, no se entrega como resultado válido.

Llamaremos *run* a una ejecución completa: llega una petición, el sistema decide ruta, llama modelos o herramientas, valida salida y termina con respuesta, abstención, error recuperable o revisión humana.

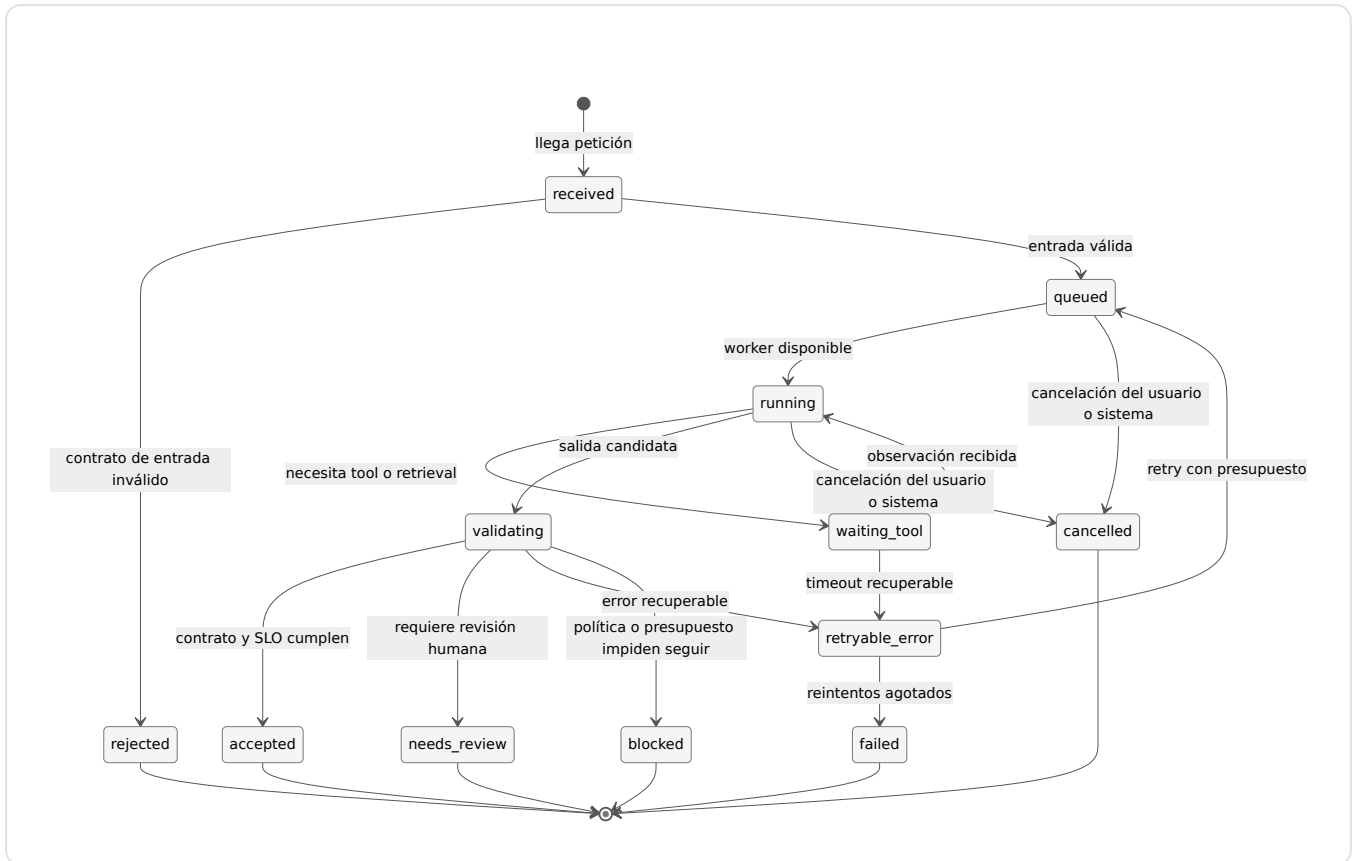
Para que una run sea operable, no basta con guardar el texto final. Debe dejar un contrato mínimo:

CAMPO	QUÉ GUARDA	POR QUÉ IMPORTA
<code>run_id</code>	Identificador único de la ejecución.	Permite buscar la historia completa.
<code>trace_id</code>	Identificador de traza que viaja entre servicios.	Une API, cola, modelo, tool y validadores.
<code>provider_request_id</code>	Identificador devuelto por el proveedor, si existe.	Permite cruzar tu traza con soporte o dashboard externo.
<code>input_hash</code>	Huella de la entrada, no necesariamente el texto completo.	Permite deduplicar sin exponer datos de más.
<code>model_version</code>	Modelo o proveedor usado.	Explica diferencias de comportamiento.
<code>prompt_version</code>	Plantilla e instrucciones usadas.	Permite rollback de prompts.
<code>context_manifest</code>	Documentos, memoria o retrieval inyectado.	Explica qué evidencia vio el modelo.
<code>policy_version</code>	Reglas de permisos, abstención y límites.	Evita decisiones invisibles.
<code>budget</code>	Tokens, coste, pasos, tiempo y reintentos permitidos.	Evita que una tarea pequeña se coma el sistema.
<code>output_contract</code>	Schema, citas, formato y validadores.	Hace comprobable la salida.
<code>decision</code>	<code>accepted</code> , <code>needs_review</code> , <code>retryable_error</code> o <code>blocked</code> .	Convierte texto en estado operativo.

OpenTelemetry define trazas como conjuntos de spans que representan unidades de trabajo, y su API permite crear spans, añadir atributos y propagar contexto.⁹ W3C Trace Context estandariza cómo llevar identificadores de traza entre servicios mediante cabeceras como `traceparent` .¹⁰ En castellano llano: si una petición atraviesa tres servicios, no quieres tres historias separadas. Quieres una sola historia con capítulos.

La run como máquina de estados

Si una run no tiene estados explícitos, acaba teniendo estados implícitos escondidos en logs, excepciones, mensajes de cola y flags sueltos. Eso dificulta depurar y enseñar el sistema. Para ingeniería, una run debe poder dibujarse.



La tabla de transiciones obliga a hablar con precisión:

ESTADO	QUÉ SIGNIFICA	EVENTO QUE LO MUEVE	DATO QUE DEBERÍA QUEDAR
received	La API recibió una petición.	Validación de entrada.	run_id , input_hash , usuario o tenant.
queued	La tarea espera ejecución.	Worker disponible o cancelación.	Tiempo en cola y prioridad.
running	El sistema está decidiendo o generando.	Tool, salida candidata o error.	Modelo, prompt, presupuesto consumido.
waiting_tool	Hay una dependencia externa pendiente.	Respuesta, timeout o error recuperable.	Tool, argumentos, timeout, intento.
validating	Hay salida candidata pendiente de contrato.	Validación pasa, falla o exige revisión.	Schema, citas, coste, latencia, rúbrica.
accepted	La salida puede entregarse.	Fin.	Output, versión, métricas finales.
needs_review	Una persona debe revisar.	Fin de la parte automática.	Motivo y datos mínimos para revisar.
retryable_error	Puede intentarse otra vez con cuidado.	Retry o fallo definitivo.	Tipo de error, contador, espera.
blocked	No debe continuar por política o presupuesto.	Fin.	Regla que bloqueó y siguiente paso.

El detalle clave es que cada transición tiene causa. No basta con saber que terminó mal; queremos saber si terminó mal por entrada inválida, timeout, presupuesto agotado, contrato roto, falta de evidencia o revisión pendiente.

Taxonomía de fallos que sí ayuda a depurar

Cuando todo se llama “fallo del modelo”, nadie aprende. Una taxonomía útil separa dónde se rompió el sistema:

TIPO DE FALLO	SÍNTOMA VISIBLE	CAUSA PROBABLE	PRIMERA PREGUNTA DE DEPURACIÓN
Entrada inválida	La API rechaza antes de llamar al modelo.	Falta campo, tipo incorrecto o tamaño excesivo.	¿El contrato de entrada está documentado y validado?
Retrieval insuficiente	Respuesta se abstiene o cita poco.	No hay documentos, chunking malo o filtro demasiado estrecho.	¿El documento esperado aparece en top-k?
Contexto contaminado	La respuesta mezcla asuntos no relevantes.	Se inyectó demasiado contexto o memoria imprecisa.	¿Qué fragmentos vio exactamente el modelo?
Salida fuera de contrato	JSON inválido, campos extra o cita ausente.	Schema débil, prompt ambiguo o postproceso incompleto.	¿El validador bloquea antes de entregar?
Latencia excesiva	p95 o p99 se disparan.	Cola, proveedor lento, contexto largo o tool pesada.	¿Dónde está el span más largo?
Coste excesivo	Buena calidad pero factura alta.	Reintentos, modelo caro, top-k alto o salida larga.	¿Cuál es el coste por tarea aceptada?
Estado incoherente	La traza dice una cosa y la base de datos otra.	Escrituras parciales o falta de idempotencia.	¿Qué se guarda antes y después de cada efecto?
Cambio no trazable	No sabemos qué versión falló.	Modelo, prompt o política sin manifest.	¿Existe manifest por run?

Esta tabla es una herramienta docente. Obliga a dejar de hablar de “la IA falla” y empezar a localizar subsistemas: entrada, retrieval, contexto, modelo, contrato, runtime, estado, coste o cambio.

Dónde mirar si el error viene del proveedor

Cuando el sistema usa OpenAI, Anthropic, Gemini, Bedrock u otro proveedor, el primer impulso suele ser abrir la página de estado y esperar una respuesta tranquilizadora. Está bien mirarla, pero no basta. Una página de estado te dice si hay un problema visible a nivel de servicio; tu traza te dice si **tu** petición falló por entrada inválida, cuota, permisos, timeout, límite de tamaño, modelo no disponible, región, credenciales o contrato de salida.

La regla práctica para diagnosticar un fallo cuando interviene un proveedor externo no es una fórmula, es una rutina de depuración: cruzar cuatro fuentes, ninguna suficiente por sí sola. Cada una aporta una pieza y, sobre todo, descarta una confusión frecuente:

PIEZA	QUÉ APORTA	QUÉ NO APORTA
Traza local	Qué ruta tomó tu sistema, qué prompt/modelo/contexto usó y cuánto tardó.	No demuestra por sí sola que el proveedor tenga una incidencia general.
Error del proveedor	Código HTTP, tipo de error, mensaje y límites aplicados.	No explica tu lógica de negocio ni tus validadores.
Dashboard del proveedor	Uso, facturación, límites, proyecto, claves o cuota según plataforma.	No sustituye tus logs ni tus evals.
Estado del servicio	Incidencias o mantenimiento publicados.	Puede ser agregado, tardar en reflejar casos concretos o no distinguir tu modelo exacto.

La tabla operativa sería esta:

SI USAS. ..	GUARDA SIEMPRE EN TU TRAZA	DÓNDE MIRAR EN EL PROVEEDOR	CÓMO INTERPRETARLO
OpenAI API	x-request-id , X-Client-Request-Id si lo envías, modelo, endpoint, HTTP status, x-ratelimit-* , openai-processing-ms , tokens y timestamp UTC.	Error codes , API reference: debugging requests , API Dashboard y status.openai.com .	Si ves 400, revisa payload y schema. Si ves 401/403, credenciales, organización, proyecto o permisos. Si ves 429, mira límites y ritmo. Si ves 5xx, reintentar con backoff y cruza con estado del servicio. OpenAI recomienda registrar request IDs para depuración. ¹¹
Anthropic / Claude API	request-id , request_id del cuerpo si aparece, error.type , error.message , modelo, anthropic-version , status, tokens y timestamp UTC.	Claude API errors , Claude Console y status.claude.com .	invalid_request_error apunta a formato o contenido; authentication_error a clave; permission_error a permisos; request_too_large a tamaño; rate_limit_error a límite; timeout_error , api_error u overloaded_error suelen exigir retry controlado y consulta de estado. Anthropic documenta que cada respuesta incluye un identificador de petición útil para soporte. ¹²
Gemini API	HTTP code, status como INVALID_ARGUMENT , RESOURCE_EXHAUSTED o UNAVAILABLE , mensaje, modelo, proyecto, región si aplica, cuota y timestamp UTC.	Gemini API troubleshooting , AI Studio status y consola del proyecto si usas Google Cloud.	INVALID_ARGUMENT suele ser petición mal formada; RESOURCE_EXHAUSTED suele apuntar a cuota o ritmo; UNAVAILABLE o 5xx requieren retry y comprobación de estado. La guía oficial separa problemas del backend de la API y de los SDKs cliente. ¹³
Amazon Bedrock	x-amzn-requestid , región, modelId , InvokeModel o endpoint usado, excepción AWS, status HTTP, cuenta/rol y timestamp UTC.	Bedrock API error troubleshooting , CloudWatch/CloudTrail si lo tienes activado y AWS Health Dashboard .	AccessDeniedException apunta a IAM; ValidationException a entrada; ThrottlingException o ServiceQuotaExceededException a cuota; ModelTimeoutException a tiempo de proceso; ServiceUnavailableException a disponibilidad. En Bedrock, la región y los permisos importan tanto como el modelo. ¹⁴
Proveedor agregado o router propio	ID interno, proveedor final, modelo final, ruta elegida, error original si se conserva, retry, fallback y coste.	Dashboard del agregador, página de estado del proveedor final y tus trazas.	Si el router oculta el error original, estás ciego. Exige conservar upstream_provider , upstream_model , upstream_status y upstream_request_id cuando exista.
Modelo local	ID de run, modelo exacto, quant , tamaño	Logs del proceso	Aquí no hay “estado del proveedor” que te salve. Si falla, mira memoria, colas, timeouts, modelo

SI USAS. ..	GUARDA SIEMPRE EN TU TRAZA	DÓNDE MIRAR EN EL PROVEEDOR	CÓMO INTERPRETARLO
con Ollama, vLLM, SGLang o similar	de contexto, uso de KV cache, GPU/CPU, cola, memoria y logs del runtime.	métricas del servidor, dashboard propio y pruebas sintéticas.	cargado, formato de pesos y presión de concurrencia.

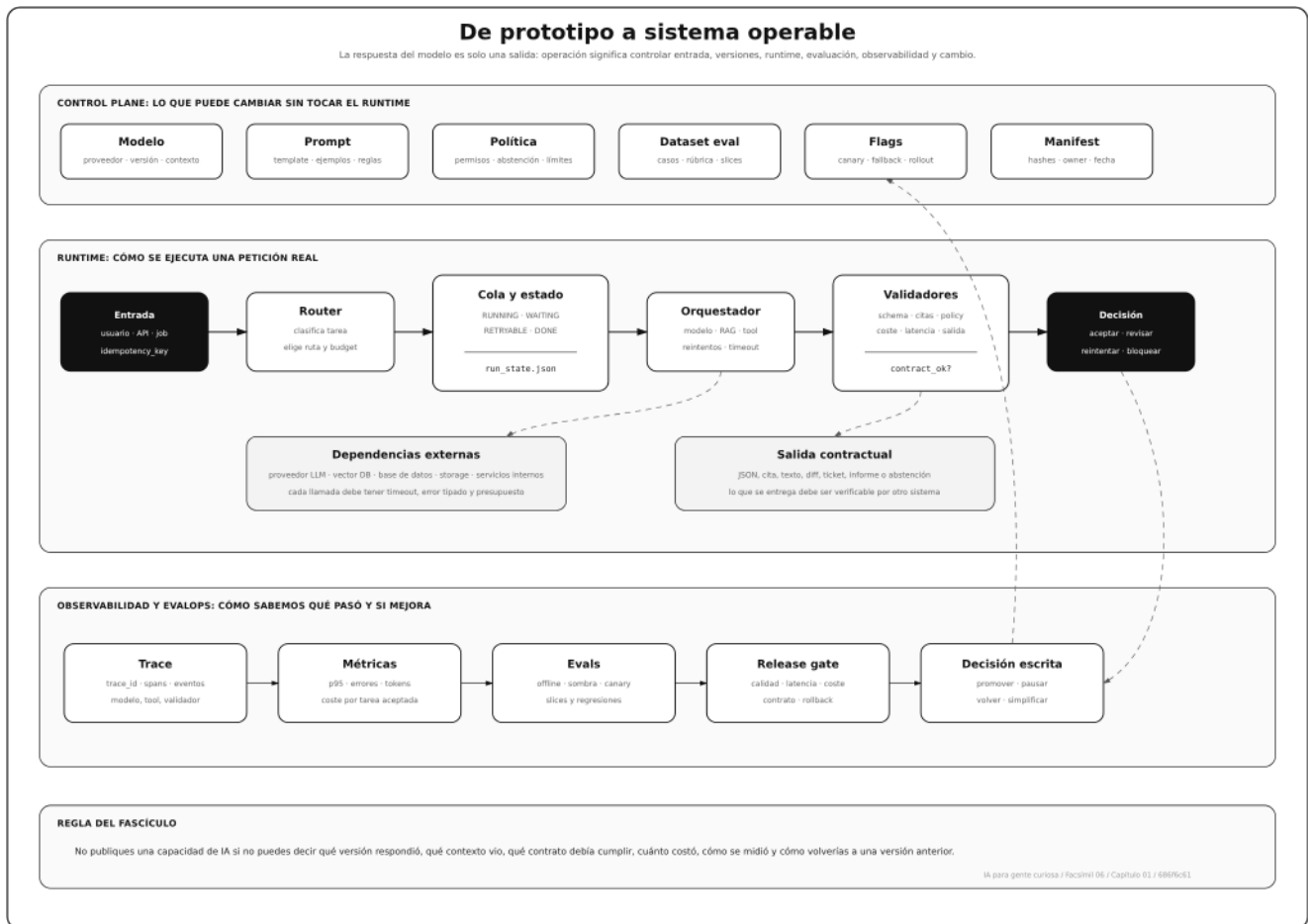
OpenAI documenta códigos de error como autenticación, permisos, límites, cuota y errores internos, y recomienda consultar la página de estado si aparece un error interno persistente.¹⁵ Su página de estado separa componentes como APIs y muestra disponibilidad agregada, con la advertencia de que la disponibilidad de un cliente concreto puede variar por tier, modelo y funcionalidad.¹⁶ Claude mantiene una página de estado separada por componentes como Claude API, Console y Claude Code.¹⁷ AWS documenta que el AWS Health Dashboard permite revisar el estado de servicios de AWS desde una página pública de salud.¹⁸

Lo que yo exigiría a cualquier equipo es un pequeño “paquete de depuración” por error:

```
run_id: run_20260527_00142
trace_id: 4bf92f3577b34da6a3ce929d0e0e4736
provider: openai
provider_request_id: req_...
model: modelo_fijado_por_manifest
endpoint: /responses
http_status: 429
provider_error_type: rate_limit
timestamp_utc: 2026-05-27T14:12:08Z
latency_ms: 1830
tokens_in: 1840
tokens_out: 0
retry_count: 2
decision: retryable_error
payload_hash: sha256:...
```

Y una regla de higiene: al pedir ayuda al proveedor, comparte IDs, timestamps, modelo, endpoint, código de error y cabeceras relevantes. No pegues datos sensibles ni prompts completos si no hace falta. Tu sistema debe poder reproducir el contexto técnico sin exponer más información de la necesaria.

Anatomía visual de un sistema operable



Este diagrama tiene una intención: separar piezas que suelen mezclarse. El control plane decide qué configuración existe. El runtime ejecuta una petición. La observabilidad permite reconstruirla. EvalOps decide si un cambio merece avanzar. Rollback evita que una versión mala se convierta en una semana de improvisación.

Cómo se ve en un proyecto real

Imagina un asistente interno para soporte universitario. Responde preguntas sobre matrícula, plazos, convalidaciones y documentación. En el facsímil 4 podríamos haberlo montado como RAG: buscar documentos, generar respuesta con citas y abstenerse si no hay evidencia. En el facsímil 5 podríamos haber añadido una tool para abrir un ticket o pedir revisión. Aquí preguntamos otra cosa: **¿cómo se opera cada lunes?**

El equipo necesita decidir:

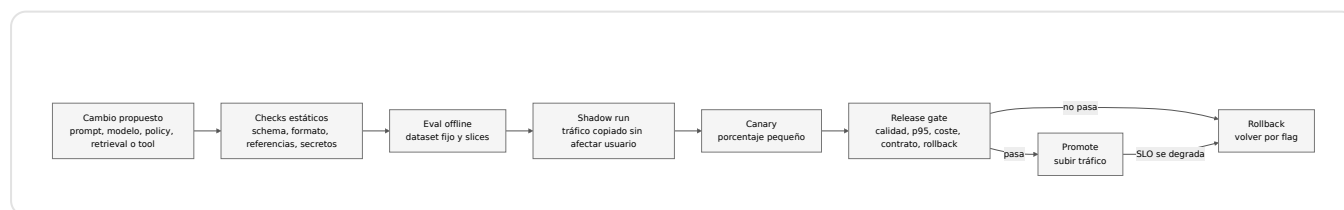
DECISIÓN	PREGUNTA CONCRETA	MALA SEÑAL	BUENA SEÑAL
Versión	¿Qué cambió hoy?	“Hemos tocado el prompt un poco”.	prompt:v1.8.2 , retriever:v3 , policy:v4 .
Calidad	¿Qué casos cubre la eval?	Solo preguntas fáciles.	Casos frecuentes, ambiguos, sin evidencia y con documentos contradictorios.
Latencia	¿Qué espera el usuario?	Media rápida, p99 invisible.	p50, p95, p99 y tasa de timeout por ruta.
Coste	¿Cuánto cuesta resolver una tarea válida?	Precio por token sin reintentos.	Coste por respuesta aceptada, incluyendo retrieval, tools y revisión.
Fallback	¿Qué pasa si no hay evidencia?	Inventar una respuesta amable.	Abstención con siguiente paso y trazabilidad.
Cambio	¿Cómo se despliega?	Todo el tráfico al nuevo prompt.	Sombra, canary, gate, owner y rollback.

El patrón se repite en banca, salud, educación, SaaS, soporte, legal, programación o administración pública: operar IA es convertir capacidad variable en comportamiento medible.

Pipeline CI/CD para sistemas de IA

En software clásico, CI/CD suele significar tests, build y despliegue. En IA añadimos artefactos que también cambian comportamiento: prompt, modelo, política, dataset, retrieval, tool schema y configuración de runtime.

El pipeline mínimo que pediría en clase sería este:



Lo importante no es que todos los equipos tengan esta cadena completa el primer día. Lo importante es saber qué parte falta. Si no hay eval offline, el canary se convierte en experimento con usuarios. Si no hay shadow run, descubres diferencias cuando ya afectan. Si no hay rollback, cada despliegue es una promesa de que todo irá bien.

FASE	QUÉ PRUEBA	QUÉ NO PRUEBA
Checks estáticos	Que el cambio está bien formado.	Que responde bien.
Eval offline	Que no rompe casos conocidos.	Que aguanta tráfico real.
Shadow run	Que se comporta con entradas reales sin impactar.	Que los usuarios aceptan la salida.
Canary	Que una fracción pequeña aguanta coste, latencia y calidad.	Que todas las colas y segmentos están cubiertos.
Promote	Que el cambio merece subir tráfico.	Que ya no hay que vigilarlo.
Rollback	Que podemos volver atrás.	Que entendimos la causa raíz.

Lo que un ingeniero informático debería exigir

Si este capítulo se convirtiera en una práctica de ingeniería del software, pediría estos artefactos:

ARTEFACTO	QUÉ CONTIENE	CÓMO SE REvisa
<code>manifest.yaml</code>	Versiones de modelo, prompt, política, dataset y runtime.	Diff revisable en PR.
<code>eval_dataset.jsonl</code>	Casos de entrada, expected, rúbrica y segmento.	Cobertura por tipo de caso.
<code>run_trace.jsonl</code>	Eventos de ejecución con <code>trace_id</code> , tiempos y atributos.	Permite reconstruir fallos.
<code>release_gate.py</code>	Código que decide si una versión pasa.	Tests unitarios y umbrales claros.
<code>rollback.md</code>	Cómo volver a la versión anterior.	Probado antes del lanzamiento.
<code>decision.md</code>	Por qué se promueve, pausa o simplifica.	Firmado por owner técnico o de producto.

El formato AGENTS.md propone instrucciones de proyecto para agentes de código: comandos, estructura, convenciones y reglas específicas del repositorio.¹⁹ Fowler lo formula desde otra esquina con *harness engineering*: el entorno alrededor del agente importa tanto como el modelo, porque aporta instrucciones, contexto, herramientas, verificación y límites.²⁰

La idea que nos llevamos al facsímil 6 es esta: **la ingeniería no empieza cuando el modelo falla; empieza antes, diseñando cómo sabremos que falla.**

En el día a día

El salto de prototipo a sistema operable no es teórico: aparece en decisiones concretas que un equipo toma cada semana. La primera es la del despliegue. Alguien ha mejorado el prompt y la calidad media sube dos puntos en la demo; la pregunta operable no es «¿es mejor?», sino «¿pasa el gate de release?», es decir, ¿mantiene la latencia p95, el coste por tarea aceptada, el contrato de salida y la posibilidad de volver atrás? Sin esas respuestas, la mejora no se despliega, por buena que se vea en una pantalla.

La segunda situación aparece cuando algo falla en producción. En un prototipo, depurar es releer la conversación; en un sistema operable, es abrir la traza de esa ejecución concreta y ver qué modelo, qué prompt y qué contexto usó, cuánto costó y dónde se rompió. La distancia entre «creo que fue el modelo» y «la traza muestra un timeout en el retrieval» es la distancia entre una hora y un día de trabajo, y se decide mucho antes, el día que alguien decide qué guardar de cada run.

La tercera es la del coste. Un prototipo se mira por el precio por token; un sistema operable, por el coste por tarea aceptada, que reparte entre los aciertos el gasto de los reintentos, las validaciones y la revisión humana. Un sistema que parece barato por llamada puede resultar carísimo por resultado útil, y esa cuenta solo se ve cuando se opera de verdad, con tráfico real y semanas por delante.

Por qué debería importarte

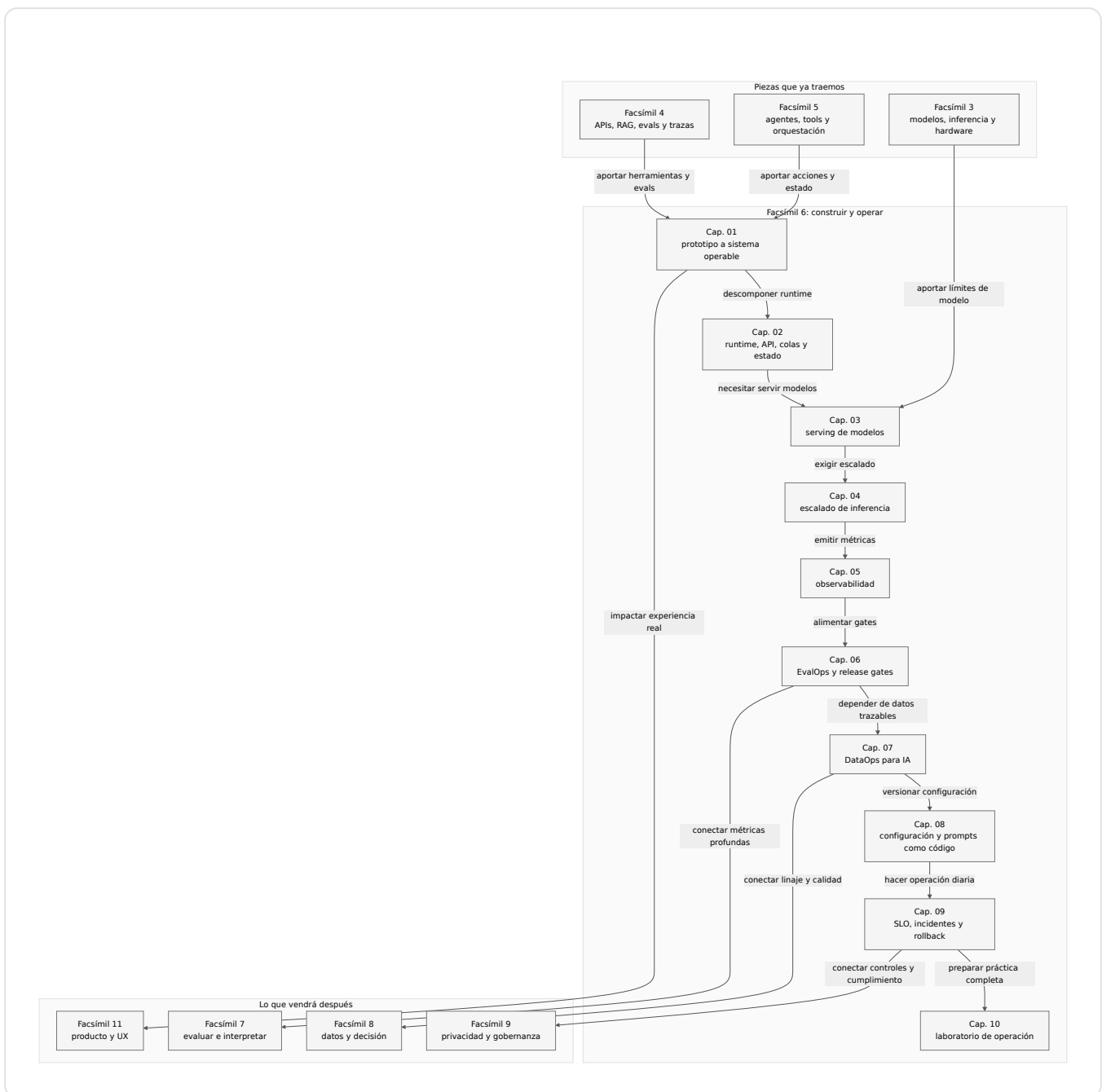
Porque la mayoría de los proyectos de IA no mueren por falta de un buen modelo, sino por no poder operarlos. Nadie sabe qué versión está en producción, por qué falló ayer, cuánto cuesta de verdad ni cómo volver atrás sin romper otra cosa. Las seis piezas de este capítulo (control plane, runtime, observabilidad, evaluación, gates y versionado) son exactamente lo que convierte una demo que impresiona en un sistema que un equipo puede sostener durante meses sin sobresaltos. No son burocracia: son lo que hace que «funciona» signifique algo comprobable.

Y hay una razón más personal. Quien sabe operar un sistema de IA deja de depender de la suerte. Puede prometer una latencia y cumplirla, puede defender un despliegue con evidencia en vez de con entusiasmo, y puede estar tranquilo sabiendo que, si algo se tuerce, hay traza para entenderlo y rollback para arreglarlo. Ese es el cambio de oficio que persigue todo el facsímil, y empieza precisamente aquí, en la decisión de tratar tu prototipo como un sistema que otra persona tendrá que mantener.

Dónde solía tropezar yo

TROIEZO	POR QUÉ ES UN PROBLEMA	ANTÍDOTO
Creer que producción empieza al desplegar	Llegas tarde a trazas, evals, límites y rollback.	Diseñar manifest, eval y gate antes del primer rollout.
Medir solo calidad media	Oculto latencia, coste, errores de contrato y segmentos débiles.	Mirar calidad por slice, p95, coste por tarea aceptada y errores tipados.
Versionar solo el código	El comportamiento cambia también por modelo, prompt, política, dataset y retrieval.	Versionar todos los artefactos que cambian una salida.
No tener estado operativo	Nadie sabe si una run está ejecutando, esperando, reintentando o cerrada.	Modelar estados y transiciones explícitas.
Confundir logs con observabilidad	Guardar texto no basta para reconstruir causalidad.	Usar trazas con IDs, spans, atributos y tiempos.
Aprobar por una sola métrica	Una versión puede mejorar calidad media y romper latencia, coste o contrato.	Exigir gates multidimensionales y presupuesto de error.

Cómo encaja todo



El mapa muestra que este facsímil no es un apéndice técnico. Es el puente entre entender piezas y poder sostenerlas en uso real.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
<code>AGENTS.md</code>	Archivo de instrucciones operativas del repositorio para agentes y personas: mapa, comandos, evidencias, límites y Definition of Done.
<code>SHOULD.md</code>	Especificación de comportamiento esperado: qué debe hacer el sistema, qué no debe hacer y cómo se evalúa.
<code>MUST / SHOULD / MAY</code>	Vocabulario de requisitos para separar obligaciones, expectativas y capacidades opcionales.
Manifest operativo	Documento versionado que fija qué modelo, prompt, política, dataset, contrato, runtime y rollback forman una release concreta.
Sistema operable	Sistema que se puede desplegar, observar, evaluar, limitar y revertir con evidencia.
Control plane	Capa de configuración y gobierno: modelos, prompts, políticas, flags, datasets y límites.
Runtime	Capa que ejecuta peticiones reales con modelo, herramientas, colas, timeouts y validadores.
Run	Ejecución completa de una petición, desde entrada hasta decisión final.
Span	Unidad observable dentro de una traza: llamada al modelo, retrieval, tool, validador o gate.
SLI	<i>Service Level Indicator</i> : indicador medido, como calidad aceptada, p95, coste por tarea aceptada o error de contrato.
SLO	<i>Service Level Objective</i> : objetivo medible que fija qué nivel mínimo queremos sostener para un SLI.
Presupuesto de error	<i>Error budget</i> : margen de fallo que permite un SLO antes de frenar cambios o investigar degradaciones.
Máquina de estados	Lista de estados y transiciones válidas para una ejecución.
Release gate	Regla que decide si un cambio avanza, se pausa o vuelve atrás.
Canary	Exposición gradual de una versión nueva a una parte pequeña del tráfico.
Shadow run	Ejecución en paralelo que no afecta al usuario, usada para comparar.
Coste por tarea aceptada	Coste real por salida válida, incluyendo reintentos, herramientas y revisión.

TÉRMINO**DEFINICIÓN**

Rollback

Vuelta preparada a una versión anterior.

Antes de pasar página

- ☐ ¿Puedes explicar la diferencia entre demo, prototipo y sistema operable?
- ☐ ¿Puedes enumerar qué piezas componen el control plane de un sistema de IA?
- ☐ ¿Puedes decir qué debería guardar una run mínima?
- ☐ ¿Puedes justificar por qué el coste por tarea aceptada es más útil que el precio por token?
- ☐ ¿Puedes convertir un criterio de release en condiciones ejecutables?
- ☐ ¿Puedes escribir un `AGENTS.md` que indique comandos, evidencia mínima, límites y Definition of Done?
- ☐ ¿Puedes escribir un `SHOULD.md` que separe comportamiento obligatorio, esperado y opcional?
- ☐ ¿Puedes convertir un DEBE de `SHOULD.md` en una señal de evaluación dentro del gate?
- ☐ ¿Puedes leer un `manifest.yaml` y reconstruir qué modelo, prompt, dataset y contrato se evaluaron?
- ☐ ¿Puedes explicar por qué una mejora de calidad puede no merecer despliegue si rompe latencia, coste, contrato o rollback?
- ☐ ¿Puedes dibujar la máquina de estados de una run y decir qué evento mueve cada transición?
- ☐ ¿Puedes distinguir SLI, SLO y presupuesto de error con un ejemplo numérico?
- ☐ ¿Puedes clasificar un fallo como entrada, retrieval, contexto, contrato, latencia, coste, estado o cambio?

En resumen

IDEA	QUÉ DEBES LLEVARTE
Una demo no es un sistema.	Un sistema operable deja versiones, trazas, métricas, contratos, gates y rollback.
La IA añade artefactos al cambio.	No basta versionar código: también cambian modelo, prompt, contexto, política, dataset y runtime.
Operar es medir decisiones.	Calidad, latencia, coste, contrato y rollback deben convertirse en criterios ejecutables.
Ingeniería es modelar fallos.	Estados, transiciones, SLI, SLO, taxonomía de fallos y presupuesto de error hacen el sistema depurable.
El facsímil empieza aquí.	A partir de ahora bajaremos pieza a pieza: runtime, serving, escalado, observabilidad, EvalOps, DataOps y operación diaria.

Para saber más

- Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B. y Zimmermann, T. (2019). *Software Engineering for Machine Learning: A Case Study*. International Conference on Software Engineering: Software Engineering in Practice, 291-300. <https://doi.org/10.1109/ICSE-SEIP.2019.00042>
- Amazon Web Services. (2026). *AWS Health Dashboard: Service health*. <https://docs.aws.amazon.com/health/latest/ug/aws-health-dashboard-status.html>
- Amazon Web Services. (2026). *Troubleshooting Amazon Bedrock API Error Codes*. <https://docs.aws.amazon.com/bedrock/latest/userguide/troubleshooting-api-error-codes.html>
- Anthropic. (2026). *Claude Status*. <https://status.claude.com/>
- Anthropic. (2026). *Errors*. <https://platform.claude.com/docs/en/api/errors>
- Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., Koo, C. Y., Lew, L., Mewald, C., Modi, A. N., Polyzotis, N., Ramesh, S., Roy, S., Whang, S. E., Wicke, M., Wilkiewicz, J., Zhang, X. y Zinkevich, M. (2017). *TFX: A TensorFlow-Based Production-Scale Machine Learning Platform*. Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 1387-1395. <https://doi.org/10.1145/3097983.3098021>
- Bradner, S. (1997). *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119. <https://www.rfc-editor.org/rfc/rfc2119>

- Fowler, M. (2025). *Harness Engineering for Coding Agent Users*. <https://martinfowler.com/articles/harness-engineering.html>
- Google. (2026). *Gemini API troubleshooting guide*. <https://ai.google.dev/gemini-api/docs/troubleshooting>
- Leiba, B. (2017). *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*. RFC 8174. <https://www.rfc-editor.org/rfc/rfc8174>
- OpenAI. (2026). *AGENTS.md*. <https://github.com/openai/agents.md>
- OpenAI. (2026). *API reference: debugging requests*. <https://developers.openai.com/api/reference/overview#debugging-requests>
- OpenAI. (2026). *Error codes*. <https://developers.openai.com/api/docs/guides/error-codes>
- OpenAI. (2026). *OpenAI Status*. <https://status.openai.com/>
- OpenTelemetry. (2026). *Tracing API*. <https://opentelemetry.io/docs/specs/otel/trace/api/>
- Preston-Werner, T. (2026). *Semantic Versioning 2.0.0*. <https://semver.org/>
- Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F. y Dennison, D. (2015). *Hidden Technical Debt in Machine Learning Systems*. Advances in Neural Information Processing Systems, 28. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcdf2674f757a2463eba-Abstract.html
- World Wide Web Consortium. (2021). *Trace Context Level 2*. <https://www.w3.org/TR/trace-context-2/>

Notas

1. Sculley, D. y otros (2015). *Hidden Technical Debt in Machine Learning Systems*. Advances in Neural Information Processing Systems, 28. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcdf2674f757a2463eba-Abstract.html. Consultado el 27 de mayo de 2026. ↵
2. Amershi, S. y otros (2019). *Software Engineering for Machine Learning: A Case Study*. International Conference on Software Engineering: Software Engineering in Practice, 291-300. <https://doi.org/10.1109/ICSE-SEIP.2019.00042>. Consultado el 27 de mayo de 2026. ↵
3. Preston-Werner, T. (2026). *Semantic Versioning 2.0.0*. <https://semver.org/>. Consultado el 27 de mayo de 2026. ↵
4. Humble, J. y Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley. Formaliza las puertas de calidad automatizadas en el pipeline de entrega. ↵

5. Trivedi, K. S. (2016). *Probability and Statistics with Reliability, Queuing, and Computer Science Applications* (2.ª ed.). Wiley. La disponibilidad de estado estacionario de un sistema reparable es el cociente entre el tiempo medio en funcionamiento y la suma del tiempo medio en funcionamiento más el de reparación. ↵
6. Trivedi, K. S. (2016). *Probability and Statistics with Reliability, Queuing, and Computer Science Applications* (2.ª ed.). Wiley. Para componentes en serie (todos necesarios e independientes), la fiabilidad o disponibilidad del sistema es el producto de las de cada componente. ↵
7. Wilson, E. B. (1927). Probable Inference, the Law of Succession, and Statistical Inference. *Journal of the American Statistical Association*, 22(158), 209-212.
<https://doi.org/10.2307/2276774>. Introduce el intervalo de confianza de puntuación para una proporción binomial. ↵
8. Baylor, D. y otros (2017). *TFX: A TensorFlow-Based Production-Scale Machine Learning Platform*. Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 1387-1395. <https://doi.org/10.1145/3097983.3098021>. Consultado el 27 de mayo de 2026. ↵
9. OpenTelemetry. (2026). *Tracing API*. <https://opentelemetry.io/docs/specs/otel/trace/api/>. Consultado el 27 de mayo de 2026. ↵
10. World Wide Web Consortium. (2021). *Trace Context Level 2*. <https://www.w3.org/TR/trace-context-2/>. Consultado el 27 de mayo de 2026. ↵
11. OpenAI. (2026). *API reference: debugging requests*.
<https://developers.openai.com/api/reference/overview#debugging-requests>. Consultado el 27 de mayo de 2026. ↵
12. Anthropic. (2026). *Errors*. <https://platform.claude.com/docs/en/api/errors>. Consultado el 27 de mayo de 2026. ↵
13. Google. (2026). *Gemini API troubleshooting guide*. <https://ai.google.dev/gemini-api/docs/troubleshooting>. Consultado el 27 de mayo de 2026. ↵
14. Amazon Web Services. (2026). *Troubleshooting Amazon Bedrock API Error Codes*.
<https://docs.aws.amazon.com/bedrock/latest/userguide/troubleshooting-api-error-codes.html>. Consultado el 27 de mayo de 2026. ↵
15. OpenAI. (2026). *Error codes*. <https://developers.openai.com/api/docs/guides/error-codes>. Consultado el 27 de mayo de 2026. ↵
16. OpenAI. (2026). *OpenAI Status*. <https://status.openai.com/>. Consultado el 27 de mayo de 2026. ↵
17. Anthropic. (2026). *Claude Status*. <https://status.claude.com/>. Consultado el 27 de mayo de 2026. ↵

- .8. Amazon Web Services. (2026). *AWS Health Dashboard: Service health*.
<https://docs.aws.amazon.com/health/latest/ug/aws-health-dashboard-status.html>. Consultado el 27 de mayo de 2026. ↵
- .9. OpenAI. (2026). *AGENTS.md*. <https://github.com/openai/agents.md>. Consultado el 27 de mayo de 2026. ↵
- .10. Fowler, M. (2025). *Harness Engineering for Coding Agent Users*.
<https://martinfowler.com/articles/harness-engineering.html>. Consultado el 27 de mayo de 2026. ↵
- .11. Bradner, S. (1997). *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119.
<https://www.rfc-editor.org/rfc/rfc2119>. Consultado el 27 de mayo de 2026. ↵
- .12. Leiba, B. (2017). *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*. RFC 8174.
<https://www.rfc-editor.org/rfc/rfc8174>. Consultado el 27 de mayo de 2026. ↵

Capítulo 02: Arquitectura de runtime: API, colas, estado y contratos

Qué deberías poder hacer al terminar

En el capítulo anterior construimos el marco: manifest, AGENTS.md , SHOULD.md , gates, trazas y decisiones. Ahora bajamos un nivel. Si alguien llama a tu sistema de IA, ¿qué ocurre exactamente desde que entra la petición hasta que sale la respuesta?

Al terminar, deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Diseñar la entrada de una API de IA.	Separas usuario, tarea, payload, permisos, idempotencia y presupuesto.
Modelar una petición como una run.	Distingues run_id , request_id , trace_id , estado, spans y resultado.
Explicar por qué una cola cambia la arquitectura.	Calculas presión de cola, tiempo de espera y capacidad mínima.
Diseñar contratos de entrada y salida.	Usas esquemas, validadores, errores tipados y versión de contrato.
Definir timeouts y streaming.	Repartes presupuesto de latencia entre fases y decides qué se entrega progresivamente.

La idea central es sencilla pero potente: **un runtime de IA no es una llamada al modelo; es una máquina de coordinación.**

La petición que parece simple

Imagina un botón en una aplicación universitaria: “resumir expediente y proponer siguiente paso”. Desde la interfaz parece una acción limpia: pulsas, espera unos segundos y aparece una respuesta. Pero por debajo ocurren muchas cosas.

El sistema debe comprobar quién llama, qué permiso tiene, qué versión del asistente está activa, si esa petición ya llegó antes, qué documentos necesita recuperar, qué modelo debe usar, cuánto tiempo queda, qué salida será válida, qué hacer si el proveedor tarda demasiado y qué traza dejar para poder depurar.

Por eso este capítulo no empieza en el modelo. Empieza en el borde del sistema: la API. En ingeniería, los bordes importan porque ahí se convierten intenciones humanas en contratos técnicos.

Qué no es el runtime

Un runtime no es “el SDK del proveedor”. El SDK ayuda a llamar a un modelo, pero no decide por sí solo cómo versionas prompts, cómo deduplicas peticiones, cómo gestionas una cola, cómo validas una respuesta o cómo haces rollback.

Tampoco es solo “el servidor”. Puedes tener un servidor HTTP y seguir sin runtime operativo: si no hay estado de run, si no hay contrato, si no hay trazas y si no sabes qué hacer cuando una dependencia va lenta, tienes una puerta de entrada, no un sistema.

La diferencia práctica es esta:

CONFUSIÓN	QUÉ FALTA
Endpoint HTTP	Falta estado, idempotencia, validación, trazas y política de errores.
SDK del modelo	Falta coordinación con retrieval, tools, colas, contratos y presupuesto.
Script que llama a una API	Falta operación: retries, timeouts, owners, métricas y rollback.
Worker suelto	Falta control de entrada, orden, prioridad, cancelación y deduplicación.

Qué sí es un runtime de IA

Un runtime de IA es la capa que ejecuta peticiones reales bajo reglas operativas. Recibe una intención, la convierte en una run, coordina pasos, conserva estado, llama a dependencias, valida salida y deja evidencia.

La API HTTP sigue siendo importante. RFC 9110 define la semántica de HTTP: métodos, estado, representación y comportamiento esperado de las peticiones.¹ Pero una API de IA añade dimensiones que un CRUD clásico no siempre tenía tan visibles:

DIMENSIÓN	PREGUNTA QUE RESPONDE
Identidad	¿Quién pide esto y con qué permisos?
Idempotencia	¿Qué pasa si el cliente repite la petición?
Contexto	¿Qué documentos, memoria o estado entran en la run?
Presupuesto	¿Cuánto tiempo, coste y tokens puede gastar?
Contrato	¿Qué forma debe tener la salida para ser aceptada?
Observabilidad	¿Cómo reconstruimos lo ocurrido?
Degradación	¿Qué hacemos si una dependencia no responde a tiempo?

Fecha de corte: 27 de mayo de 2026. Fuentes consultadas ese día: documentación oficial de OpenAI Responses API, streaming y salidas estructuradas; documentación de Anthropic Messages API y streaming; JSON Schema Draft 2020-12; OpenTelemetry Tracing API y convenciones GenAI; W3C Trace Context. Lo estable es la arquitectura: API, estado, colas, contratos, timeouts y trazas. Lo cambiante son endpoints concretos, eventos de streaming, nombres de modelos, SDKs y límites de proveedor.

El mecanismo por dentro

Una petición robusta suele pasar por estas fases:

1. **Entrada:** llega una petición HTTP con usuario, payload, clave de idempotencia y presupuesto.
2. **Normalización:** el sistema valida formato, permisos, tamaño y campos obligatorios.
3. **Creación de run:** se asignan `run_id`, `trace_id`, versión de manifest y estado inicial.
4. **Planificación:** se decide si la petición va directa, a cola, a worker especializado o a revisión.
5. **Ejecución:** retrieval, herramientas, llamada al modelo, streaming y validación.
6. **Cierre:** se guarda resultado, métricas, coste, contrato, estado final y decisión.

La latencia total se puede leer como suma de fases:

$$T_{\text{total}} = T_{\text{entrada}} + T_{\text{cola}} + T_{\text{retrieval}} + T_{\text{tools}} + T_{\text{modelo}} + T_{\text{validación}} + T_{\text{salida}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
T_{entrada}	Tiempo de autenticación, parseo y validación inicial.	80 ms
T_{cola}	Tiempo esperando a que un worker pueda ejecutar.	600 ms
$T_{\text{retrieval}}$	Tiempo de buscar y preparar contexto.	350 ms
T_{tools}	Tiempo consumido por herramientas externas.	500 ms
T_{modelo}	Tiempo de prefill, generación y respuesta del modelo.	1800 ms
$T_{\text{validación}}$	Tiempo de comprobar contrato y reglas.	90 ms
T_{salida}	Tiempo de serializar, emitir eventos o guardar resultado.	50 ms

Con esos números:

$$T_{\text{total}} = 80 + 600 + 350 + 500 + 1800 + 90 + 50 = 3470 \text{ ms}$$

Si tu SLO (*Service Level Objective*, objetivo de nivel de servicio) era `respuesta final <= 3000 ms`, el problema no se arregla diciendo “el modelo es lento”. Tienes que mirar la descomposición. En este ejemplo, cola y modelo suman 2400 ms. Si además retrieval o tools se mueven un poco, el SLO cae.

Las colas tienen una ley muy útil para razonar. La ley de Little dice:

$$L = \lambda W$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
L	Número medio de trabajos dentro del sistema o de una cola.	24 runs
λ	Tasa media de llegada de trabajos.	6 runs/s
W	Tiempo medio que un trabajo pasa dentro del sistema.	4 s

La lectura sería:

$$24 = 6 \times 4$$

Little demostró esta relación para sistemas de colas bajo condiciones generales.² Para nosotros la intuición es suficiente: si llegan más runs por segundo y cada una tarda más, la cola crece. Si la cola crece, sube T_{cola} . Si sube T_{cola} , el usuario siente que “la IA va lenta”,

aunque el modelo no haya cambiado.

También necesitamos repartir presupuesto de tiempo:

$$B_{\text{SLO}} \geq B_{\text{cola}} + B_{\text{retrieval}} + B_{\text{tools}} + B_{\text{modelo}} + B_{\text{validación}} + M$$

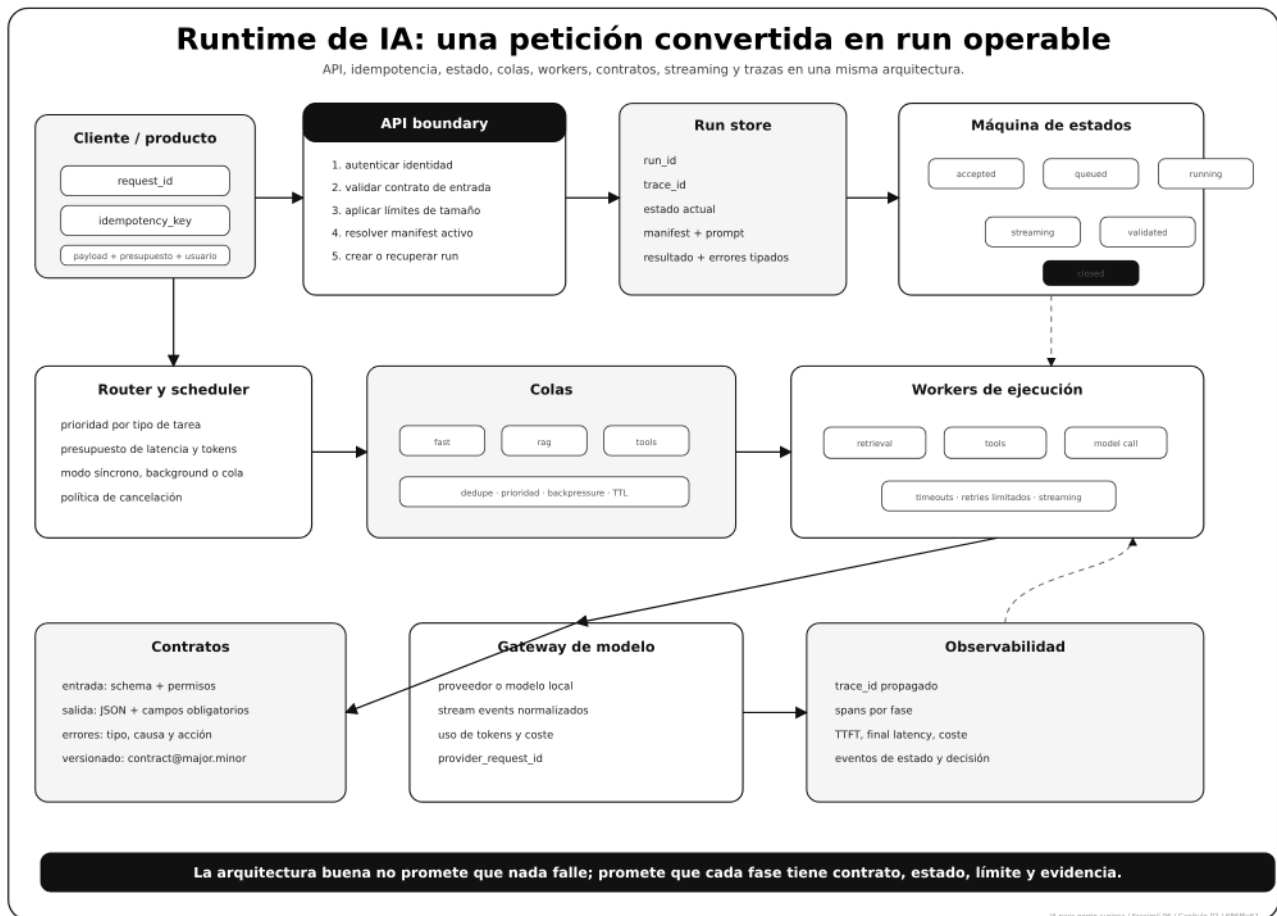
SÍMBOLO	SIGNIFICADO	EJEMPLO
B_{SLO}	Presupuesto total de latencia.	3000 ms
B_{cola}	Máximo tolerable de espera en cola.	400 ms
$B_{\text{retrieval}}$	Máximo para recuperar contexto.	300 ms
B_{tools}	Máximo para herramientas externas.	500 ms
B_{modelo}	Máximo para modelo.	1500 ms
$B_{\text{validación}}$	Máximo para validadores.	100 ms
M	Margen para red, serialización y variabilidad.	200 ms

Aquí:

$$3000 \geq 400 + 300 + 500 + 1500 + 100 + 200$$

La igualdad no es el objetivo. El objetivo es detectar que si `tools` tarda 900 ms, no puedes fingir que todo sigue dentro de presupuesto.

Anatomía visual de un runtime



El diagrama no pretende imponer una única arquitectura. Pretende darte una checklist mental. Si una petición no tiene `run_id`, no puedes seguirla. Si no tiene idempotencia, una repetición puede duplicar trabajo. Si no hay cola, los picos llegan directos al modelo. Si no hay contrato de salida, el frontend y los sistemas aguas abajo quedan expuestos a respuestas imprevisibles.

API de entrada: más que un prompt

Una API de IA debería recibir algo más estructurado que `{ "prompt": "..."}` . Ese formato sirve para un tutorial. En un sistema real, la entrada debería distinguir intención, usuario, tarea, permisos, presupuesto y contrato esperado.

Un ejemplo mínimo:

```
{
  "request_id": "req_20260527_0001",
  "idempotency_key": "user42:support-summary:expediente-983:v1",
  "tenant_id": "universidad-demo",
  "actor": {
    "user_id": "user42",
    "role": "orientador"
  },
  "task": {
    "type": "support_summary",
    "input": {
      "case_id": "expediente-983",
      "question": "Resume la situación y propone siguiente paso."
    }
  },
  "budget": {
    "max_latency_ms": 3000,
    "max_input_tokens": 8000,
    "max_output_tokens": 700,
    "max_cost_eur": 0.03
  },
  "response_contract": "support-answer@1.2.0",
  "stream": true
}
```

Para no depender mentalmente de un único proveedor, conviene mirar cómo aparecen estas piezas en varias documentaciones:

PIEZA	FUENTE CONSULTADA	QUÉ NOS INTERESA PARA EL RUNTIME
Creación de respuestas	OpenAI Responses API. ³	Entrada, modelo, herramientas, instrucciones y respuesta como recurso.
Streaming	OpenAI streaming. ⁴	Eventos progresivos y experiencia de usuario.
Salida estructurada	OpenAI structured outputs. ⁵	Contrato de salida validable.
Mensajes	Anthropic Messages API. ⁶	Conversación stateless o multi-turn en formato de mensajes.
Streaming de mensajes	Anthropic streaming. ⁷	Secuencia de eventos y tratamiento de bloques de contenido.

El punto de ingeniería no es memorizar el endpoint de hoy. Es diseñar una capa interna que pueda hablar con varios proveedores sin que el resto del producto dependa de cada detalle.

Identidad, permisos y tenant

La API boundary no solo valida JSON. También decide **quién** puede hacer **qué** sobre **qué run**. En sistemas de IA esto importa porque una run puede contener documentos recuperados, eventos internos, costes, decisiones y salidas que no deberían ser visibles para cualquiera.

Tres preguntas mínimas:

PREGUNTA	QUÉ COMPRUEBA	EJEMPLO
Autenticación	¿Quién eres?	Token de sesión, API key, OAuth, certificado interno.
Autorización	¿Qué puedes hacer?	Crear runs, cancelar, leer eventos, ver fuentes, reintentar.
Tenancy	¿De qué organización o espacios eres?	<code>tenant_id = universidad-demo</code> .

Una regla práctica: cada operación sobre `/runs/{run_id}` debe comprobar que el actor pertenece al tenant correcto y tiene permiso para esa acción. No basta con que conozca el `run_id`. Un identificador no es una autorización.

OPERACIÓN	PERMISO MÍNIMO	RIESGO SI SE OLVIDA
<code>POST /runs</code>	Crear run en ese tenant.	Ejecutar tareas fuera de contexto.
<code>GET /runs/{run_id}</code>	Leer run propia o del equipo autorizado.	Exponer documentos, coste o decisión.
<code>GET /runs/{run_id}/events</code>	Leer eventos de esa run.	Filtrar trazas internas o fuentes.
<code>POST /runs/{run_id}/cancel</code>	Cancelar runs propias o administradas.	Parar trabajo de otra persona.
<code>POST /runs/{run_id}/retry</code>	Reintentar bajo presupuesto y permiso.	Duplicar coste o saltarse revisión.

En el run store, `tenant_id` y `actor_id` no son metadatos decorativos. Son parte de la frontera de seguridad y de auditoría. Si una traza no conserva tenant y actor, luego no sabes quién hizo qué ni qué podía ver.

Patrones de respuesta de una API de IA

En una API clásica, muchas veces esperas que una petición entre y salga con un `200 OK` . En IA eso no siempre es lo mejor. Algunas tareas son cortas y pueden responder en la misma conexión. Otras requieren cola, streaming, revisión o ejecución en segundo plano. Diseñar bien el runtime empieza por elegir el patrón de respuesta adecuado.

PATRÓN	RESPUESTA INICIAL	CUÁNDO ENCAJA	QUÉ DEBE QUEDAR GUARDADO
Síncrono	<code>200 OK</code> con resultado final.	Tareas rápidas, baratas y de bajo impacto.	<code>run_id</code> , contrato validado, coste y traza.
Aceptado y polling	<code>202 Accepted</code> con <code>run_id</code> .	Tareas lentas o con cola.	Estado de run, endpoint de consulta y TTL.
Streaming	<code>200 OK</code> con eventos SSE o stream equivalente.	Experiencia interactiva o progreso visible.	Eventos emitidos, primer evento, cierre y errores.
Background job	<code>202 Accepted</code> y trabajo en cola.	Procesos largos que no necesitan conexión abierta.	Cola, prioridad, owner, reintentos y resultado.
Webhook	<code>202 Accepted</code> y callback posterior.	Integraciones entre sistemas.	URL registrada, firma, intentos y confirmación de entrega.
Híbrido	Progreso por streaming y resultado persistido.	Tareas con UX viva y necesidad de auditoría.	Stream, run final y contrato validado.

El patrón no es solo una decisión de backend. Afecta al usuario. Un `202 Accepted` honesto dice: “he recibido tu trabajo, aquí tienes un identificador, vuelve a consultar”. Un streaming honesto dice: “estoy trabajando y estos son eventos de progreso”. Un `200 OK` forzado en una tarea lenta suele terminar en timeouts, pantallas congeladas o reintentos del cliente que duplican coste.

Una API de runtime publicable podría tener esta forma:

ENDPOINT	MÉTODO	PROPÓSITO	RESPUESTA NORMAL
/runs	POST	Crear una run o recuperar una existente por idempotencia.	201 Created o 200 OK si era replay.
/runs/{run_id}	GET	Consultar estado, resultado y metadatos.	200 OK .
/runs/{run_id}/events	GET	Leer eventos de progreso por streaming.	200 OK con text/event-stream .
/runs/{run_id}/cancel	POST	Pedir cancelación cooperativa.	202 Accepted .
/runs/{run_id}/retry	POST	Reintentar una run cerrada con error recuperable.	202 Accepted .
/runs/{run_id}/decision	GET	Ver por qué se cerró, se pausó o se envió a revisión.	200 OK .

HTTP ya nos da códigos útiles, pero hay que usarlos con intención:

CÓDIGO	USO RAZONABLE EN RUNTIME DE IA
200 OK	Resultado disponible o replay idempotente ya resuelto.
201 Created	Run creada y aceptada.
202 Accepted	Run aceptada, pero todavía no terminada.
400 Bad Request	Payload inválido.
401/403	Identidad ausente o permiso insuficiente.
404 Not Found	run_id inexistente o no visible para ese actor.
409 Conflict	La run no admite esa transición de estado.
422 Unprocessable Content	Entrada bien formada, pero no cumple contrato semántico.
429 Too Many Requests	Límite por usuario, tenant o presupuesto.
503 Service Unavailable	Sistema saturado o dependencia crítica no disponible.

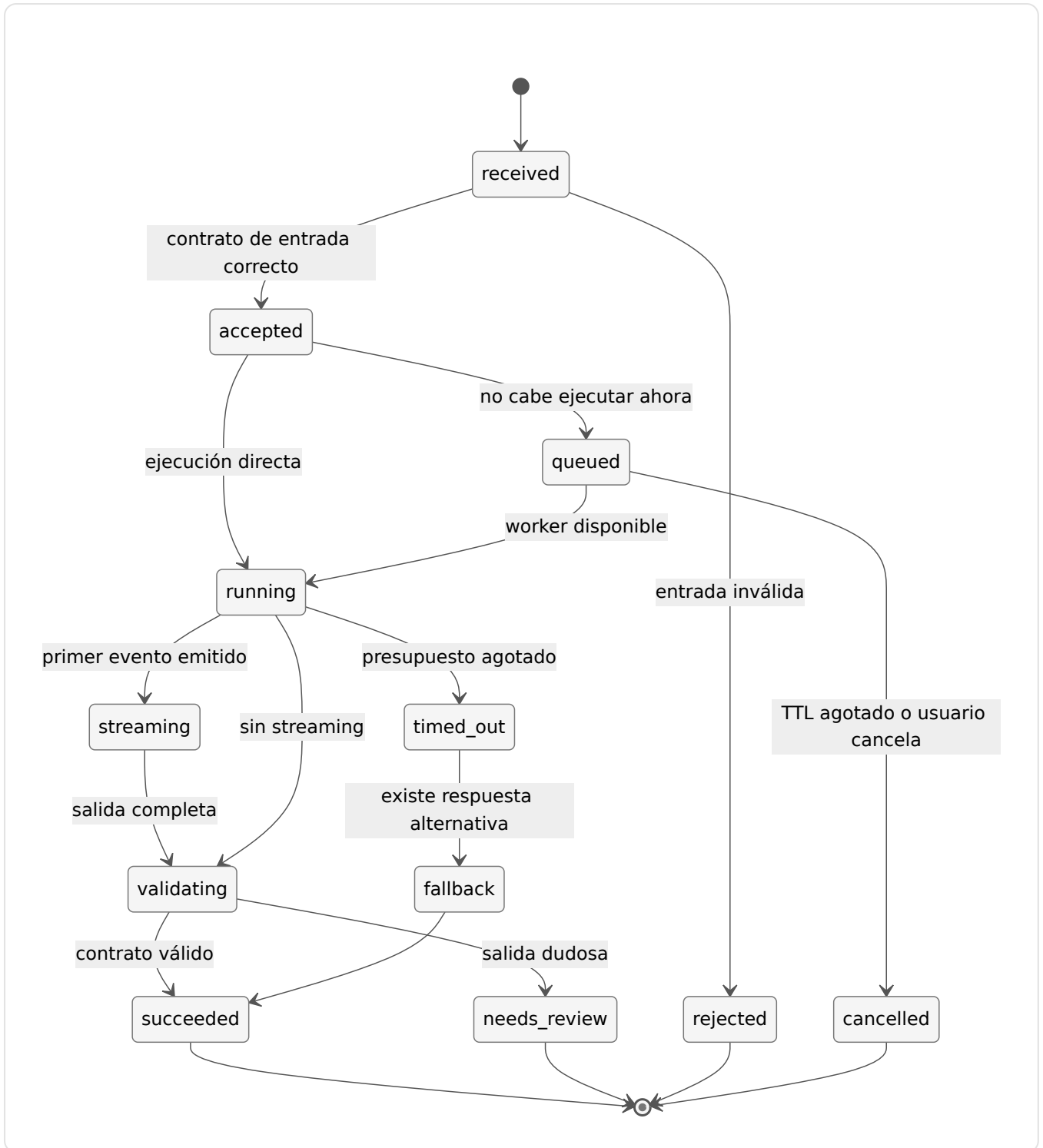
El cliente no debería tener que adivinar. Cada error debería incluir `error_code` , `message` , `run_id` si existe, `trace_id` si existe, `retryable` , `retry_after_ms` cuando aplique y una acción recomendada.

Estado: la run como fuente de verdad

Una run es una ejecución completa. No es lo mismo que una request HTTP, porque una request puede terminar rápido diciendo “aceptado” mientras la run sigue en background. Tampoco es lo mismo que una llamada al modelo, porque una run puede incluir retrieval, herramientas, validación, streaming, reintentos y cierre.

IDENTIFICADOR	QUÉ IDENTIFICA	QUIÉN LO USA
<code>request_id</code>	La petición recibida por tu API.	Cliente, gateway y logs HTTP.
<code>idempotency_key</code>	La intención repetible que no debe duplicarse.	API boundary y run store.
<code>run_id</code>	La ejecución de negocio que queremos seguir.	Producto, soporte, evals y operación.
<code>trace_id</code>	La traza distribuida de todos los spans.	Observabilidad.
<code>provider_request_id</code>	La llamada concreta a un proveedor/modelo.	Depuración con proveedor o runtime.

Una máquina de estados mínima podría ser:



Cada transición debería guardar evento, timestamp y motivo. No basta con saber que una run terminó mal. Hay que saber si fue rechazada por contrato, cancelada por TTL, parada por timeout, enviada a revisión o cerrada correctamente.

Persistencia mínima del run store

El run store no tiene por qué ser sofisticado al principio, pero sí debe guardar lo suficiente para reconstruir una ejecución. Si solo guardas el texto final, pierdes la historia. Si guardas todo sin criterio, creas una base de datos difícil de proteger y consultar.

Un modelo mínimo podría tener cuatro tablas o colecciones:

TABLA	QUÉ GUARDA	CAMPOS MÍNIMOS
runs	La entidad principal de ejecución.	run_id , tenant_id , actor_id , state , manifest_version , created_at , updated_at .
run_events	Cambios de estado y eventos importantes.	run_id , event_name , state , timestamp , attrs_json .
run_outputs	Resultado final o salida candidata.	run_id , contract_version , output_json , validated , created_at .
run_errors	Fallos normalizados.	run_id , error_code , retryable , provider_request_id , message , created_at .

Y una tabla opcional cuando hay streaming:

TABLA	QUÉ GUARDA	CUÁNDO MERECE LA PENA
run_stream_events	Eventos enviados al cliente, con orden.	Si necesitas reanudar stream, auditar UX o depurar cortes de conexión.

Hay una decisión delicada: qué guardar de la entrada y la salida. En muchos sistemas no conviene guardar prompts completos o documentos enteros. Puede bastar con hashes, IDs de documentos, metadatos mínimos y una política clara de retención. Operar no significa guardar todo; significa guardar lo necesario para explicar y mejorar el sistema sin acumular datos de más.

Cancelar, reanudar y expirar

El ciclo de vida de una run se complica cuando el usuario cierra la pestaña, recarga, cancela o pierde conexión. Un runtime serio no debería depender de que la conexión HTTP sobreviva hasta el final.

SITUACIÓN	QUÉ DEBERÍA HACER EL RUNTIME
El usuario recarga la página.	Recuperar por <code>run_id</code> o <code>idempotency_key</code> ; no crear otra run.
Se corta el streaming.	Mantener la run viva y permitir reconectar a <code>/runs/{run_id}/events</code> .
El usuario cancela.	Marcar <code>cancelling</code> , avisar a workers y cerrar como <code>cancelled</code> si no hay fase irreversible.
La run espera demasiado en cola.	Cerrar como <code>expired</code> si supera TTL.
El worker se cae.	Reencolar si la fase es recuperable y no supera máximo de intentos.
El contrato falla.	Cerrar como <code>contract_failed</code> y no mostrar resultado final.

Para reanudación hace falta distinguir dos cosas:

PIEZA	QUÉ RESUELVE
<code>run_id</code>	Permite consultar una ejecución ya creada.
<code>resume_token</code>	Permite reconectar a un stream desde un punto conocido si el protocolo lo soporta.
<code>last_event_id</code>	Permite saber qué eventos ya vio el cliente.
<code>result_url</code>	Permite recuperar resultado final aunque el stream se perdiera.

La cancelación también debe ser honesta. Cancelar una run que todavía está en cola es fácil. Cancelar una llamada al modelo ya enviada puede depender del proveedor o del runtime. Cancelar una acción externa que ya se ejecutó quizá no sea posible. Por eso una buena API no promete “cancelado” demasiado pronto; primero marca `cancelling` , luego confirma `cancelled` o explica que la fase ya no era cancelable.

Colas: cuando el tráfico deja de ser educado

La cola aparece cuando no quieres que todas las peticiones entren al runtime al mismo tiempo. En IA esto es especialmente importante porque cada petición puede tener coste muy distinto. Una FAQ corta y un análisis con 80.000 tokens no consumen lo mismo.

Las colas sirven para:

PROBLEMA	CÓMO AYUDA LA COLA
Picos de tráfico	Absorbe entradas y permite ejecutar a ritmo controlado.
Tareas lentas	Separa respuesta inmediata de trabajo en background.
Prioridades	Permite colas por tipo: rápido, RAG largo, herramientas, revisión.
Coste	Evita disparar llamadas caras sin control.
Backpressure	Permite decir “no acepto más” antes de romper SLO.

Pero una cola también puede esconder problemas. Si nadie mira longitud, edad máxima y tiempo de espera, la cola se convierte en una sala de espera infinita. Una buena cola tiene TTL, prioridad, límites y métricas.

Una cola profesional suele añadir estas piezas:

PIEZA	QUÉ RESUELVE	SEÑAL QUE DEBERÍAS MEDIR
Prioridad	No todas las tareas tienen la misma urgencia.	Tiempo de espera por prioridad.
Fairness por tenant	Un cliente grande no debería ocupar toda la capacidad.	Cuota usada por tenant.
Backoff	Reintentar demasiado rápido empeora la saturación.	Intentos, retraso y causa.
Dead-letter queue	Los trabajos que fallan siempre no bloquean la cola principal.	Entradas en DLQ y motivo.
TTL	Una tarea demasiado vieja quizá ya no sirve.	Edad máxima en cola.
Dedupe	Dos trabajos equivalentes no deberían gastar dos veces.	Replays por idempotencia.
Backpressure	El sistema avisa antes de romperse.	429 , 503 , cola llena y rechazos.

Un patrón típico de reintento es backoff exponencial con jitter:

$$d_n = \min(d_{\max}, d_0 \cdot 2^n) + \epsilon$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
d_n	Espera antes del intento n .	4000 ms
d_0	Espera inicial.	500 ms
n	Número de intento, empezando en 0.	3
$d_{\max}$	Límite máximo de espera.	10.000 ms
ϵ	Pequeña variación aleatoria para evitar sincronización.	120 ms

Sin jitter, muchos workers pueden reintentar a la vez y crear otro pico artificial. Con jitter, los reintentos se reparten un poco mejor. La idea importante: reintentar no es gratis. Cada reintento consume cola, coste, tokens y atención operativa.

Cuánto hay dentro: la ley de Little. La relación más útil de toda la teoría de colas no necesita suponer nada sobre cómo llegan las peticiones ni cuánto tardan: en régimen estable, el número medio de peticiones dentro del sistema es la tasa de llegada por el tiempo medio que cada una pasa dentro.⁸

$$L = \lambda W$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
L	Peticiones medias dentro del sistema (en cola más en servicio).	40
λ	Tasa media de llegada.	20 peticiones por segundo.
W	Tiempo medio que cada petición pasa dentro.	2 segundos.

En palabras: si llegan 20 por segundo y cada una vive 2 segundos en tu sistema, en cualquier instante hay unas 40 en vuelo. Sirve en los dos sentidos: mides dos cantidades y deduces la tercera. Si tu pool de conexiones, tu memoria o tu límite de concurrencia no aguanta esas 40 a la vez, no necesitas esperar al incidente para saber que se va a saturar.

Por qué la espera se dispara antes de llenarse: M/M/1. La intuición dice que mientras el servidor no esté al 100% hay margen. La teoría de colas dice lo contrario: el tiempo de espera crece de forma no lineal con la utilización y se va al infinito según te acercas al 100%, mucho antes de «llenarte».⁹

$$W_q = \frac{\rho}{\mu(1-\rho)}, \quad \rho = \frac{\lambda}{\mu}$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
W_q	Tiempo medio de espera en cola (antes de empezar a servirse).	Crece sin límite cuando $\rho \rightarrow 1$.
ρ	Utilización: fracción de tiempo que el servidor está ocupado.	0,8
λ	Tasa de llegada.	8 por segundo.
μ	Tasa de servicio (capacidad).	10 por segundo.

En palabras: al 50% de utilización esperas el equivalente a un servicio; al 90%, nueve veces más; al 95%, diecinueve. Por eso un buen runtime no apunta a exprimir el worker al 99%: deja holgura adrede, porque los últimos puntos de utilización se pagan con latencia que explota.

Por qué la irregularidad empeora la cola: la fórmula de Kingman. M/M/1 supone llegadas y servicios «bien portados». La realidad de un LLM no lo es: hay ráfagas de tráfico y respuestas cuya duración varía muchísimo según los tokens. Kingman cuantifica ese castigo: la espera se multiplica por la variabilidad de las llegadas y de los servicios.¹⁰

$$W_q \approx \left(\frac{\rho}{1 - \rho} \right) \left(\frac{c_a^2 + c_s^2}{2} \right) \mathbb{E}[S]$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
c_a^2	Variabilidad relativa de los tiempos entre llegadas.	1 si es regular tipo Poisson; mayor si hay ráfagas.
c_s^2	Variabilidad relativa de la duración del servicio.	Alta en LLMs: respuestas cortas y largas mezcladas.
$\mathbb{E}[S]$	Tiempo medio de servicio.	0,4 segundos.
ρ	Utilización del servidor.	0,8

En palabras: con la misma carga, una cola con respuestas muy variables espera mucho más que una con respuestas uniformes. Esto justifica dos decisiones de runtime: limitar el tamaño de salida (reduce c_s^2) y suavizar las ráfagas con control de admisión (reduce c_a^2) bajan la latencia sin tocar la capacidad bruta.

Cómo la variabilidad del servicio infla la espera: Pollaczek-Khinchine. Para una cola con un servidor y duraciones de servicio cualesquiera, la fórmula de Pollaczek-Khinchine da la espera exacta y deja ver, sin aproximar, que lo que importa no es solo la media del servicio sino su segundo momento, es decir, su dispersión.¹¹

$$W_q = \frac{\lambda \mathbb{E}[S^2]}{2(1 - \rho)}$$

SÍMBOL O	SIGNIFICADO	EJEMPLO
$\mathbb{E}[S^2]$	Segundo momento del tiempo de servicio (incluye su varianza).	Sube con respuestas largas ocasionales.
λ	Tasa de llegada.	8 por segundo.
ρ	Utilización.	0,8

En palabras: dos sistemas con la misma duración media de respuesta pueden tener esperas muy distintas si uno tiene colas largas ocasionales. Una respuesta lenta de vez en cuando no solo perjudica a quien la sufre: empeora la espera de todos los que van detrás. Por eso acotar el peor caso (timeouts, límite de tokens, troceado) mejora la cola entera, no solo ese caso.

Semántica de entrega: at-most-once y at-least-once

Cuando una cola entrega trabajos a workers, aparece una pregunta clásica: ¿qué garantiza la entrega? Hay tres vocabularios que conviene conocer:

SEMÁNTICA	QUÉ SIGNIFICA	PROBLEMA PRÁCTICO
At-most-once	Como máximo se procesa una vez.	Puedes perder trabajos si el worker cae.
At-least-once	Se procesa una o más veces hasta confirmar.	Puede duplicarse trabajo.
Exactly-once	Parece que se procesa exactamente una vez.	Es difícil y suele depender de transacciones e idempotencia.

En sistemas reales, muchas colas se diseñan alrededor de **at-least-once**: prefieren repetir antes que perder. Eso obliga a que los workers sean idempotentes. Si el mismo trabajo llega dos veces, no debería crear dos resultados incompatibles, gastar sin control o ejecutar dos acciones equivalentes.

La idempotencia se consigue combinando:

TÉCNICA	QUÉ EVITA
<code>idempotency_key</code>	Duplicar una intención del cliente.
<code>run_id</code> estable	Mezclar dos ejecuciones distintas.
Bloqueo por estado	Ejecutar una transición no válida.
Resultado persistido	Reutilizar salida ya validada.
Dedupe de cola	No encolar el mismo trabajo varias veces.

La lección es incómoda pero útil: no diseñes el worker suponiendo que cada mensaje llega una sola vez. Diseña como si pudiera repetirse.

Circuit breakers y bulkheads

Un runtime de IA depende de piezas lentas o variables: proveedor de modelo, vector DB, herramientas internas, base de datos, cola y red. Si una dependencia se degrada, no quieres que arrastre todo el sistema.

Dos patrones ayudan:

PATRÓN	TRADUCCIÓN	USO EN RUNTIME DE IA
Circuit breaker	Cortacircuitos.	Si una dependencia falla demasiado, se deja de llamar durante un tiempo y se degrada.
Bulkhead	Compartimento estanco.	Separar recursos para que una cola o dependencia no consuma toda la capacidad.

Ejemplo: si la herramienta de facturación está lenta, el runtime puede marcar casos como `needs_review`, responder con estado claro o usar una ruta sin herramienta. Lo que no debería hacer es llenar todos los workers esperando esa herramienta hasta que también fallen las preguntas sencillas.

DEPENDENCIA LENTA	CIRCUIT BREAKER	BULKHEAD
Vector DB	Reducir <code>top_k</code> o responder con revisión si no hay contexto.	Cola separada para RAG.
Tool interna	No llamar durante una ventana corta si acumula errores.	Workers separados para tools.
Proveedor de modelo	Cambiar a modelo alternativo o pausar runs nuevas.	Presupuesto por proveedor/modelo.

No hace falta implementar todos estos patrones el primer día. Sí hace falta saber dónde irían. Si el runtime no tiene sitio para degradar, solo le queda fallar.

Contratos: entrada, salida y error

Antes de hablar de JSON Schema conviene parar un momento. En este capítulo, un contrato no es un documento legal ni una promesa vaga. Es una **interfaz verificable**. Dice qué forma debe tener una entrada, qué forma tendrá una salida, qué errores pueden aparecer y qué significan los estados del sistema.

Un contrato técnico sirve para que dos piezas puedan trabajar juntas sin conocerse por dentro:

CONTRATO	PREGUNTA QUE RESPONDE	EJEMPLO
Entrada	¿Qué debe enviar el cliente para que aceptemos la petición?	<code>request_id</code> , <code>idempotency_key</code> , <code>task</code> , <code>budget</code> .
Salida	¿Qué puede esperar quien consume la respuesta?	<code>answer</code> , <code>sources</code> , <code>confidence</code> , <code>needs_review</code> .
Error	¿Cómo se informa un fallo de forma accionable?	<code>error_code</code> , <code>retryable</code> , <code>trace_id</code> , <code>retry_after_ms</code> .
Estado	¿Qué fases puede atravesar una run?	<code>queued</code> , <code>running</code> , <code>succeeded</code> , <code>cancelled</code> .

La palabra clave es “verificable”. Si dices “el modelo responderá bien”, no hay contrato. Si dices “la respuesta debe ser JSON válido, sin campos extra, con al menos una fuente y `confidence` entre 0 y 1”, ya puedes validarlo antes de mostrarlo.

JSON Schema permite describir restricciones estructurales sobre documentos JSON: tipos, campos obligatorios, valores permitidos, mínimos, máximos y otras condiciones.¹² En IA, un esquema no garantiza que el contenido sea correcto, pero sí elimina una clase entera de errores: respuestas que no se pueden parsear, campos que faltan, tipos incorrectos o claves inesperadas.

Ejemplo de contrato de salida:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "support-answer@1.2.0",
  "type": "object",
  "additionalProperties": false,
  "required": ["answer", "sources", "confidence", "needs_review", "next_step"],
  "properties": {
    "answer": { "type": "string", "minLength": 1 },
    "sources": {
      "type": "array",
      "minItems": 1,
      "items": {
        "type": "object",
        "required": ["source_id", "chunk_id", "title"],
        "additionalProperties": false,
        "properties": {
          "source_id": { "type": "string" },
          "chunk_id": { "type": "string" },
          "title": { "type": "string" }
        }
      }
    },
    "confidence": { "type": "number", "minimum": 0, "maximum": 1 },
    "needs_review": { "type": "boolean" },
    "next_step": { "type": "string", "minLength": 1 }
  }
}
```

El contrato de error también importa:

CÓDIGO INTERNO	SIGNIFICADO	QUÉ DEBERÍA HACER EL CLIENTE
<code>input_invalid</code>	La entrada no cumple contrato.	Corregir payload.
<code>idempotency_replay</code>	Ya existe una run para esa intención.	Recuperar run existente.
<code>queue_full</code>	El sistema no acepta más trabajo ahora.	Reintentar después o degradar.
<code>budget_exceeded</code>	Tokens, coste o tiempo superan presupuesto.	Reducir tarea o pedir confirmación.
<code>contract_failed</code>	La salida no cumple esquema.	No mostrar como resultado final.
<code>provider_unavailable</code>	Dependencia externa no responde bien.	Usar fallback o avisar con estado claro.

Un error útil no humilla al usuario ni esconde la causa técnica. Dice qué ocurrió, qué se puede hacer y qué identificador permite depurarlo.

OpenAPI permite describir una API HTTP: rutas, métodos, parámetros, esquemas, respuestas y seguridad.¹³ En un sistema de IA, OpenAPI no sustituye a `SHOULD.md` ; lo traduce al borde técnico. `SHOULD.md` dice cómo debe comportarse el sistema. OpenAPI dice cómo se llama, qué acepta y qué devuelve.

Un fragmento mínimo sería:

```

openapi: 3.1.0
info:
  title: Support RAG Runtime API
  version: 1.2.0
paths:
  /runs:
    post:
      summary: Crear o recuperar una run por idempotencia
      operationId: createRun
      parameters:
        - name: Idempotency-Key
          in: header
          required: true
          schema:
            type: string
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/CreateRunRequest"
      responses:
        "201":
          description: Run creada
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Run"
        "200":
          description: Run existente recuperada por idempotencia
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Run"
        "429":
          description: Límite de entrada superado
          headers:
            Retry-After:
              schema:
                type: integer
components:
  schemas:
    CreateRunRequest:
      type: object
      required: [task, budget, response_contract]
      additionalProperties: false
      properties:
        task:
          type: object
        budget:
          type: object
        response_contract:

```

```
    type: string
Run:
  type: object
  required: [run_id, state, trace_id]
  properties:
    run_id:
      type: string
    state:
      enum: [accepted, queued, running, streaming, succeeded, cancelled, expired,
contract_failed, timed_out]
    trace_id:
      type: string
```

Este YAML no es “documentación bonita”. Sirve para generar clientes, validar compatibilidad, revisar cambios en PR y detectar si una nueva versión rompe consumidores.

Versionar contratos sin romper clientes

Un contrato no es estático. Cambia cuando el producto aprende. El problema no es cambiarlo; el problema es no saber si el cambio rompe a quien lo consume.

Una regla conservadora:

CAMBIO	¿COMPATIBLE?	POR QUÉ
Añadir campo opcional	Normalmente sí.	Los clientes antiguos pueden ignorarlo.
Añadir campo obligatorio	No.	Los clientes antiguos no lo envían o no lo esperan.
Cambiar tipo de un campo	No.	Rompe validadores y clientes generados.
Eliminar campo	No.	Alguien puede depender de él.
Cambiar significado de un estado	No.	Aunque el nombre sea igual, la semántica cambia.
Añadir nuevo estado final	Depende.	Los clientes deben saber qué hacer con estados desconocidos.
Endurecer validación	Puede romper.	Entradas antes aceptadas pasan a fallar.

Por eso conviene versionar contratos como `support-answer@1.2.0` y escribir una política:

VERSIÓN	CUÁNDO SUBIR
Patch	Aclaraciones, ejemplos, descripciones, errores tipográficos.
Minor	Campos opcionales o estados no finales que no rompen clientes preparados.
Major	Cambios obligatorios, tipos distintos, estados finales nuevos o semántica incompatible.

Un buen cliente de runtime también debe ser robusto ante estados desconocidos: si recibe un estado que no entiende, no debería tratarlo como éxito. Mejor mostrar “estado no reconocido”, conservar `run_id` y pedir actualización del contrato.

Streaming: experiencia y operación a la vez

Streaming no significa solo “que se vea escribir”. En un runtime operable, streaming puede emitir eventos de estado:

```
run.accepted
run.queued
run.started
retrieval.completed
model.output_text.delta
model.completed
contract.validated
run.succeeded
```

Esto mejora la experiencia, pero también la operación. Si el usuario ve “buscando documentos” y luego “redactando respuesta”, entiende que el sistema está trabajando. Si soporte ve los mismos eventos en la traza, puede localizar dónde se quedó una run.

El streaming obliga a una decisión de producto: ¿qué puedes mostrar antes de validar? Para respuestas de bajo impacto, quizá puedes mostrar texto parcial. Para salidas que deben cumplir JSON estricto, quizá conviene transmitir eventos de progreso y no enseñar el resultado hasta validar contrato.

Polling, SSE, WebSocket y webhook resuelven problemas distintos:

OPCIÓN	DIRECCIÓN	CUÁNDO ELEGIRLA	COSTE MENTAL
Polling	Cliente pregunta cada cierto tiempo.	Simplicidad y tareas no interactivas.	Fácil, pero puede ser ineficiente.
SSE	Servidor envía eventos al cliente.	Progreso unidireccional y streaming de texto.	Simple para navegador y HTTP.
WebSocket	Comunicación bidireccional.	Interacción viva, cancelación inmediata o colaboración.	Más complejo de operar.
Webhook	Servidor llama a otro sistema.	Integraciones sistema-sistema.	Exige firma, reintentos y entrega verificable.

Para muchos productos de IA, SSE es suficiente: el cliente crea una run, abre `/runs/{run_id}/events` y recibe eventos. WebSocket tiene sentido cuando el cliente también debe enviar señales frecuentes durante la ejecución. Webhook encaja cuando no hay usuario mirando la pantalla y otro sistema debe recibir el resultado.

Observabilidad del runtime

OpenTelemetry define una API de trazas para representar operaciones como spans conectados.¹⁴ Además, sus convenciones semánticas para GenAI están en desarrollo y cubren señales propias de sistemas generativos como operaciones de modelo, agentes y herramientas.¹⁵ W3C Trace Context estandariza cómo propagar contexto de traza entre servicios.¹⁶

Una traza de runtime debería tener, como mínimo:

SPAN	ATRIBUTOS ÚTILES
<code>api.receive</code>	<code>request_id</code> , <code>tenant_id</code> , endpoint, tamaño de payload.
<code>input.validate</code>	contrato, versión, resultado, errores.
<code>run.create</code>	<code>run_id</code> , <code>trace_id</code> , manifest, idempotency key.
<code>queue.wait</code>	cola, prioridad, edad, posición aproximada.
<code>retrieval.search</code>	corpus, versión, <code>top_k</code> , latencia, documentos usados.
<code>tool.call</code>	nombre de herramienta, timeout, resultado, coste si aplica.
<code>model.call</code>	proveedor, modelo, tokens, latencia, request id del proveedor.
<code>output.validate</code>	schema, versión, errores de contrato.
<code>run.close</code>	estado final, coste total, decisión y duración.

Sin estos spans, depurar se convierte en mirar logs sueltos y construir una historia a mano.

En el día a día

Un equipo suele empezar con una función como `answer(question)` . Después llega la realidad: hay usuarios, permisos, documentos, colas, coste, límites y errores. El salto profesional consiste en convertir esa función en un servicio con contrato.

En un proyecto real, el runtime suele tener estas piezas:

PIEZA	DECISIÓN DE INGENIERÍA
API gateway	Qué endpoints existen y cómo autentican.
Run store	Dónde guardamos estado y resultados.
Queue broker	Qué va síncrono y qué va a background.
Workers	Qué tareas ejecuta cada tipo de worker.
Model gateway	Cómo normalizamos proveedores y modelos locales.
Contract validator	Qué salidas pasan o se bloquean.
Event stream	Qué eventos ve el cliente y qué eventos quedan internos.
Observability pipeline	Qué trazas, métricas y logs se conservan.

No hay una única implementación correcta. Puedes hacerlo con FastAPI y Redis, con Node y una cola gestionada, con serverless y colas cloud, o con un backend monolítico bien diseñado. Lo que no cambia es la responsabilidad: entrada clara, estado explícito, colas medibles, contratos validados y trazas.

Por qué debería importarte

Si el runtime está mal diseñado, los errores parecen inexplicables. El usuario dice “se quedó pensando”. Producto dice “a veces no responde”. Ingeniería mira un log y ve una llamada al modelo. Nadie sabe si el problema fue cola, retrieval, proveedor, contrato, timeout o frontend.

Una arquitectura de runtime buena reduce esa ambigüedad. No evita todos los problemas, pero convierte cada problema en una pregunta investigable:

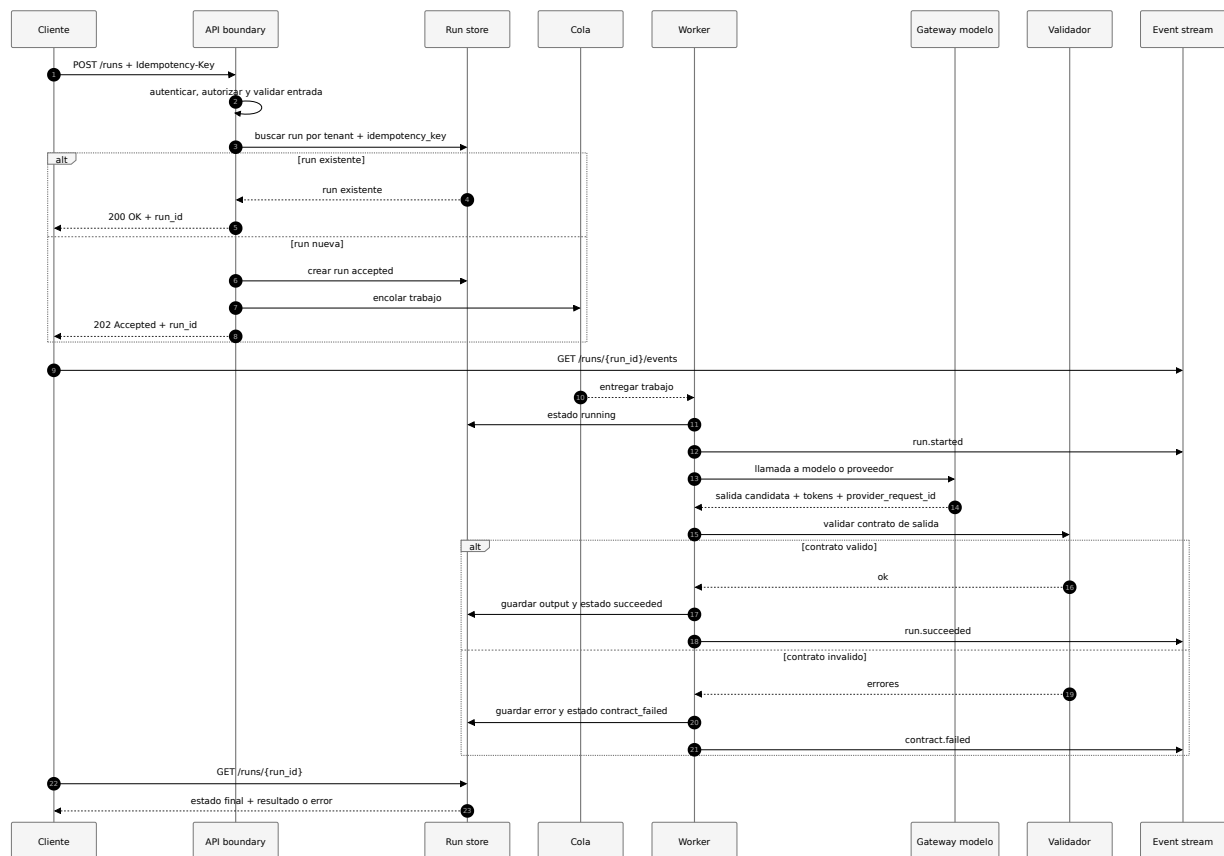
SÍNTOMA	PREGUNTA TÉCNICA
Tarda en empezar	¿TTFT alto, cola larga o retrieval lento?
Responde incompleto	¿Timeout, streaming cortado o contrato rechazado?
Duplica trabajo	¿Falta idempotencia o dedupe de cola?
Coste inesperado	¿Reintentos, contexto largo o routing incorrecto?
No se puede depurar	¿Falta <code>trace_id</code> , spans o estado final?

Dónde solía tropezar yo

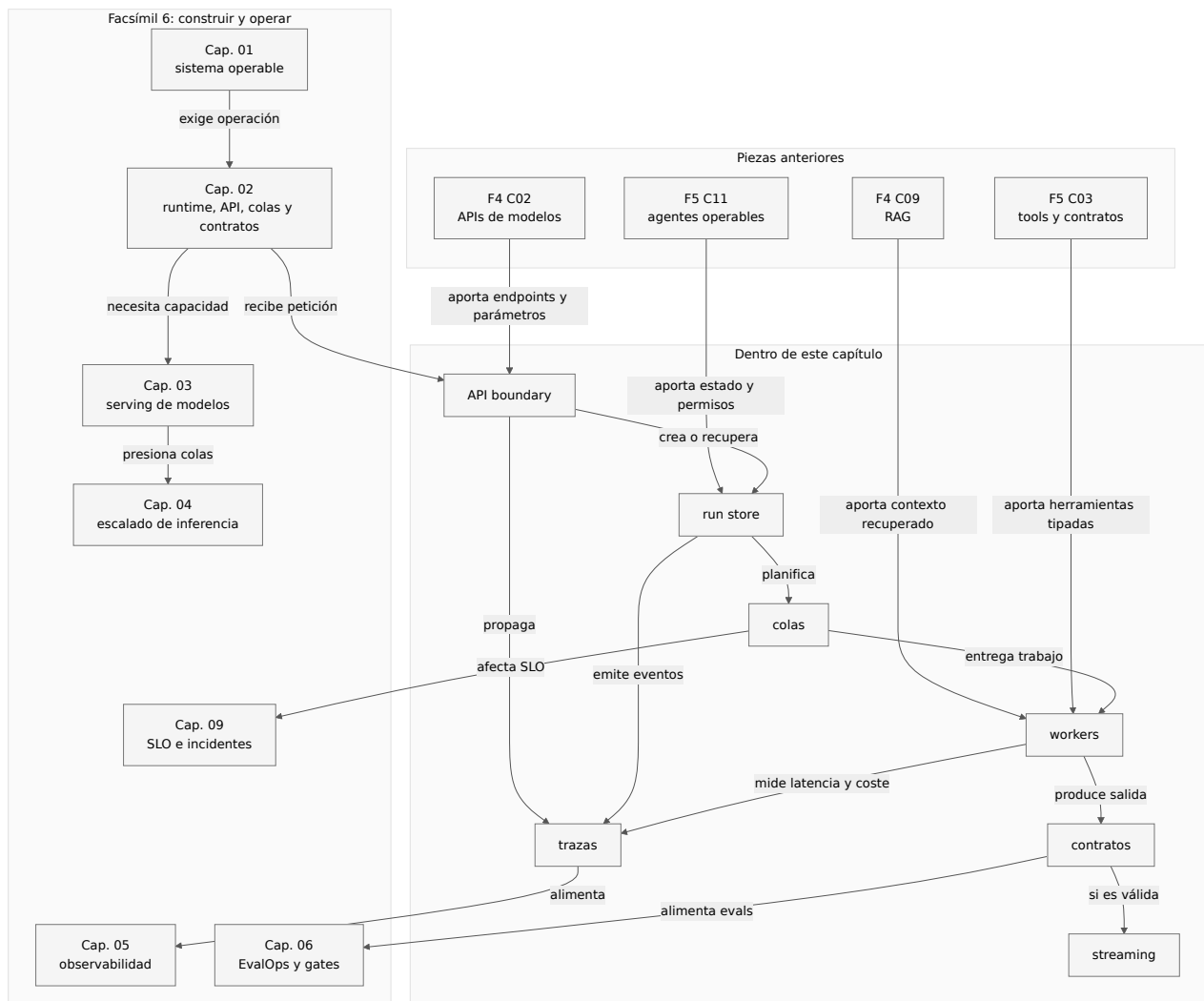
TROPIEZO	POR QUÉ ES UN PROBLEMA	ANTÍDOTO
Empezar por el proveedor	El sistema queda acoplado a un SDK concreto.	Diseñar primero contrato interno y gateway.
No usar idempotencia	Un retry del cliente puede duplicar trabajo y coste.	Exigir <code>idempotency_key</code> en tareas relevantes.
Ver la cola como detalle invisible	La cola puede comerse todo el SLO sin que el modelo sea lento.	Medir profundidad, edad máxima y tiempo de espera.
No diseñar cancelación	El usuario cree que paró algo, pero el sistema sigue gastando.	Separar <code>cancelling</code> , <code>cancelled</code> y fases no cancelables.
Reintentar sin backoff	Los reintentos crean otro pico de carga.	Backoff con jitter, límite de intentos y DLQ.
Mostrar streaming sin validar	El usuario puede ver contenido que luego el contrato rechaza.	Separar eventos de progreso de resultado final validado.
Guardar logs sin estado	Hay texto, pero no reconstrucción causal.	Guardar <code>run_id</code> , estado, eventos y <code>trace_id</code> .
Tratar el contrato como formato	El contrato no solo parsea; decide qué puede entrar y salir.	Versionar entrada, salida y errores.

Cómo encaja todo

Antes del mapa conceptual, conviene ver el flujo temporal. Este diagrama no muestra todas las ramas, pero sí la ruta típica de una run que entra, espera, ejecuta, valida y emite eventos:



Lo importante está en los puntos de control: la API no acepta cualquier cosa, el run store evita duplicados, la cola desacopla entrada y ejecución, el worker actualiza estado, el modelo queda detrás de un gateway y el validador decide si la salida puede llegar al producto.



La conexión importante es esta: el runtime no es una pieza aislada. Es donde se encuentran las APIs del facsímil 4, los agentes del facsímil 5 y la operación que estamos construyendo ahora.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
API boundary	Borde donde una petición externa se convierte en una petición interna validada.
Autenticación	Comprobación de identidad: quién llama al sistema.
Autorización	Comprobación de permiso: qué puede hacer esa identidad.
Tenant	Organización, cliente o espacio lógico al que pertenece una run.
Idempotency key	Clave que identifica una intención repetible para evitar duplicados.
Run store	Almacén donde vive el estado de cada ejecución.
At-most-once	Semántica de entrega donde un trabajo se procesa como máximo una vez, con riesgo de pérdida.
At-least-once	Semántica donde un trabajo puede procesarse más de una vez, por lo que exige idempotencia.
Queue depth	Número de trabajos esperando en una cola.
Backpressure	Mecanismo para frenar entrada cuando el sistema no puede absorber más trabajo.
Dead-letter queue	Cola separada donde se envían trabajos que fallan repetidamente para no bloquear la cola principal.
Backoff	Espera creciente entre reintentos para no saturar una dependencia.
Fairness por tenant	Reparto de capacidad para que un tenant no consuma todo el runtime.
Circuit breaker	Patrón que deja de llamar temporalmente a una dependencia cuando acumula fallos.
Bulkhead	Separación de recursos para que una parte lenta no arrastre a todo el sistema.
Polling	Patrón donde el cliente consulta periódicamente el estado de una run.
SSE	Server-Sent Events: canal HTTP donde el servidor envía eventos unidireccionales al cliente.
WebSocket	Canal persistente bidireccional entre cliente y servidor.
Webhook	Callback HTTP que recibe el resultado o cambio de estado cuando termina una run.

TÉRMINO	DEFINICIÓN
OpenAPI	Especificación que describe rutas, métodos, esquemas, respuestas y seguridad de una API HTTP.
Worker	Proceso que toma un trabajo y ejecuta una fase.
Timeout budget	Presupuesto de tiempo repartido entre fases.
Contract validator	Componente que acepta o rechaza una entrada o salida según esquema.
TTFT	Tiempo hasta recibir el primer token o primer evento útil.
Trace context	Contexto que permite conectar spans entre servicios.
Goodput	Trabajo correcto dentro de SLO por unidad de tiempo.

Antes de pasar página

- ☐ ¿Puedes explicar por qué una request HTTP no es lo mismo que una run?
- ☐ ¿Puedes diseñar una entrada con `request_id` , `idempotency_key` , presupuesto y contrato?
- ☐ ¿Puedes explicar qué validan autenticación, autorización y tenancy?
- ☐ ¿Puedes elegir entre `200 OK` , `202 Accepted` , polling, streaming, background job y webhook?
- ☐ ¿Puedes diseñar endpoints mínimos para crear, consultar, cancelar y seguir eventos de una run?
- ☐ ¿Puedes proponer una persistencia mínima con `runs` , `run_events` , `run_outputs` y `run_errors` ?
- ☐ ¿Puedes calcular T_{total} sumando fases de runtime?
- ☐ ¿Puedes usar $L = \lambda W$ para explicar por qué crece una cola?
- ☐ ¿Puedes distinguir at-most-once y at-least-once, y justificar por qué el worker debe ser idempotente?
- ☐ ¿Puedes explicar para qué sirven TTL, dead-letter queue, backoff y fairness por tenant?
- ☐ ¿Puedes explicar cuándo usar circuit breaker o bulkhead?
- ☐ ¿Puedes decir qué spans mínimos debería tener una traza de runtime?
- ☐ ¿Puedes distinguir contrato de entrada, contrato de salida y contrato de error?

- ☐ ¿Puedes decir qué cambios de contrato son compatibles y cuáles rompen clientes?
- ☐ ¿Puedes explicar qué aporta OpenAPI además de JSON Schema?
- ☐ ¿Puedes decidir cuándo usar streaming de progreso y cuándo esperar a validar salida?
- ☐ ¿Puedes explicar qué haría el sistema si se repite una petición con la misma clave de idempotencia?
- ☐ ¿Puedes convertir un fallo de contrato en una respuesta operativa útil para cliente y soporte?

En resumen

IDEA	QUÉ DEBES LLEVARTE
El runtime coordina, no solo llama al modelo.	Una petición real necesita API, estado, colas, contratos, workers, streaming y trazas.
El patrón de API cambia la experiencia.	No es igual <code>200 OK</code> que <code>202 Accepted</code> , polling, streaming, webhook o background job.
La cola forma parte del producto.	Si no mides espera, profundidad y TTL, la latencia aparece como misterio.
Los contratos protegen el sistema.	Entrada, salida y error deben estar versionados y validados.
La run es la unidad operativa.	<code>run_id</code> , <code>trace_id</code> , estado y eventos permiten reconstruir qué pasó.
La arquitectura prepara los siguientes capítulos.	Serving, escalado, observabilidad y EvalOps dependen de este runtime.

Para saber más

- Anthropic. (2026). *Messages API*. <https://platform.claude.com/docs/en/api/messages>
- Anthropic. (2026). *Streaming Messages*. <https://platform.claude.com/docs/en/api/streaming>
- Fielding, R. T., Nottingham, M. y Reschke, J. (2022). *HTTP Semantics*. RFC 9110. <https://datatracker.ietf.org/doc/html/rfc9110>
- Hohpe, G. y Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.
- JSON Schema. (2020). *JSON Schema Validation: A Vocabulary for Structural Validation of JSON*. <https://json-schema.org/draft/2020-12/json-schema-validation>

- Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.
- Little, J. D. C. (1961). *A Proof for the Queuing Formula: $L = \lambda W$* . Operations Research, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>
- OpenAI. (2026). *Create a model response*. <https://developers.openai.com/api/reference/resources/responses/methods/create>
- OpenAI. (2026). *Streaming API responses*. <https://developers.openai.com/api/docs/guides/streaming-responses>
- OpenAI. (2026). *Structured model outputs*. <https://developers.openai.com/api/docs/guides/structured-outputs>
- OpenAPI Initiative. (2025). *OpenAPI Specification*. <https://spec.openapis.org/oas/latest.html>
- OpenTelemetry. (2026). *Semantic conventions for generative AI systems*. <https://opentelemetry.io/docs/specs/semconv/gen-ai/>
- OpenTelemetry. (2026). *Tracing API*. <https://opentelemetry.io/docs/specs/otel/trace/api/>
- World Wide Web Consortium. (2021). *Trace Context Level 2*. <https://www.w3.org/TR/trace-context-2/>

Notas

1. Fielding, R. T., Nottingham, M. y Reschke, J. (2022). *HTTP Semantics*. RFC 9110. <https://datatracker.ietf.org/doc/html/rfc9110>. Consultado el 27 de mayo de 2026. ↵
2. Little, J. D. C. (1961). *A Proof for the Queuing Formula: $L = \lambda W$* . Operations Research, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>. ↵
3. OpenAI. (2026). *Create a model response*. <https://developers.openai.com/api/reference/resources/responses/methods/create>. Consultado el 27 de mayo de 2026. ↵
4. OpenAI. (2026). *Streaming API responses*. <https://developers.openai.com/api/docs/guides/streaming-responses>. Consultado el 27 de mayo de 2026. ↵
5. OpenAI. (2026). *Structured model outputs*. <https://developers.openai.com/api/docs/guides/structured-outputs>. Consultado el 27 de mayo de 2026. ↵
6. Anthropic. (2026). *Messages API*. <https://platform.claude.com/docs/en/api/messages>. Consultado el 27 de mayo de 2026. ↵
7. Anthropic. (2026). *Streaming Messages*. <https://platform.claude.com/docs/en/api/streaming>. Consultado el 27 de mayo de 2026. ↵

8. Little, J. D. C. (1961). A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research*, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>. Demuestra la relación con total generalidad, sin hipótesis sobre la distribución de llegadas o de servicio. ↵
9. Kleinrock, L. (1975). *Queueing Systems, Volume 1: Theory*. Wiley. Deriva, para la cola M/M/1, el tiempo medio de espera en función de la utilización del servidor. ↵
10. Kingman, J. F. C. (1961). The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society*, 57(4), 902-904. <https://doi.org/10.1017/S0305004100036094>. Aproximación de la espera en tráfico intenso para colas generales G/G/1. ↵
1. La fórmula de Pollaczek-Khinchine para la cola M/G/1 se debe a los trabajos independientes de Pollaczek (1930) y Khinchine (1932); su tratamiento estándar aparece en Kleinrock, L. (1975). *Queueing Systems, Volume 1: Theory*. Wiley. ↵
2. JSON Schema. (2020). *JSON Schema Validation: A Vocabulary for Structural Validation of JSON*. <https://json-schema.org/draft/2020-12/json-schema-validation>. Consultado el 27 de mayo de 2026. ↵
3. OpenAPI Initiative. (2025). *OpenAPI Specification*. <https://spec.openapis.org/oas/latest.html>. Consultado el 27 de mayo de 2026. ↵
4. OpenTelemetry. (2026). *Tracing API*. <https://opentelemetry.io/docs/specs/otel/trace/api/>. Consultado el 27 de mayo de 2026. ↵
5. OpenTelemetry. (2026). *Semantic conventions for generative AI systems*. <https://opentelemetry.io/docs/specs/semconv/gen-ai/>. Consultado el 27 de mayo de 2026. ↵
6. World Wide Web Consortium. (2021). *Trace Context Level 2*. <https://www.w3.org/TR/trace-context-2/>. Consultado el 27 de mayo de 2026. ↵

Capítulo 03: Serving de modelos: workers, batching y capacidad

Qué deberías poder hacer al terminar

En el capítulo anterior diseñamos la entrada de una run: API, estado, cola, contrato, streaming y trazas. Ahora miramos la pieza que suele decidir si el sistema aguanta o se rompe: el **serving de modelos**.

Serving significa mantener modelos disponibles para responder peticiones reales. No es “tener un modelo descargado”. Es cargarlo, reservar memoria, aceptar tráfico, agrupar peticiones, repartir trabajo entre workers, medir latencia, controlar colas, limitar coste y decidir cuándo escalar.

Al terminar, deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Separar runtime de producto y runtime de modelo.	Distingues API boundary, cola de runs, scheduler, engine de inferencia y workers.
Explicar prefill, decode y KV cache.	Puedes decir qué fase consume contexto, qué fase genera tokens y por qué crece la memoria.
Estimar memoria de un modelo servido.	Calculas pesos, KV cache, batch, contexto y margen operativo.
Leer métricas de serving.	Distingues TTFT, TPOT, throughput, cola, tokens de entrada, tokens de salida y goodput.
Dimensionar capacidad inicial.	Puedes justificar réplicas, concurrencia, cola máxima y señal de escalado.
Elegir entre proveedor cloud, runtime propio y modo híbrido.	Relacionas control, latencia, coste, privacidad, complejidad y mantenimiento.

La idea central: **el modelo no “vive” en tu API; vive en una máquina de inferencia con memoria, scheduler, colas internas y límites físicos.**

La demo que iba perfecta en mi portátil

Una demo suele empezar así: hacemos una llamada al modelo, esperamos unos segundos y la respuesta aparece. Si la demo la usa una persona, todo parece razonable. Si la usan cincuenta a la vez, aparece otra realidad: algunas peticiones esperan, otras tardan en empezar, unas consumen mucho contexto, otras generan muchas palabras y la GPU ya no trabaja como una calculadora aislada, sino como una estación compartida.

Ese cambio de escala altera la conversación. Ya no preguntamos solo “¿qué modelo responde mejor?”. Preguntamos “¿cuántas runs por minuto podemos atender?”, “¿cuánto tarda el primer token?”, “¿cuánta memoria se come el contexto?”, “¿qué ocurre si entran muchas peticiones largas?” y “¿qué medimos para saber si debemos escalar?”.

Por eso este capítulo se coloca justo después de API y colas. La cola puede aceptar trabajo, pero alguien tiene que ejecutarlo. Ese alguien no es una abstracción: es el sistema de serving.

Qué no es servir un modelo

Servir un modelo no es ejecutar un notebook. Un notebook ayuda a explorar, pero no resuelve concurrencia, colas, memoria, métricas, límites por usuario, reinicios ni despliegue.

Tampoco es simplemente exponer un endpoint compatible con OpenAI. La compatibilidad de API puede ser comodísima, pero no te dice si el runtime soporta tu modelo, si gestiona bien KV cache, si hace batching continuo, si el streaming respeta tu contrato o si la latencia en p95 entra en tu SLO.

Y tampoco es “usar GPU” sin más. La GPU puede estar cargada y aun así ofrecer mal servicio: quizá está llena de peticiones largas, quizá la KV cache fragmenta memoria, quizá el batch está mal configurado, quizá faltan réplicas, quizá el cuello está en cola o quizá el modelo está generando demasiados tokens.

CONFUSIÓN	LO QUE FALTA
“Tengo el modelo descargado”	Falta runtime, memoria reservada, endpoint, scheduler y métricas.
“Responde en local”	Falta concurrencia, contratos, colas, observabilidad y operación.
“La GPU está al 95%”	Falta saber si produce tokens útiles dentro de SLO.
“Es compatible con OpenAI”	Falta verificar streaming, errores, tools, JSON, límites y parámetros soportados.
“Añado otra réplica y ya está”	Falta comprobar cuello real: cola, memoria, red, CPU, disco o modelo.

Qué sí es el serving de modelos

El serving de modelos es la capa que convierte capacidad de cómputo en inferencias disponibles para usuarios o sistemas. Mantiene el modelo cargado, decide qué peticiones entran juntas, ejecuta prefill y decode, conserva KV cache, emite tokens, expone métricas y devuelve resultados bajo un contrato.

Fecha de corte: 27 de mayo de 2026. Fuentes consultadas ese día: documentación estable de vLLM, servidor OpenAI-compatible de vLLM, documentación de SGLang, documentación de NVIDIA TensorRT-LLM, documentación y métricas de Hugging Face Text Generation Inference, MLPerf Inference, Kubernetes HPA, Prometheus y OpenTelemetry GenAI. Lo estable es el mecanismo: prefill, decode, KV cache, batching, colas, paralelismo, métricas y capacidad. Lo cambiante son versiones, flags, modelos soportados, aceleradores disponibles, límites de proveedor y nombres exactos de métricas.

Los runtimes modernos de inferencia no son envoltorios triviales. vLLM se presenta como una librería de inferencia y serving con servidor compatible con OpenAI, streaming, paralelismos y soporte de muchas arquitecturas.¹ SGLang se describe como framework de serving de alto rendimiento para modelos de lenguaje y multimodales, con foco en baja latencia, alto throughput, RadixAttention, prefix caching y paralelismo multi-GPU.² NVIDIA TensorRT-LLM ofrece APIs y runtimes para ejecutar engines optimizados en GPUs NVIDIA, con documentación sobre arquitectura, runtimes, cuantización y soporte de despliegue.³ Hugging Face TGI documenta serving, streaming por SSE, tensor parallelism, Prometheus/OpenTelemetry, continuous batching y métricas exportadas.⁴

No hace falta memorizar todos esos nombres. Sí hace falta entender qué problema resuelven: **servir tokens de forma concurrente sin desperdiciar memoria ni perder control operativo.**

El mecanismo por dentro: de la run al token

Una run que llega desde nuestra API no entra directamente al modelo. Normalmente pasa por varias capas:

1. **Gateway de modelo:** traduce el contrato interno al formato del runtime o proveedor.
2. **Scheduler:** decide cuándo entra la petición en ejecución.
3. **Prefill:** procesa tokens de entrada y crea la KV cache inicial.
4. **Decode:** genera tokens de salida de forma iterativa.
5. **Streaming:** entrega deltas o eventos si el producto lo permite.
6. **Métricas:** mide latencia, tokens, cola, memoria, errores y resultado.
7. **Cierre:** devuelve salida al runtime de producto para validación y persistencia.

La división entre prefill y decode es fundamental.

FASE	QUÉ HACE	QUÉ SUELE DOLER
Prefill	Lee todos los tokens de entrada y calcula las representaciones iniciales.	Contextos largos, documentos grandes, prompts con mucho historial.
Decode	Genera tokens de salida uno a uno, reutilizando KV cache.	Salidas largas, muchos usuarios concurrentes, baja velocidad por token.
KV cache	Guarda claves y valores de atención para no recalcular todo el contexto.	Memoria de GPU, fragmentación, límites de batch y contexto.

En el [facsimilar 3, capítulo 03](#) vimos Q, K y V como piezas de atención. En el [facsimilar 4, capítulo 03](#) vimos tokens, contexto y caché. Aquí juntamos ambas ideas desde operación: cada token de contexto puede dejar rastro en memoria, y cada token generado necesita usar esa memoria una y otra vez.

La latencia de una petición de generación se puede aproximar así:

$$T_{\text{run}} = T_{\text{cola\_producto}} + T_{\text{gateway}} + T_{\text{cola\_serving}} + T_{\text{prefill}} + N_{\text{out}} \cdot T_{\text{decode/token}} + T_{\text{validación}}$$

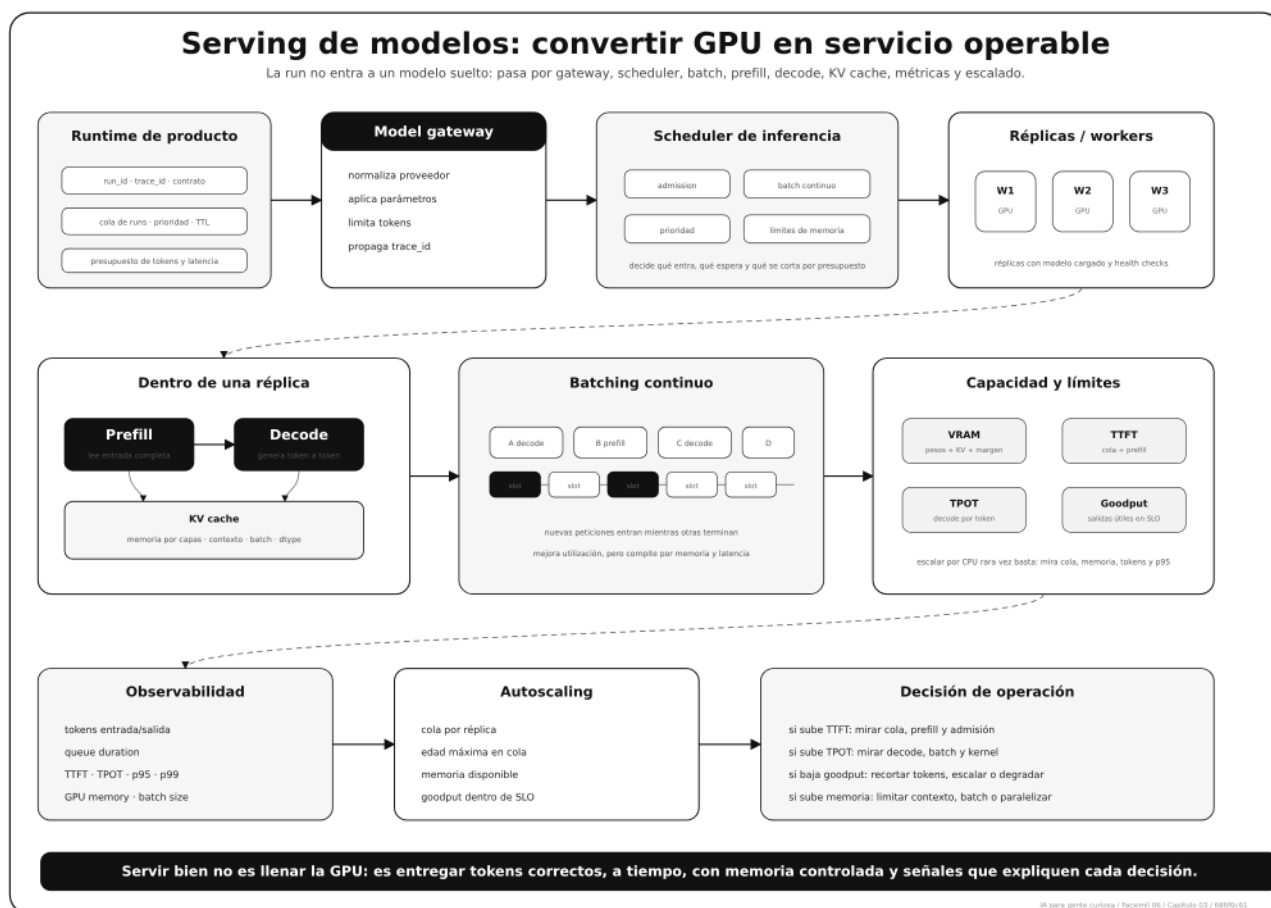
SÍMBOLO	SIGNIFICADO	EJEMPLO
$T_{\text{cola\_producto}}$	Espera en la cola de runs del producto.	350 ms
T_{gateway}	Traducción, autenticación interna y envío al runtime de modelo.	60 ms
$T_{\text{cola\_serving}}$	Espera dentro del engine de inferencia.	240 ms
T_{prefill}	Procesamiento de tokens de entrada.	520 ms
N_{out}	Número de tokens generados.	180 tokens
$T_{\text{decode/token}}$	Tiempo medio por token generado.	18 ms/token
$T_{\text{validación}}$	Validación de contrato y cierre.	90 ms

Con esos números:

$$T_{\text{run}} = 350 + 60 + 240 + 520 + 180 \cdot 18 + 90 = 4500 \text{ ms}$$

Esto explica algo que se ve mucho en producto: dos peticiones al mismo modelo no tardan lo mismo. Una pregunta corta con salida breve puede volar. Un informe con diez documentos y salida larga puede multiplicar prefill, KV cache y decode.

Anatomía visual de un servicio de inferencia



La figura muestra una separación que conviene respetar: el runtime de producto sabe de runs, contratos, permisos y trazas. El runtime de modelo sabe de batch, memoria, tokens y kernels. Si mezclamos ambas responsabilidades, acabamos con un sistema difícil de depurar: producto habla de usuarios y estados; inferencia habla de tokens, colas internas y GPU.

Memoria: el límite que aparece antes de que lo esperes

En serving, la memoria no la ocupan solo los pesos del modelo. Hay al menos cuatro bolsas:

BOLSA DE MEMORIA	QUÉ CONTIENE	DE QUÉ DEPENDE
Pesos	Parámetros del modelo cargados en GPU o CPU.	Número de parámetros y precisión.
KV cache	Claves y valores de atención por token activo.	Capas, cabezas KV, dimensión, contexto, batch y dtype.
Activaciones y buffers	Tensores temporales del runtime.	Kernels, batch, backend, paralelismo y optimizaciones.
Overhead operativo	Fragmentación, allocator, runtime, CUDA, servidor, margen.	Implementación y configuración.

Una fórmula útil, aunque aproximada:

$$M_{total} \approx M_{pesos} + M_{KV} + M_{buffers} + M_{margen}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
M_{pesos}	Memoria ocupada por pesos del modelo.	14 GiB
M_{KV}	Memoria de KV cache para secuencias activas.	16 GiB
$M_{buffers}$	Memoria temporal de kernels, logits, activaciones y runtime.	3 GiB
M_{margen}	Margen para fragmentación y variabilidad.	5 GiB

Con esos valores:

$$M_{total} \approx 14 + 16 + 3 + 5 = 38 \text{ GiB}$$

En una GPU de 40 GiB, eso parece caber. En operación, “parece” no basta. Una petición más larga, un batch mayor, un modelo con más capas o un runtime que reserve buffers distintos puede llevarte al límite. Por eso capacidad no se calcula con una media bonita, sino con escenarios.

Para pesos:

$$M_{pesos} = \frac{P \cdot b}{S}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
P	Número de parámetros.	7.000 millones
b	Bytes por parámetro.	2 bytes en BF16/FP16
S	Número de shards si hay tensor parallel.	1

Ejemplo:

$$M_{\text{pesos}} = \frac{7.000.000.000 \cdot 2}{1} \approx 14 \text{ GB}$$

Si el modelo se reparte en dos GPUs con tensor parallel, cada GPU no carga necesariamente “la mitad exacta de todo”, porque hay buffers y comunicación, pero la intuición de pesos por shard ayuda:

$$M_{\text{pesos_por_GPU}} \approx \frac{14}{2} = 7 \text{ GB}$$

La KV cache se puede aproximar así:

$$M_{\text{KV}} = 2 \cdot L_{\text{capas}} \cdot H_{\text{KV}} \cdot D_{\text{head}} \cdot B_{\text{dtype}} \cdot T_{\text{activos}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
2	Guardamos K y V.	2 tensores
L_{capas}	Número de capas Transformer.	32
H_{KV}	Número de cabezas KV.	8
D_{head}	Dimensión de cada cabeza.	128
B_{dtype}	Bytes por valor.	2 bytes
T_{activos}	Tokens activos en el batch: entrada + salida generada.	16 secuencias x 8192 tokens

Con batch 16 y 8192 tokens activos por secuencia:

$$T_{\text{activos}} = 16 \cdot 8192 = 131.072$$

$$M_{\text{KV}} = 2 \cdot 32 \cdot 8 \cdot 128 \cdot 2 \cdot 131.072 = 17.179.869.184 \text{ bytes} \approx 16 \text{ GiB}$$

Esta cuenta no pretende sustituir al profiler. Pretende darte criterio. Si duplicas contexto, sube KV. Si duplicas batch, sube KV. Si usas más capas o más cabezas KV, sube KV. Si reduces precisión de KV, puede bajar memoria, pero debes medir calidad y estabilidad.

El paper de PagedAttention nace precisamente de este problema: la KV cache de cada request es grande, crece y decrece dinámicamente, y una gestión ineficiente desperdicia memoria por fragmentación o duplicación. Sus autores comparan la idea con paginación de memoria de sistemas operativos y reportan mejoras de throughput al construir vLLM sobre ese enfoque.⁵

Batching: no todas las peticiones avanzan al mismo ritmo

En un servidor web clásico, hacer batch puede sonar extraño: cada petición es independiente. En inferencia de LLMs, agrupar peticiones puede mejorar muchísimo el uso de GPU, pero tiene matices.

El batch estático agrupa peticiones y las ejecuta juntas. El problema es que las generaciones no terminan a la vez. Si una petición necesita 40 tokens y otra 800, la corta puede quedar esperando a que la larga termine, o el runtime desperdicia huecos.

El batching continuo intenta resolverlo: en cada paso de generación, el scheduler recompone el conjunto de secuencias activas. Cuando una termina, entra otra. Hugging Face TGI documenta continuous batching como una optimización para aumentar throughput, junto con streaming, tensor parallelism y métricas de producción.⁶

La intuición:

ENFOQUE	VENTAJA	COSTE
Sin batching	Simple y fácil de razonar.	GPU infrautilizada con tráfico concurrente.
Batch fijo	Mejor uso de GPU.	Esperas por secuencias largas y lotes mal equilibrados.
Batching continuo	Mejor ocupación con peticiones variables.	Scheduler más complejo y presión sobre KV cache.
Prefix caching	Reutiliza prefijos compartidos.	Requiere detectar y gestionar prefijos con cuidado.
Chunked prefill	Trocea entradas largas para no bloquear decode.	Añade complejidad de planificación.

Una métrica clave aquí es la diferencia entre **throughput bruto** y **goodput**:

$$\text{throughput} = \frac{\text{tokens generados}}{\text{segundos}}$$

$$\text{goodput} = \frac{\text{tokens de respuestas válidas dentro de SLO}}{\text{segundos}}$$

MÉTRICA	QUÉ CUENTA	QUÉ PUEDE OCULTAR
Throughput	Tokens por segundo, sin mirar si llegaron a tiempo.	Puede ser alto con usuarios esperando demasiado.
Goodput	Tokens útiles dentro del objetivo de servicio.	Exige definir qué significa “útil” y qué SLO aplica.
TTFT	Tiempo hasta primer token.	Puede ser bueno aunque el final tarde demasiado.
TPOT	Tiempo por token durante decode.	No incluye espera en cola ni validación final.
p95/p99	Cola de latencias altas.	La media puede parecer bien mientras p99 duele.

Dean y Barroso explicaron en *The Tail at Scale* que, en sistemas grandes, la cola de latencias altas importa mucho porque muchas operaciones dependen de varias piezas y basta una pieza lenta para que la experiencia final se degrade.⁷ En IA esto se nota enseguida: el usuario no vive la media, vive su petición.

Workers: una réplica no es solo un proceso

Un worker de serving suele ser una réplica con modelo cargado y capacidad asignada. Puede vivir en una GPU completa, en varias GPUs, en CPU para modelos pequeños, en una partición de acelerador o detrás de un proveedor cloud.

Lo importante es que cada worker tiene un contrato operativo:

PROPIEDAD	PREGUNTA CONCRETA
Modelo servido	¿Qué <code>model_id</code> , versión, cuantización y plantilla de chat carga?
Capacidad	¿Cuántas secuencias concurrentes acepta sin romper SLO?
Memoria	¿Cuánta VRAM queda libre en escenarios largos?
Contexto máximo	¿Qué límite real de entrada y salida se permite en producto?
Parámetros admitidos	¿Acepta JSON, tools, logprobs, reasoning, imágenes o solo texto?
Health check	¿Cómo sabemos que está vivo y realmente puede generar?
Warmup	¿Se calienta el modelo antes de aceptar tráfico?
Drenaje	¿Cómo deja de aceptar runs antes de reiniciar?

El warmup merece una línea propia. Una réplica puede responder al health check HTTP y aun así estar fría: modelo no cargado, kernels no inicializados o primera petición más lenta. Por eso muchos equipos separan:

CHECK	QUÉ COMPRUEBA
Liveness	El proceso no está colgado.
Readiness	Puede aceptar tráfico.
Model readiness	El modelo está cargado, inicializado y con memoria suficiente.
Synthetic generation	Puede generar una respuesta pequeña y medir latencia básica.

El drenaje también importa. Si matas una réplica con muchas secuencias activas, puedes cortar streams o perder trabajo. Un shutdown correcto suele marcar la réplica como no lista, deja de recibir nuevas peticiones, termina las activas dentro de un límite y luego se apaga.

Runtime propio, proveedor cloud o mezcla

No todo sistema necesita servir modelos propios. A veces usar una API cloud es lo correcto. Otras veces necesitas modelos locales, runtimes propios o una mezcla.

OPCIÓN	QUÉ CONTROLAS	QUÉ DELEGAS	CUÁNDO ENCAJA
API de proveedor	Prompt, parámetros, contrato de producto, evals y routing.	Modelo, GPU, scheduling interno y capacidad base.	Producto que prioriza velocidad de integración y operación simple.
Runtime propio	Modelo, versión, cuantización, hardware, scheduler y datos.	Nada esencial de inferencia, salvo cloud si alquilas GPU.	Necesitas control de pesos, coste, latencia, región o investigación aplicada.
OpenAI-compatible local	Contrato parecido a APIs conocidas.	Depende del runtime concreto.	Quieres cambiar proveedor por runtime propio con menos fricción.
Híbrido	Rutas cloud y rutas propias según tarea.	Parte de la complejidad se reparte.	Quieres fallback, coste controlado o separación por sensibilidad/tamaño.
Batch offline	Ejecución diferida, no interactiva.	Experiencia en tiempo real.	Grandes volúmenes sin presión de latencia interactiva.

MLPerf Inference sirve como referencia de benchmarking de inferencia en datacenter, con resultados comparables bajo reglas específicas.⁸ No sustituye tus pruebas, pero recuerda algo sano: medir inferencia exige protocolo. Si comparas runtimes con prompts distintos, batch distinto, contexto distinto y SLO distinto, no estás comparando.

Dimensionar capacidad con números

Capacidad empieza con una pregunta concreta: ¿qué carga queremos soportar y con qué objetivo?

Ejemplo:

VARIABLE DE CARGA	VALOR
Llegadas medias	2 runs/s
Pico esperado	8 runs/s durante 5 minutos
Tokens de entrada p50	1.200
Tokens de entrada p95	8.000
Tokens de salida p50	180
Tokens de salida p95	700
SLO interactivo	p95 menor o igual a 6 s
Error budget mensual	2% de runs fuera de SLO

Con la ley de Little, que ya usamos en el capítulo anterior:

$$L = \lambda W$$

Si llegan $\lambda = 8$ runs/s en pico y queremos que el tiempo medio dentro del sistema sea $W = 4$ s:

$$L = 8 \cdot 4 = 32 \text{ runs concurrentes}$$

Pero eso no significa “necesito 32 GPUs”. Significa que el sistema completo tendrá unas 32 runs en curso: algunas en cola, algunas en prefill, algunas en decode, algunas validando. Ahora hay que saber cuántas puede sostener cada réplica sin romper memoria ni p95.

Una aproximación inicial de réplicas:

$$R = \left\lceil \frac{\lambda_{\text{pico}} \cdot T_{\text{serving,p95}}}{C_{\text{réplica}} \cdot U_{\text{objetivo}}} \right\rceil$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
R	Réplicas necesarias.	?
λ_{pico}	Llegadas pico.	8 runs/s
$T_{\text{serving,p95}}$	Tiempo de serving p95 por run.	4 s
$C_{\text{réplica}}$	Concurrencia útil por réplica.	10 runs
U_{objetivo}	Utilización objetivo, dejando margen.	0,70

$$R = \left\lceil \frac{8 \cdot 4}{10 \cdot 0,70} \right\rceil = \lceil 4,57 \rceil = 5$$

Esta fórmula no te da la verdad. Te da una hipótesis para probar. Después haces load test con distribución realista de tokens y miras p50, p95, p99, memoria y coste.

Cuántos workers para que casi nadie espere: Erlang C. La pregunta «¿cuántas réplicas necesito?» es la misma que las centralitas telefónicas resolvieron hace un siglo. La fórmula de Erlang C da la probabilidad de que una petición tenga que esperar dado el número de servidores y la carga ofrecida, y de ahí se despeja cuántos servidores hacen falta para que esa probabilidad sea pequeña.⁹

$$P_{\text{espera}} = \frac{\frac{a^c}{c!} \frac{c}{c-a}}{\sum_{k=0}^{c-1} \frac{a^k}{k!} + \frac{a^c}{c!} \frac{c}{c-a}}, \quad a = \lambda \mathbb{E}[S]$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
a	Carga ofrecida en erlangs (llegadas por duración media).	$20 \times 0,4 = 8$ erlangs.
c	Número de servidores o réplicas.	12
$\mathbb{E}[S]$	Duración media de una petición.	0,4 segundos.
P_{espera}	Probabilidad de que una petición tenga que esperar.	Baja al subir c por encima de a .

En palabras: necesitas, como mínimo, más servidores que erlangs de carga ($c > a$), o la cola es infinita. Pero para que casi nadie espere hace falta un margen por encima de ese mínimo, y Erlang C dice exactamente cuánto. Es la diferencia entre dimensionar con una cuenta y dimensionar a ojo y rezar en la hora punta.

Cuánto ganas al paralelizar: la ley de Amdahl. Repartir el trabajo entre más núcleos o réplicas no acelera lo que es inherentemente secuencial. Amdahl puso número a ese techo: la aceleración máxima la fija la fracción del trabajo que no se puede paralelizar.¹⁰

$$S(N) = \frac{1}{(1 - p) + \frac{p}{N}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
p	Fracción del trabajo paralelizable.	0,9
N	Número de unidades de proceso.	16
$S(N)$	Aceleración respecto a una sola unidad.	Tiende a $1/(1 - p) = 10$ aunque $N \rightarrow \infty$.

En palabras: si un 10% del trabajo es secuencial (validación, serialización, una escritura en base de datos), nunca irás más de diez veces más rápido por mucho hardware que añadas. En serving esto recuerda que el prefill, el postproceso o un validador caro pueden ser el techo real, y que comprar más GPUs no lo mueve.

Por qué a veces más máquinas rinden menos: la ley universal de escalabilidad.

Amdahl ignora un coste que en sistemas reales es decisivo: la coordinación entre nodos. La ley universal de escalabilidad añade ese término y predice algo que todo operador ha visto, que pasado cierto punto añadir nodos baja el rendimiento.¹¹

$$C(N) = \frac{N}{1 + \alpha (N - 1) + \beta N (N - 1)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
N	Número de nodos o réplicas.	32
α	Contención: parte que se serializa por recursos compartidos.	0,05
β	Coherencia: coste de mantener los nodos sincronizados.	0,001
$C(N)$	Capacidad relativa frente a un nodo.	Tiene un máximo y luego decrece.

En palabras: con coordinación cero ($\beta = 0$) recuperas Amdahl; con coordinación real, la curva sube, se aplana y baja. Hay un número óptimo de réplicas, y pasarse cuesta dinero y latencia. Medir α y β con dos o tres puntos de carga evita el «escalado horizontal infinito» que

casi nunca existe.

Por qué la cola larga manda a escala: el «tail at scale». Si responder a un usuario toca muchos servicios (retrieval, reranker, varias herramientas), basta que uno vaya lento para que la respuesta entera vaya lenta. La probabilidad de tocar al menos una respuesta lenta crece con el número de servicios implicados.¹²

$$P(\text{lenta}) = 1 - (1 - p)^n$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
p	Probabilidad de que un servicio responda lento (su cola).	0,01 (el p99).
n	Número de servicios que deben responder.	100
$P(\text{lenta})$	Probabilidad de que la petición completa sea lenta.	$1 - 0,99^{100} \approx 0,63$.

En palabras: con cada componente fallando el 1% de las veces en latencia, tocar 100 de ellos hace que casi dos de cada tres peticiones completas sean lentas. Por eso a escala se atacan los percentiles altos (p99) y se usan técnicas como peticiones cubiertas o con plazo: la media no te salva cuando dependes de muchas piezas.

Señales de autoscaling

Kubernetes HPA ajusta réplicas a partir de métricas observadas, como CPU, memoria o métricas personalizadas.¹³ Para IA, CPU no suele bastar. Puedes tener CPU tranquila y GPU saturada, KV cache llena o cola creciendo.

Prometheus recomienda buenas prácticas de nombres y etiquetas para métricas, con prefijos de aplicación y unidades coherentes.¹⁴ Esto parece burocracia hasta que tienes que escribir una alerta a las tres de la mañana. Una métrica mal nombrada se convierte en confusión.

Señales útiles para escalar serving:

SEÑAL	QUÉ INDICA	DECISIÓN POSIBLE
<code>serving_queue_depth</code>	Hay más peticiones esperando.	Añadir réplica o aplicar backpressure.
<code>serving_queue_oldest_seconds</code>	La petición más vieja espera demasiado.	Priorizar, escalar o rechazar entrada nueva.
<code>gpu_memory_available_bytes</code>	Queda poco margen de KV cache.	Bajar batch, limitar contexto o añadir GPU.
<code>ttft_seconds_p95</code>	Primer token llega tarde.	Revisar cola, prefill, contexto y admisión.
<code>tpot_seconds_p95</code>	Cada token tarda demasiado.	Revisar decode, batch, kernels y modelo.
<code>tokens_per_second</code>	Producción bruta.	Mirar junto a goodput, no sola.
<code>goodput_runs_per_second</code>	Runs válidas dentro de SLO.	Escalar si cae durante tráfico normal.
<code>contract_failure_ratio</code>	Salidas rechazadas por contrato.	No se arregla con más GPU; mirar prompt, modelo o schema.

TGI expone métricas como tamaño de batch, duración de decode, duración de inferencia, tamaño de cola, duración total, tokens generados, longitud de entrada, tiempo medio por token y duración de cola.¹⁵ Aunque uses otro runtime, esa lista es pedagógica: te enseña qué mirar.

OpenTelemetry también trabaja convenciones semánticas para sistemas GenAI, de modo que proveedores, modelos, operaciones, tokens y atributos de inferencia puedan observarse con nombres compartidos.¹⁶ No todo estará cerrado para siempre, pero la dirección es clara: si no etiquetas bien la inferencia, luego no puedes compararla.

En el día a día

En un equipo real, este capítulo aparece cuando alguien dice “el modelo tarda” y nadie sabe qué significa “tarda”. Puede tardar en entrar al serving, tardar en prefill, tardar en generar, tardar en validar o tardar en llegar al cliente.

Una conversación seria separa síntomas:

SÍNTOMA	PREGUNTA DE INGENIERÍA
Primer token lento	¿Cola interna, prefill largo, modelo frío o scheduler saturado?
Tokens salen despacio	¿Decode lento, batch demasiado grande, kernel, cuantización o modelo grande?
Memoria al límite	¿KV cache, contexto, batch, pesos, buffers o fragmentación?
GPU alta pero usuarios esperando	¿Throughput bruto alto, pero goodput bajo?
Funciona con 1 usuario y falla con 50	¿No se probó distribución realista de concurrencia y tokens?
Escalado no mejora	¿El cuello no era réplica de serving, sino cola, base de datos, red o contrato?

El trabajo del ingeniero no es adivinar. Es diseñar el sistema para que la respuesta esté en las métricas.

Por qué debería importarte

Porque muchas decisiones caras se esconden en serving. Un modelo pequeño bien servido puede dar mejor experiencia que un modelo grande mal configurado. Un contexto gigantesco puede disparar KV cache y empeorar p95. Una cola sin backpressure puede aceptar trabajo que nunca cumplirá SLO. Un autoscaler basado solo en CPU puede reaccionar tarde o mirar la señal equivocada.

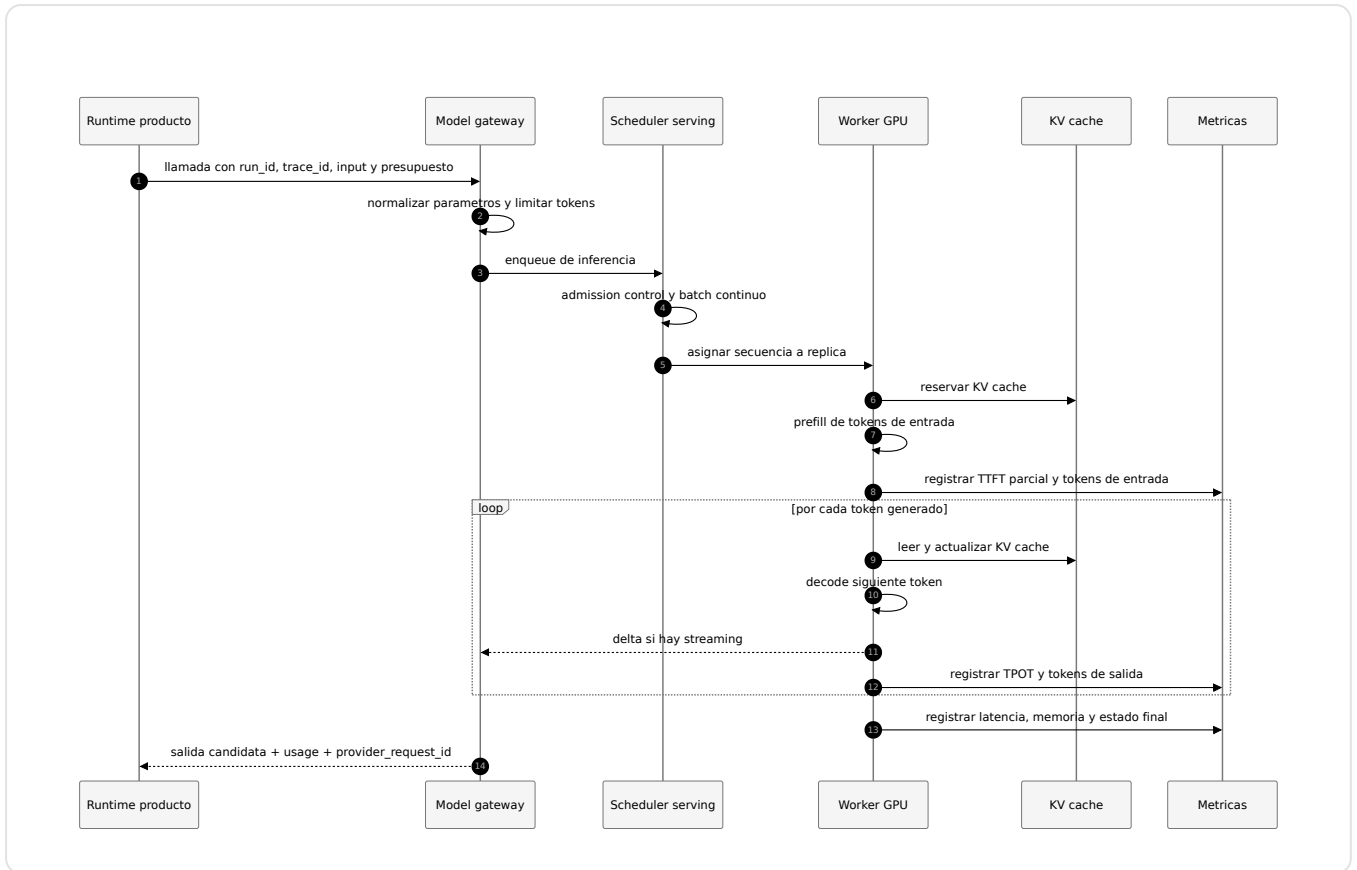
Y porque este capítulo conecta directamente con presupuesto. En IA, coste no es solo “precio por token”. También es GPU alquilada, memoria desaprovechada, réplicas calientes, reintentos, colas largas, salidas descartadas por contrato y tiempo de ingeniería depurando sin señales.

Dónde solía tropezar yo

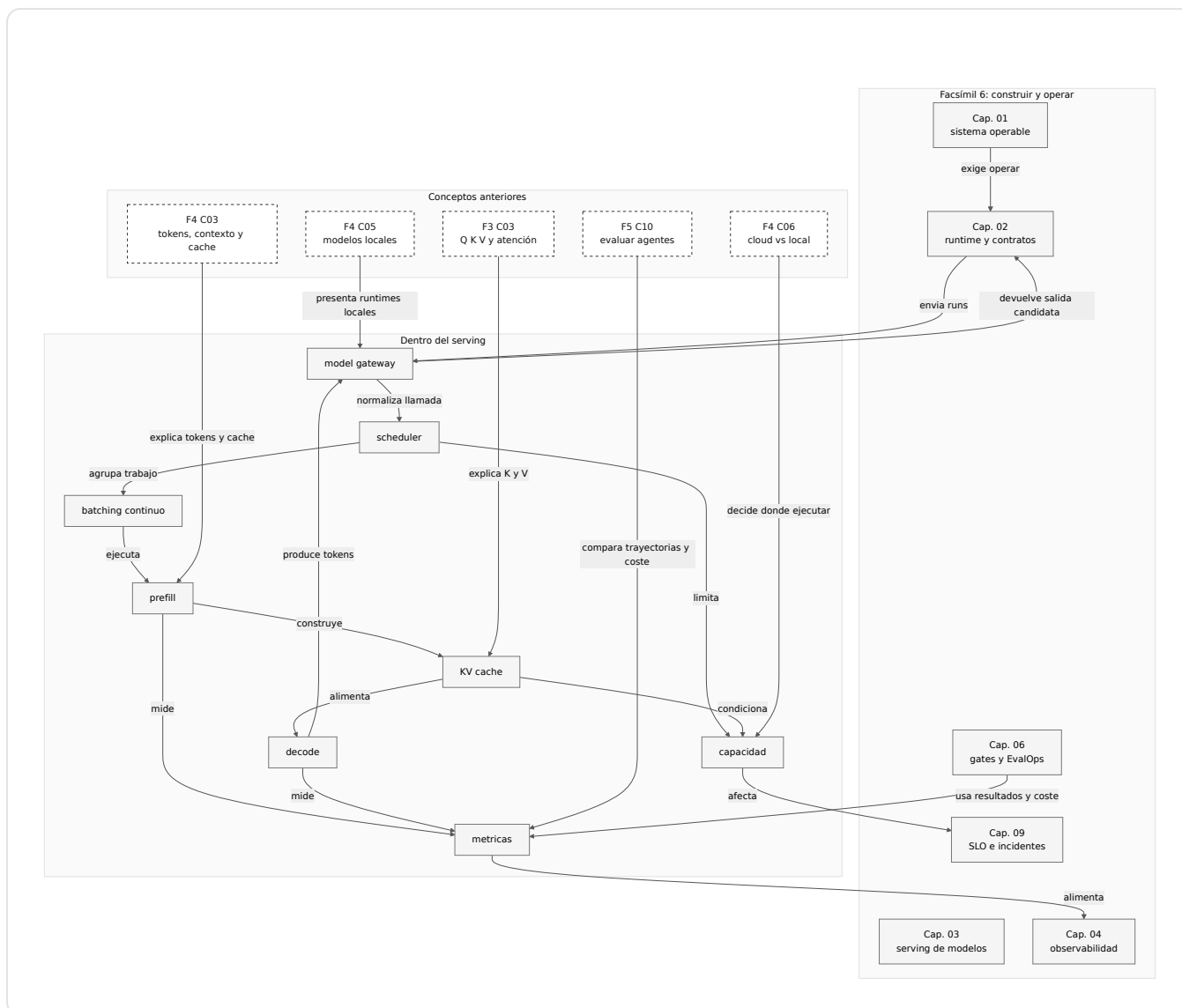
TROPIEZO	POR QUÉ ES UN PROBLEMA	ANTÍDOTO
Medir solo tokens por segundo	Puedes producir mucho y aun así llegar tarde.	Medir goodput, TTFT, TPOT y p95/p99.
Confundir contexto máximo con contexto recomendable	El máximo puede caber, pero destruir coste y latencia.	Diseñar límites por caso de uso y medir KV cache.
Escalar por CPU	El cuello puede estar en GPU, KV cache, cola o scheduler.	Escalar con métricas de serving y cola.
Olvidar el warmup	La primera petición puede parecer un fallo de latencia.	Separar liveness, readiness y model readiness.
Poner batch muy alto	Mejora throughput bruto, pero puede empeorar TTFT o memoria.	Probar batch con distribución realista y SLO.
No distinguir prefill de decode	No sabes si duele la entrada larga o la salida larga.	Medir ambas fases por separado.
Comparar runtimes sin protocolo	Cada prueba cambia prompt, tokens, batch o hardware.	Escribir un benchmark reproducible antes de decidir.
Tratar la GPU como infinita	KV cache y buffers aparecen antes que el entusiasmo.	Calcular memoria con margen y probar p95.

Cómo encaja todo

Primero veamos el flujo temporal de una run que llega al serving:



Ahora el mapa conceptual del capítulo:



La relación más importante es esta: el capítulo 2 decide qué entra y con qué contrato; el capítulo 3 decide si hay capacidad real para ejecutarlo; el capítulo 4 nos dará señales para no operar a oscuras.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Serving	Capa que mantiene modelos cargados y atiende inferencias con límites de memoria, latencia y capacidad.
Engine de inferencia	Motor que ejecuta el modelo, gestiona memoria, kernels, batch y generación.
Scheduler	Componente que decide qué peticiones entran en ejecución y cuándo.
Prefill	Fase que procesa tokens de entrada antes de generar.
Decode	Fase que genera tokens de salida paso a paso.
KV cache	Memoria que guarda claves y valores de atención para reutilizar contexto durante generación.
Batching continuo	Planificación que recompone el batch mientras unas secuencias terminan y otras entran.
Prefix caching	Reutilización de KV cache cuando varias peticiones comparten prefijo.
Chunked prefill	Técnica que divide entradas largas para que no bloqueen completamente otras generaciones.
TTFT	Tiempo hasta el primer token o primer evento útil.
TPOT	Tiempo por token generado durante decode.
Throughput	Trabajo completado por unidad de tiempo.
Goodput	Trabajo correcto y dentro de SLO por unidad de tiempo.
Tensor parallel	Reparto de operaciones o pesos de una capa entre varias GPUs.
Pipeline parallel	Reparto de capas entre dispositivos, con comunicación entre etapas.
Data parallel	Réplicas completas del modelo atienden tráfico distinto.
Model readiness	Señal de que el modelo está cargado y puede generar, no solo de que el proceso vive.
Warmup	Ejecución inicial para cargar pesos, inicializar kernels y reducir primera latencia.
Drenaje	Proceso de dejar de aceptar nuevas peticiones antes de apagar una réplica.

TÉRMINO**DEFINICIÓN**

Autoscaling

Ajuste automático de réplicas a partir de métricas observadas.

Antes de pasar página

- ☐ ¿Puedes explicar por qué servir un modelo no es lo mismo que tenerlo descargado?
- ☐ ¿Puedes dibujar el camino runtime de producto -> gateway -> scheduler -> worker -> KV cache -> salida?
- ☐ ¿Puedes explicar prefill y decode con tus palabras?
- ☐ ¿Puedes decir por qué la KV cache crece con batch y contexto?
- ☐ ¿Puedes estimar memoria de pesos con parámetros y bytes por peso?
- ☐ ¿Puedes estimar memoria de KV cache con capas, cabezas KV, dimensión y tokens activos?
- ☐ ¿Puedes distinguir throughput y goodput?
- ☐ ¿Puedes explicar TTFT y TPOT sin mezclarlos?
- ☐ ¿Puedes justificar por qué p95 y p99 importan más que una media cómoda?
- ☐ ¿Puedes decir cuándo el batching continuo ayuda y cuándo puede complicar memoria o latencia?
- ☐ ¿Puedes definir qué debería comprobar un health check de modelo?
- ☐ ¿Puedes proponer señales de autoscaling mejores que CPU para inferencia?
- ☐ ¿Puedes explicar por qué un contrato fallido no se arregla añadiendo GPU?
- ☐ ¿Puedes usar el calculador de capacidad y cambiar supuestos para ver el impacto?
- ☐ ¿Puedes decidir cuándo usar API cloud, runtime propio o arquitectura híbrida?

En resumen

IDEA	QUÉ DEBES LLEVARTE
Serving es una capa operativa, no un endpoint bonito.	Mantiene modelos cargados, agenda peticiones, gestiona memoria, genera tokens y emite señales.
Prefill y decode explican latencias distintas.	Entrada larga duele en prefill; salida larga duele en decode.
La KV cache convierte contexto y batch en memoria real.	Más tokens activos pueden limitar concurrencia antes que los pesos del modelo.
Batching mejora throughput, pero no es gratis.	Puede subir utilización y también presión de memoria o TTFT.
Capacidad se dimensiona con supuestos escritos.	Llegadas, p95, concurrencia útil, memoria, margen y goodput deben quedar explícitos.
Escalar exige mirar la señal correcta.	CPU sola no basta para inferencia; hacen falta cola, TTFT, TPOT, memoria, tokens y SLO.

Para saber más

- Dean, J. y Barroso, L. A. (2013). *The Tail at Scale*. Communications of the ACM, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>
- Hugging Face. (2026). *Text Generation Inference*. <https://huggingface.co/docs/text-generation-inference/en/index>
- Hugging Face. (2026). *Text Generation Inference: Metrics*. <https://huggingface.co/docs/text-generation-inference/reference/metrics>
- Kubernetes. (2026). *Horizontal Pod Autoscaling*. <https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/>
- Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H. y Stoica, I. (2023). *Efficient Memory Management for Large Language Model Serving with PagedAttention*. Proceedings of the ACM Symposium on Operating Systems Principles. <https://doi.org/10.48550/arXiv.2309.06180>
- Little, J. D. C. (1961). *A Proof for the Queuing Formula: $L = \lambda W$* . Operations Research, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>
- MLCommons. (2026). *MLPerf Inference: Datacenter Benchmark*. <https://mlcommons.org/benchmarks/inference-datacenter/>
- NVIDIA. (2026). *TensorRT-LLM Documentation*. <https://docs.nvidia.com/tensorrt-llm/>

- OpenTelemetry. (2026). *Semantic Conventions for Generative AI Systems*. <https://opentelemetry.io/docs/specs/semconv/gen-ai/>
 - Prometheus. (2026). *Metric and label naming*. <https://prometheus.io/docs/practices/naming/>
 - SGLang. (2026). *Welcome to SGLang*. <https://docs.sglang.io/index.html>
 - vLLM Project. (2026). *vLLM Documentation*. <https://docs.vllm.ai/en/stable/>
 - vLLM. (2026). *OpenAI-Compatible Server*. https://docs.vllm.ai/en/latest/serving/openai_compatible_server/
-

Notas

1. vLLM Project. (2026). *vLLM Documentation*. <https://docs.vllm.ai/en/stable/>. Consultado el 27 de mayo de 2026. vLLM. (2026). *OpenAI-Compatible Server*. https://docs.vllm.ai/en/latest/serving/openai_compatible_server/. Consultado el 27 de mayo de 2026. ↵
2. SGLang. (2026). *Welcome to SGLang*. <https://docs.sglang.io/index.html>. Consultado el 27 de mayo de 2026. ↵
3. NVIDIA. (2026). *TensorRT-LLM Documentation*. <https://docs.nvidia.com/tensorrt-llm/>. Consultado el 27 de mayo de 2026. ↵
4. Hugging Face. (2026). *Text Generation Inference*. <https://huggingface.co/docs/text-generation-inference/en/index>. Consultado el 27 de mayo de 2026. Hugging Face. (2026). *Text Generation Inference: Metrics*. <https://huggingface.co/docs/text-generation-inference/reference/metrics>. Consultado el 27 de mayo de 2026. ↵
5. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H. y Stoica, I. (2023). *Efficient Memory Management for Large Language Model Serving with PagedAttention*. Proceedings of the ACM Symposium on Operating Systems Principles. <https://doi.org/10.48550/arXiv.2309.06180>. ↵
6. Hugging Face. (2026). *Text Generation Inference*. <https://huggingface.co/docs/text-generation-inference/en/index>. Consultado el 27 de mayo de 2026. ↵
7. Dean, J. y Barroso, L. A. (2013). *The Tail at Scale*. Communications of the ACM, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>. ↵
8. MLCommons. (2026). *MLPerf Inference: Datacenter Benchmark*. <https://mlcommons.org/benchmarks/inference-datacenter/>. Consultado el 27 de mayo de 2026. ↵

9. Erlang, A. K. (1917). Solution of some Problems in the Theory of Probabilities of Significance in Automatic Telephone Exchanges. *The Post Office Electrical Engineers' Journal*, 10, 189-197. Origen de las fórmulas de Erlang B y C para sistemas con varios servidores. ↵
10. Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 30, 483-485. <https://doi.org/10.1145/1465482.1465560>. Establece el límite de aceleración por paralelización. ↵
11. Gunther, N. J. (2007). *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*. Springer. Formula la ley universal de escalabilidad con términos de contención y de coherencia. ↵
12. Dean, J., & Barroso, L. A. (2013). The Tail at Scale. *Communications of the ACM*, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>. Explica cómo la latencia de cola domina cuando una petición depende de muchos componentes. ↵
13. Kubernetes. (2026). *Horizontal Pod Autoscaling*. <https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/>. Consultado el 27 de mayo de 2026. ↵
14. Prometheus. (2026). *Metric and label naming*. <https://prometheus.io/docs/practices/naming/>. Consultado el 27 de mayo de 2026. ↵
15. Hugging Face. (2026). *Text Generation Inference: Metrics*. <https://huggingface.co/docs/text-generation-inference/reference/metrics>. Consultado el 27 de mayo de 2026. ↵
16. OpenTelemetry. (2026). *Semantic Conventions for Generative AI Systems*. <https://opentelemetry.io/docs/specs/semconv/gen-ai/>. Consultado el 27 de mayo de 2026. ↵

Capítulo 04: Observabilidad: logs, métricas, trazas y costes

Qué deberías poder hacer al terminar

En el capítulo 02 aprendimos a convertir una petición en una run. En el capítulo 03 vimos que el modelo se sirve dentro de un sistema con scheduler, KV cache, workers y límites de capacidad. Ahora toca una pregunta más incómoda: cuando algo va lento, caro, incorrecto o raro, **¿cómo lo sabemos sin adivinar?**

Observabilidad no es “tener muchos logs”. Tampoco es abrir un dashboard enorme y esperar que una gráfica confiese. Observabilidad es diseñar el sistema para que cada run deje señales suficientes, estructuradas y útiles.

Al terminar, deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Separar logs, métricas y trazas.	Sabes cuándo usar un evento, una serie agregada o una historia completa de ejecución.
Diseñar una traza de una run de IA.	Incluyes <code>trace_id</code> , spans, atributos, eventos, versiones, tokens, costes y estado final.
Definir SLIs y SLOs específicos de IA.	Mides latencia, coste, salida válida, contrato, recuperación, tools y serving.
Evitar métricas imposibles de mantener.	Controlas cardinalidad, etiquetas, nombres y retención.
Diseñar alertas accionables.	Alertas por síntomas de usuario, burn rate y señales que alguien pueda corregir.
Calcular coste por run aceptada.	No te quedas en precio por token; conectas coste con calidad y contrato.

La idea central: **si una run no deja una historia reconstruible, no existe operativamente.**

El problema que venimos a resolver

Imagina que un usuario dice: “ayer el asistente tardó mucho y al final me dio una respuesta que no servía”. Sin observabilidad, el equipo abre logs sueltos, busca por hora aproximada, mira si hubo error del proveedor, revisa alguna métrica de CPU y acaba con una hipótesis floja.

Con observabilidad bien diseñada, la pregunta cambia. Buscamos el `run_id`, abrimos la traza y vemos: cola de producto 800 ms, cola de serving 1200 ms, prefill largo por contexto de 38.000 tokens, dos reintentos de retrieval, salida rechazada una vez por contrato, coste final 0,18 EUR y estado `succeeded_with_review`.

La diferencia no es estética. La primera escena obliga a investigar a oscuras. La segunda permite razonar.

Qué no es observabilidad

Observabilidad no es acumular todo “por si acaso”. Guardar prompts completos, documentos, salidas y dumps internos sin criterio crea coste, riesgo de privacidad, ruido y deuda.

Tampoco es una lista interminable de métricas. Si nadie sabe qué significa una serie, quién la mantiene, qué umbral importa o qué acción provoca, esa métrica es decoración operativa.

Y no es alertar por cada anomalía. El libro de SRE de Google insiste en que monitorizar debe ayudar a entender qué está roto y por qué, pero las alertas que interrumpen a personas deben ser simples, accionables y con poco ruido.¹

CONFUSIÓN	POR QUÉ FALLA	MEJOR ENFOQUE
Guardar todos los prompts completos	Aumenta coste, sensibilidad y ruido.	Guardar IDs, hashes, tamaños, versiones y muestras controladas.
Medir solo latencia media	Oculto p95, p99 y usuarios que esperan demasiado.	Usar histogramas, percentiles y SLO.
Alertar por causa interna	Puede sonar grave sin afectar al usuario.	Alertar por síntoma medible y accionable.
Mirar solo logs	Pierdes agregación y causalidad entre servicios.	Combinar logs, métricas y trazas.
Mirar solo coste por token	Ignora reintentos, fallos de contrato y revisión.	Medir coste por run aceptada.

Qué sí es observabilidad en IA

Observabilidad es la capacidad de contestar preguntas de ingeniería con señales del sistema. En IA generativa, esas preguntas tienen una forma propia:

PREGUNTA	SEÑAL NECESARIA
¿Qué versión respondió?	<code>model_id</code> , <code>prompt_version</code> , <code>manifest_version</code> , <code>contract_version</code> .
¿Qué vio el modelo?	IDs de documentos, chunks, tamaños, hashes y política de retención.
¿Cuánto costó?	Tokens de entrada, tokens de salida, precio aplicado, reintentos y proveedor.
¿Dónde se fue la latencia?	Spans de cola, retrieval, tools, prefill, decode, validación y salida.
¿La respuesta era válida?	Estado de contrato, evaluadores, revisión y motivo de rechazo.
¿Se saturó el serving?	Cola interna, TTFT, TPOT, memoria, batch y goodput.
¿Qué debería hacer soporte?	<code>run_id</code> , estado final, error tipado, resumen de decisión y siguiente acción.

Fecha de corte: 27 de mayo de 2026. Fuentes consultadas ese día: Google SRE Book, Google SRE Workbook, OpenTelemetry traces, logs, metrics y convenciones semánticas GenAI, Prometheus metric naming, W3C Trace Context y documentación oficial de Grafana, Langfuse, LangSmith, Phoenix, Helicone, Braintrust, OpenInference, OpenLLMetry y Datadog LLM Observability. Lo estable es la arquitectura: logs, métricas, trazas, SLIs, SLOs, cardinalidad, sampling, alertas y coste. Lo cambiante son nombres exactos de atributos, SDKs, backends de observabilidad, productos comerciales y convenciones GenAI en evolución.

OpenTelemetry describe trazas como una forma de entender el camino de una petición a través de una aplicación, compuestas por spans relacionados.² También define APIs para crear spans, asociarlos al contexto activo y registrar atributos, eventos y estado.³ Sus convenciones semánticas para GenAI incorporan atributos sobre sistemas generativos, operaciones, modelos y uso de tokens.⁴

Las tres señales principales

Los tres nombres clásicos son logs, métricas y trazas. No compiten: responden a preguntas distintas.

SEÑAL	PREGUNTA QUE RESPONDE	EJEMPLO EN UNA RUN DE IA
Log	¿Qué evento ocurrió?	<code>contract.failed</code> con campo ausente y versión de schema.
Métrica	¿Cuánto ocurre en el tiempo?	<code>ai_run_latency_seconds_bucket</code> por ruta y estado.
Traza	¿Qué camino siguió esta run concreta?	<code>api.receive -> queue.wait -> retrieval.search -> model.call -> output.validate</code> .

OpenTelemetry trata logs como registros con cuerpo, severidad, timestamp, atributos y contexto de traza cuando existe.⁵ Las métricas, en cambio, representan medidas agregables en el tiempo: contadores, histogramas, gauges y otros instrumentos.⁶

Una forma de pensarlo:

SI PREGUNTAS...	USA PRINCIPALMENTE...	PORQUE...
“¿Qué ocurrió en esta run?”	Traza	Necesitas causalidad y orden.
“¿Cuántas runs fallan por contrato?”	Métrica	Necesitas agregación temporal.
“¿Qué campo faltaba en esta salida?”	Log o evento de span	Necesitas detalle puntual.
“¿Se degradó el p95 tras desplegar?”	Métrica + traza de muestra	Necesitas tendencia y ejemplos.
“¿Qué documento se recuperó?”	Traza con atributos controlados	Necesitas contexto sin guardar contenido completo.

La traza de una run de IA

Una traza debería parecerse a una historia técnica. No una novela, no un volcado masivo. Una historia con capítulos claros.

```
trace_id = trace_8f21...
run_id   = run_20260527_0042

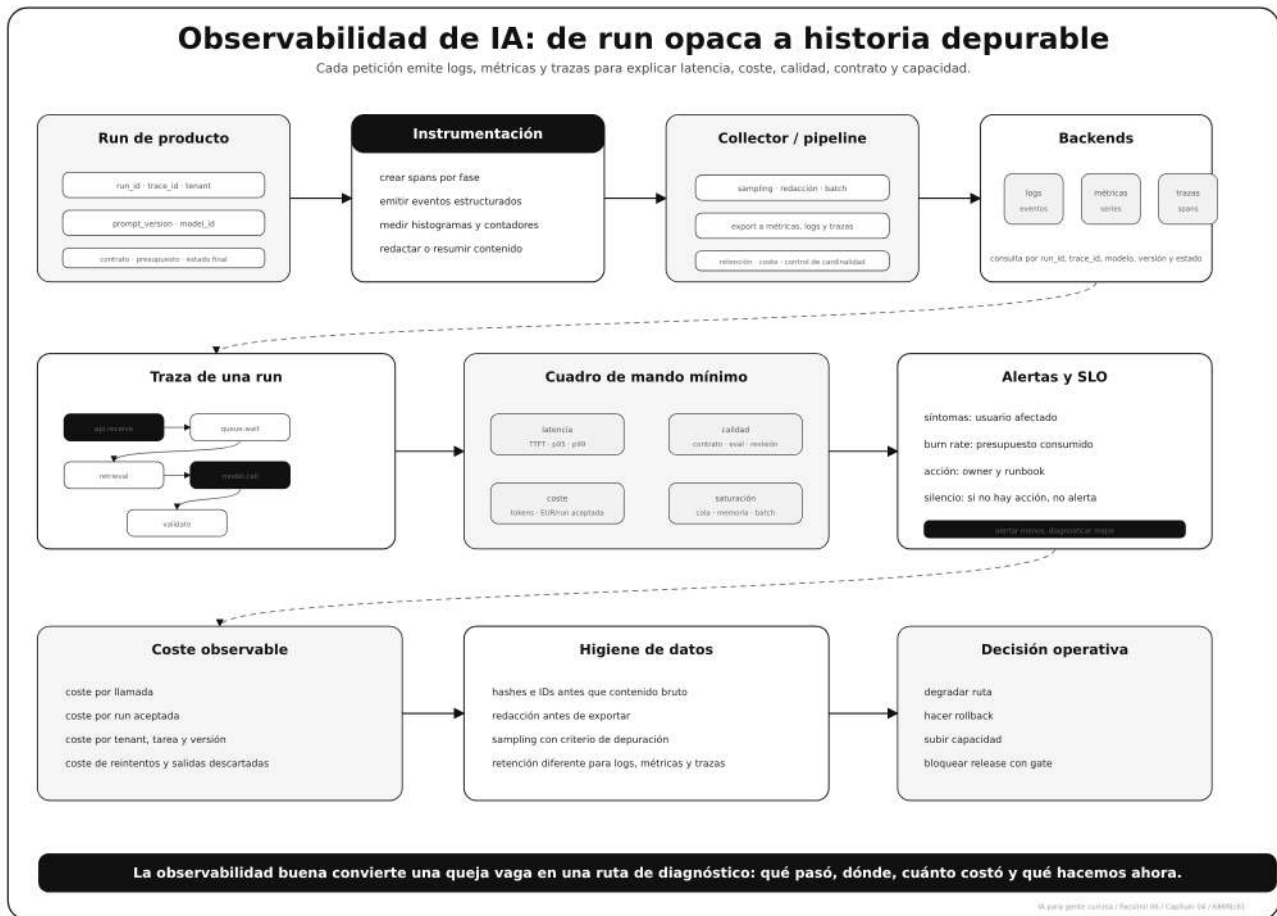
api.receive
  input.validate
  run.create
queue.wait
retrieval.search
  retrieval.rerank
tool.call
model.call
  model.prefill
  model.decode
output.validate
run.close
```

Cada span debe tener atributos. La regla: guarda lo necesario para depurar, evaluar y auditar sin conservar más contenido del necesario.

SPAN	ATRIBUTOS ÚTILES	QUÉ NO CONVIENE METER SIN POLÍTICA
api.receive	tenant_id , route , request_bytes , idempotency_replay .	Payload completo.
input.validate	contract_version , valid , error_count .	Datos sensibles de entrada.
retrieval.search	index_version , top_k , query_tokens , source_ids .	Texto completo de documentos.
tool.call	tool_name , timeout_ms , result_state , retry_count .	Secretos, credenciales o respuestas extensas.
model.call	provider , model_id , input_tokens , output_tokens , ttft_ms , tpot_ms .	Prompt completo por defecto.
output.validate	contract_version , valid , missing_fields , extra_fields .	Salida completa si contiene datos sensibles.
run.close	final_state , latency_ms , cost_eur , accepted , review_required .	Comentarios internos sin estructura.

El contexto de traza permite que varios servicios compartan la misma historia. W3C Trace Context estandariza cabeceras como `traceparent` para propagar identificadores de traza entre servicios.⁷ En nuestro runtime, eso significa que la API, la cola, el worker y el gateway de modelo no deberían inventar historias separadas.

Anatomía visual de observabilidad para IA



El SVG tiene muchas cajas porque la observabilidad también es arquitectura. No basta con “emitir algo”. Hay que decidir qué se instrumenta, cómo se transporta, dónde se guarda, cuánto tiempo vive, qué se muestra y qué acción provoca.

SLIs y SLOs para sistemas de IA

Ya vimos que un SLI es un indicador medible y un SLO es un objetivo interno sobre ese indicador. Aquí lo aplicamos a IA.

Las cuatro señales doradas de SRE son latencia, tráfico, errores y saturación.⁸ En IA conviene mantenerlas y añadir dos dimensiones: calidad operativa y coste.

SEÑAL	SLI POSIBLE	SLO POSIBLE
Latencia	Porcentaje de runs con p95 menor o igual a 6 s.	95% de runs interactivas terminan en 6 s o menos.
Tráfico	Runs por minuto por tarea, tenant y ruta.	El sistema soporta pico esperado sin rechazos indebidos.
Errores	Runs con estado final no aceptado.	Menos del 2% terminan en <code>timed_out</code> , <code>contract_failed</code> o <code>provider_unavailable</code> .
Saturación	Edad máxima en cola y memoria disponible.	Cola p95 menor o igual a 1 s y margen de memoria mayor al 15%.
Calidad	Salidas aceptadas por contrato y eval.	98% de respuestas publicadas pasan contrato y evaluación automática mínima.
Coste	Coste por run aceptada.	p95 de coste por run aceptada menor o igual a 0,08 EUR.

La fórmula del SLI de éxito operativo puede ser:

$$SLI_{aceptadas} = \frac{N_{runs\ aceptadas}}{N_{runs\ totales}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$N_{runs\ aceptadas}$	Runs que terminaron con salida válida, dentro de contrato y sin revisión bloqueante.	9.700
$N_{runs\ totales}$	Runs evaluables en la ventana.	10.000
$SLI_{aceptadas}$	Proporción de runs aceptadas.	0,97

$$SLI_{aceptadas} = \frac{9700}{10000} = 0,97$$

Si el SLO era 98%, estamos por debajo. No hace falta dramatizar: hace falta mirar la traza agregada y saber qué grupo pesa más.

Para coste:

$$C_{aceptada} = \frac{C_{tokens} + C_{serving} + C_{tools} + C_{revisión}}{N_{runs\ aceptadas}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
C_{tokens}	Coste de tokens de modelos o proveedor.	120 EUR
C_{serving}	Coste de GPUs o runtime propio.	80 EUR
C_{tools}	Coste de herramientas externas.	20 EUR
$C_{\text{revisión}}$	Coste estimado de revisión humana o soporte.	30 EUR
$N_{\text{runs aceptadas}}$	Runs aceptadas en la ventana.	5.000

$$C_{\text{aceptada}} = \frac{120 + 80 + 20 + 30}{5000} = 0,05 \text{ EUR}$$

Esto cambia el debate. Un modelo más caro por token puede ser más barato por run aceptada si reduce reintentos, revisiones y salidas descartadas.

Burn rate: gastar el presupuesto de error

El SRE Workbook explica cómo convertir SLOs en alertas mirando el consumo del presupuesto de error, no solo un umbral aislado.⁹

Si tu SLO es 99%, tu presupuesto de error es 1%. Si en una ventana concreta fallan 5% de runs, consumes presupuesto cinco veces más rápido que lo permitido.

$$\text{burn rate} = \frac{\text{error rate observado}}{\text{error rate permitido}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
Error rate observado	Proporción real de runs fuera de SLO en una ventana.	5%
Error rate permitido	Proporción permitida por el SLO.	1%
Burn rate	Veces que consumes el presupuesto.	5x

$$\text{burn rate} = \frac{0,05}{0,01} = 5$$

En IA, el “error” no tiene que ser solo HTTP 500. Puede ser:

ERROR OPERATIVO	CÓMO SE DETECTA
Timeout	Estado final <code>timed_out</code> .
Contrato fallido	Estado <code>contract_failed</code> .
Coste fuera de presupuesto	<code>cost_eur > budget.max_cost_eur</code> .
Latencia fuera de SLO	<code>latency_ms > slo.max_latency_ms</code> .
Salida no aceptada	Eval mínima o revisión bloqueante.
Ruta degradada sin aviso	Fallback no comunicado o no trazado.

Aquí aparece una diferencia importante: no todo error técnico afecta igual al usuario, y no toda respuesta HTTP exitosa es éxito de producto. Una run puede devolver `200 OK` y estar fuera de SLO porque costó demasiado, tardó demasiado o falló contrato semántico.

Métricas sin volverse loco con la cardinalidad

Prometheus recomienda nombres claros, unidades consistentes y etiquetas que mantengan significado estable.¹⁰ En IA, la tentación de etiquetar todo es enorme: `run_id` , `user_id` , `prompt` , `document_id` , `tool_args` , `model_id` , `tenant_id` , `cost_eur` ...

No lo hagas así. Una métrica con demasiadas combinaciones de etiquetas se vuelve cara, lenta e inmanejable.

ETIQUETA	¿BUENA PARA MÉTRICA?	MEJOR USO
route	Sí	Métrica.
model_family	Sí	Métrica.
final_state	Sí	Métrica.
tenant_tier	A veces	Métrica si hay pocos valores.
run_id	No	Traza o log.
user_id	No	Traza con política de acceso, o hash.
prompt_text	No	No en métrica; contenido controlado aparte.
document_id	No si hay muchos	Traza, log o agregación por corpus.
cost_eur	No como etiqueta	Valor numérico o atributo de span.

Nombres razonables:

```
ai_run_total
ai_run_latency_seconds_bucket
ai_run_cost_eur_bucket
ai_model_input_tokens_total
ai_model_output_tokens_total
ai_contract_failure_total
ai_serving_queue_seconds_bucket
ai_tool_call_total
ai_retrieval_documents_total
```

Cada métrica debería poder contestar algo. Si no sabes qué decisión habilita, todavía no es una buena métrica.

Logs: detalle sin ruido

Un log útil en IA debería ser estructurado. No:

```
algo falló llamando al modelo
```

Mejor:

```
{
  "event": "model.call.completed",
  "timestamp": "2026-05-27T18:30:00Z",
  "run_id": "run_42",
  "trace_id": "trace_abc",
  "span_id": "span_model",
  "provider": "local-vllm",
  "model_id": "qwen3-8b-instruct",
  "input_tokens": 4200,
  "output_tokens": 380,
  "latency_ms": 3100,
  "cost_eur": 0.021,
  "status": "ok"
}
```

El log debe ayudar a reconstruir un evento. No debe convertirse en un contenedor donde metemos todo lo que no supimos modelar.

Una regla práctica:

GUARDA EN LOG	GUARDA EN TRAZA	GUARDA EN MÉTRICA
Evento concreto y explicación breve.	Causalidad completa de la run.	Agregado temporal.
Error tipado y atributos.	Duración por fase.	Contadores, histogramas y gauges.
Decisión tomada.	Relación padre-hijo entre pasos.	SLO, p95, p99, burn rate.

Costes observables

El coste en IA tiene varias capas:

COSTE	EJEMPLO	SEÑAL
Tokens	Entrada y salida de modelo.	<code>input_tokens</code> , <code>output_tokens</code> , precio aplicado.
Serving	GPU propia, memoria y réplicas.	Coste por hora, utilización y goodput.
Herramientas	APIs externas, búsquedas, OCR, bases de datos.	<code>tool_name</code> , <code>tool_cost_eur</code> .
Reintentos	Llamadas repetidas por timeout o contrato.	<code>retry_count</code> , coste acumulado.
Revisión	Tiempo humano o soporte.	<code>review_required</code> , coste estimado.
Almacenamiento	Logs, trazas, métricas, datasets.	Retención y volumen exportado.

El coste por token es solo una pieza. El coste por run aceptada te dice si la arquitectura sirve.

Comparación sencilla:

SISTEMA	COSTE TOTAL	RUNS TOTALES	RUNS ACEPTADAS	COSTE POR ACEPTADA
A	100 EUR	10.000	5.000	0,020 EUR
B	160 EUR	10.000	9.500	0,017 EUR

El sistema B parece más caro hasta que miras aceptadas. En producto, pagar menos por respuestas que luego se descartan no es ahorro; es trabajo inútil bien facturado.

Sampling y retención

No puedes guardar todo con el mismo detalle para siempre. Debes decidir qué se conserva, cuánto tiempo y con qué nivel.

SEÑAL	RETENCIÓN TÍPICA	REGLA RAZONABLE
Métricas agregadas	Semanas o meses.	Largas para tendencias y capacidad.
Trazas completas	Menos tiempo.	Más detalle para fallos, canary y muestras.
Logs de eventos	Variable.	Retener eventos operativos, no contenido sensible sin motivo.
Prompts/salidas completas	Muy controlado.	Solo con consentimiento, redacción, muestreo o entorno de evaluación.
Datasets de eval	Versionado largo.	Sirven para comparar cambios.

Sampling no significa “tirar cosas al azar sin pensar”. Puedes muestrear:

ESTRATEGIA	CUÁNDO USARLA
100% de errores	Para investigar fallos de contrato, timeouts y rutas nuevas.
100% de canary	Mientras una versión nueva está en prueba.
1-5% de éxitos	Para tener muestra sana sin coste excesivo.
Tail sampling	Guardar trazas lentas o caras tras ver el resultado.
Muestreo por tenant/tarea	Asegurar cobertura de segmentos importantes.

La cola de latencias altas importa mucho. Dean y Barroso explican que los percentiles altos pueden dominar la experiencia de usuario cuando una operación depende de varias piezas.¹¹ Por eso no basta con muestrear “runs normales”. También necesitamos mirar las lentas, caras y rechazadas.

De métricas a decisiones

Un dashboard bueno no enseña todo. Enseña lo que decide.

PANEL	PREGUNTA	DECISIÓN
Salud de run	¿Cuántas terminan bien, mal o con revisión?	Parar release, degradar ruta o investigar contrato.
Latencia	¿Dónde se va el tiempo?	Ajustar cola, serving, contexto o tools.
Coste	¿Qué tarea/modelo/tenant dispara gasto?	Cambiar routing, budgets o límites.
Calidad	¿Baja la aceptación por versión?	Rollback o gate de despliegue.
Serving	¿Está saturado el modelo?	Escalar, limitar batch/contexto o cambiar runtime.
Retrieval	¿Se recupera demasiado o mal?	Ajustar índice, top_k , reranker o corpus.

La ley de Little vuelve a aparecer:

$$L = \lambda W$$

Si ves que sube W , el tiempo de espera, y la llegada λ no baja, entonces L , trabajo acumulado, crece. En observabilidad eso se traduce en una alerta por cola vieja, no solo por CPU alta.¹²

Resumir la experiencia en un número honesto: Apdex. La media de latencia miente y el p95 a solas no dice si el usuario está contento. Apdex es un índice estándar y publicado que combina ambas ideas: cuenta cuántas respuestas fueron satisfactorias, cuántas tolerables y cuántas frustrantes respecto a un umbral fijado.¹³

$$\text{Apdex}_t = \frac{N_{\text{satisfechas}} + \frac{N_{\text{tolerables}}}{2}}{N_{\text{total}}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
t	Umbral de satisfacción elegido.	2 segundos.
$N_{\text{satisfechas}}$	Respuestas por debajo de t .	8.000
$N_{\text{tolerables}}$	Respuestas entre t y $4t$, que cuentan la mitad.	1.500
N_{total}	Respuestas en la ventana.	10.000

En palabras: Apdex convierte la latencia en un número entre 0 y 1 que sí refleja la experiencia: con el ejemplo da 0,875. Lo potente es que fija el umbral por adelantado, así que deja de discutirse «¿2,3 segundos es mucho?» y se discute «¿qué Apdex aceptamos?».

Cuánto te puedes fiar de una métrica muestreada: el error de muestreo. Casi nadie guarda el 100% de las trazas; se muestrea. Pero una proporción calculada sobre una muestra tiene un error que depende del tamaño de la muestra, y el muestreo lo amplía. La teoría de muestreo da ese error estándar.¹⁴

$$EE(\hat{p}) = \frac{\hat{p}(1 - \hat{p})}{n}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\hat{p}$	Proporción estimada (por ejemplo, tasa de error).	0,02
n	Número de trazas en la muestra.	1.000
$EE(\hat{p})$	Error estándar de la estimación.	$\approx 0,0044$.

En palabras: con 1.000 trazas, una tasa de error «del 2%» tiene un error estándar de 0,44 puntos: el valor real ronda el 1,1%-2,9%. Si muestreas el 1% del tráfico, tu n efectivo es cien veces menor de lo que crees, y métricas de eventos raros se vuelven puro ruido. Por eso los errores y las anomalías se suelen muestrear al 100% aunque el tráfico normal no.

Por qué una alerta puede ser cierta el 99% y aun así inútil: el teorema de Bayes. Una alerta con 99% de acierto suena fiable hasta que la disparas sobre un evento raro. El teorema de Bayes explica la paradoja: si lo que buscas es poco frecuente, casi todas las alertas serán falsas por muy bueno que sea el detector.¹⁵

$$VPP = \frac{TPR \cdot \pi}{TPR \cdot \pi + FPR \cdot (1 - \pi)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
VPP	Valor predictivo positivo: si salta, ¿cuántas veces acierta?	0,16
TPR	Sensibilidad: incidencias detectadas.	0,99
FPR	Tasa de falsos positivos.	0,01
π	Prevalencia: cómo de raro es el problema.	0,002

En palabras: con un detector «al 99%» pero un problema que ocurre 2 de cada 1.000 veces, solo 1 de cada 6 alertas es real. La consecuencia operativa es directa: para que una alerta merezca despertar a alguien, no basta con un buen detector, hay que subir la prevalencia

(alertar sobre síntomas frecuentes y agregados, no sobre rarezas) o el equipo aprenderá a ignorarla.

Por qué no se puede promediar un p95: estimación de percentiles. Un error común es guardar el p95 de cada minuto y promediarlos para sacar el p95 de la hora. Está mal: el percentil de la unión no es la media de los percentiles. Calcularlo bien sobre flujos requiere estructuras como el t-digest, que estiman cuantiles con precisión acotada sin guardar todos los datos.¹⁶

$$Q(p) = \inf\{x : F(x) \geq p\}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$Q(p)$	Cuantil de orden p (el p95 es $Q(0,95)$).	2,4 segundos.
$F(x)$	Función de distribución acumulada de la latencia.	Empírica, a partir de las muestras.
p	Proporción objetivo.	0,95

En palabras: el p95 es el valor que deja por debajo el 95% de las muestras, y eso no es promediable. Para agregar percentiles por servicio o por ventana hay que combinar las distribuciones, no los números: por eso los buenos paneles guardan histogramas o digests, no percentiles ya calculados. Si tu panel promedia p95, está mintiendo justo donde más duele.

Correlacionar por versión

En sistemas de IA, muchas averías no aparecen como “se cayó el servidor”. Aparecen como “desde esta mañana responde peor”, “ahora cuesta más”, “este tipo de casos ya no pasa contrato” o “el RAG recupera fuentes menos útiles”. Para investigar eso, cada señal debe poder conectarse con la versión que la produjo.

No basta con tener `run_id`. Necesitamos que métricas, trazas y logs incluyan las versiones que cambian el comportamiento.

VERSIÓN	DÓNDE DEBERÍA APARECER	QUÉ PREGUNTA PERMITE RESPONDER
<code>model_id</code>	Span <code>model.call</code> , métrica de coste y dashboard.	¿El cambio viene del modelo?
<code>prompt_version</code>	Run, traza, logs de validación y evals.	¿El prompt nuevo rompió formato o criterio?
<code>contract_version</code>	Entrada, salida, error y validador.	¿Cambió el schema o falló la salida?
<code>retrieval_index_version</code>	Span <code>retrieval.search</code> .	¿El índice nuevo recupera peor?
<code>embedding_model_id</code>	Retrieval y métricas de cobertura.	¿Cambió el espacio vectorial?
<code>tool_version</code>	Span <code>tool.call</code> .	¿La tool cambió comportamiento o respuesta?
<code>runtime_version</code>	Serving y worker.	¿El runtime nuevo alteró latencia o streaming?
<code>release_id</code>	Toda la run.	¿Qué despliegue estaba activo?

La regla de ingeniería es simple: **si una pieza puede cambiar el resultado, debe estar versionada en la traza**. Si no, el postmortem se convierte en arqueología.

Dashboard mínimo que sí usaría

Un dashboard mínimo no intenta enseñar todas las series. Intenta guiar una investigación en menos de dos minutos. Primero enseña síntomas; después permite bajar a causas.

Yo lo diseñaría con seis paneles:

PANEL	MÉTRICAS PRINCIPALES	FILTRO OBLIGATORIO	PREGUNTA QUE RESPONDE
Salud de runs	ai_run_total , ratio de aceptadas, estados finales.	task , route , release_id .	¿El sistema está entregando resultados aceptables?
Latencia por fase	p50/p95/p99 de API, cola, retrieval, model, validación.	task , model_id , queue .	¿Dónde se va el tiempo?
Coste	coste por run, coste por aceptada, tokens, reintentos.	tenant_tier , task , model_id .	¿Qué está encareciendo el sistema?
Contratos y evals	contract_failed , score mínimo, revisión requerida.	contract_version , prompt_version .	¿La salida sirve o solo “responde”?
Serving	TTFT, TPOT, cola de serving, memoria, batch, goodput.	model_id , runtime_version .	¿El cuello está en inferencia?
Retrieval y tools	top_k, chunks usados, tool duration, tool error, permisos.	index_version , tool_name .	¿El contexto o una herramienta explica el problema?

El dashboard no reemplaza la traza. Te dice dónde mirar. La traza te cuenta la historia de una run concreta.

Alertas concretas, no ruido

Una alerta buena tiene cinco partes: síntoma, umbral, ventana, owner y acción. Si falta la acción, probablemente no debería interrumpir a nadie.

Algunas reglas razonables para empezar:

ALERTA	CONDICIÓN	PRIMERA ACCIÓN
Burn rate alto	<code>burn_rate_15m > 4</code> y al menos 30 runs evaluables.	Abrir dashboard de salud, filtrar por <code>release_id</code> y <code>task</code> .
Contratos fallando	<code>contract_failed_ratio_15m > 0.03</code> .	Comparar <code>prompt_version</code> , <code>contract_version</code> y ejemplos de trazas.
Coste fuera de presupuesto	<code>cost_per_accepted_p95_30m > budget</code> .	Ver tokens, reintentos, routing y tareas dominantes.
Cola vieja	<code>queue_oldest_seconds > queue_slo_seconds</code> .	Mirar llegadas, workers sanos, backpressure y serving.
TTFT alto	<code>ttft_p95_15m > ttft_slo</code> .	Revisar cola de serving, prefill, contexto y warmup.
TPOT alto	<code>tpot_p95_15m > tpot_slo</code> .	Revisar decode, batch, memoria y runtime.
Trazas incompletas	<code>trace_missing_ratio_1h > 0.01</code> .	Revisar propagación de <code>trace_id</code> y collector.
Reintentos altos	<code>retry_per_run_p95 > 1</code> .	Localizar dependencia lenta y política de backoff.

Ejemplo expresado como pseudorregla:

```

alert: AIHighBurnRate
if: burn_rate_15m > 4 and ai_run_total_15m >= 30
for: 10m
owner: ai-runtime
action: revisar /dashboards/ai-runs filtrando release_id, task y final_state

```

Lo importante no es copiar esta sintaxis. Lo importante es que la alerta ya trae una hipótesis de trabajo.

Runbook operativo

Una alerta sin runbook deja al equipo improvisando. Un runbook no debe ser una enciclopedia; debe decir qué mirar primero, qué acción tomar y cuándo escalar la investigación.

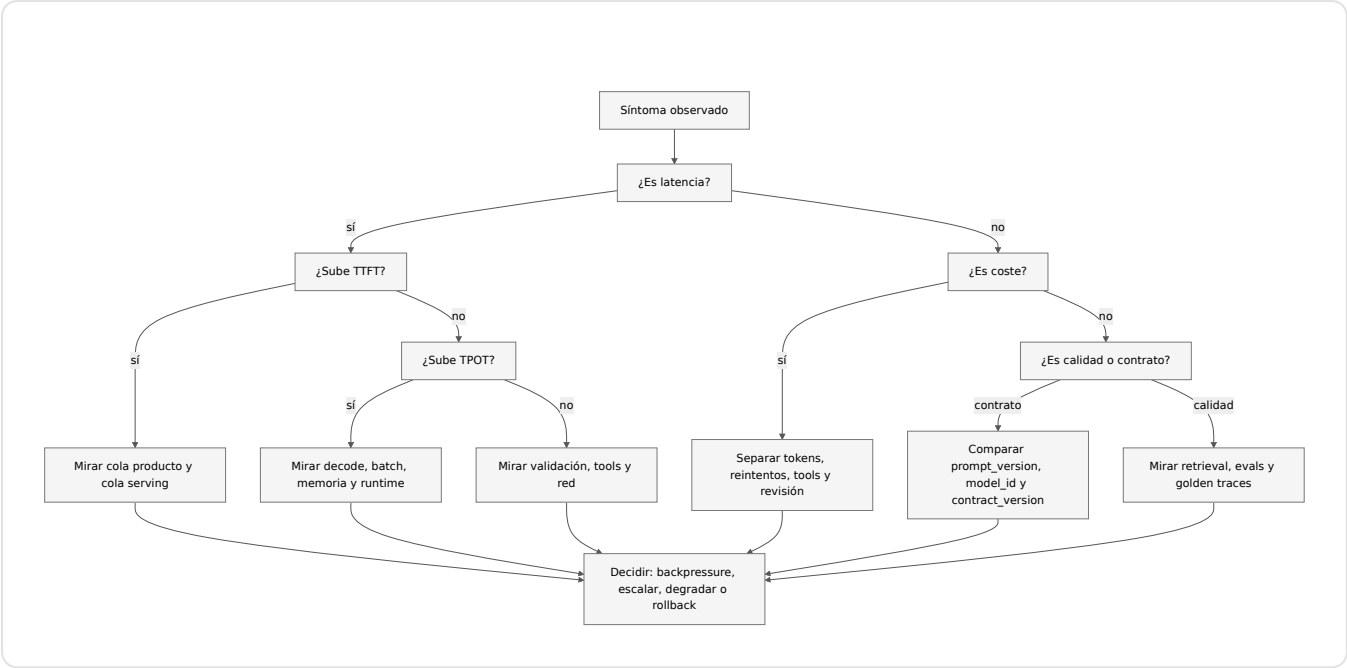
Para este capítulo, un runbook mínimo podría ser:

SÍNTOMA	SEÑAL	PRIMERA COMPROBACIÓN	ACCIÓN INMEDIATA	ACCIÓN DE FONDO
Suben los timeouts	<code>timed_out_ratio</code> y p95 de latencia.	¿Cola, retrieval, modelo o tool?	Reducir entrada nueva, activar ruta degradada o ampliar workers.	Ajustar SLO, capacidad y límites por tarea.
Fallan contratos	<code>contract_failed_ratio</code> .	¿Cambió prompt, modelo o contrato?	Rollback de <code>prompt_version</code> o bloquear release.	Añadir casos al dataset de eval.
Sube coste por aceptada	<code>cost_per_accepted_p95</code> .	¿Tokens, reintentos, modelo o revisión?	Limitar salida, routing a modelo menor o pedir confirmación.	Rediseñar budgets y eval de coste.
Sube TTFT	<code>ttft_p95</code> .	¿Prefill largo o cola de serving?	Limitar contexto, bajar batch o calentar réplica.	Separar rutas largas de interactivas.
Sube TPOT	<code>tpot_p95</code> .	¿Decode lento, memoria o runtime?	Reducir salida máxima o mover tráfico.	Revisar runtime, cuantización y hardware.
Faltan trazas	<code>trace_missing_ratio</code> .	¿Se perdió <code>traceparent</code> en cola o worker?	Elevar sampling de errores y revisar collector.	Test de propagación en CI.
Retrieval flojo	baja cobertura o baja aceptación en RAG.	¿Cambió índice, embedding o corpus?	Volver a índice anterior o bajar confianza.	Eval específica de retrieval.

Fíjate en el patrón: el runbook separa acción inmediata de acción de fondo. La inmediata protege el servicio. La de fondo evita repetir el problema.

Árbol de diagnóstico

Un árbol ayuda a que dos personas distintas investiguen de forma parecida. No sustituye criterio, pero evita saltar directamente al modelo.



La frase que me repetiría un ingeniero: **no empieces por cambiar el prompt si la traza dice que el problema vive en cola, coste o contrato.**

Golden traces

Igual que usamos datasets de evaluación, podemos guardar trazas representativas. Una golden trace no es “una traza bonita”; es una ejecución esperada que sirve como patrón.

Ejemplos:

GOLDEN TRACE	QUÉ REPRESENTA	QUÉ DEBERÍA PERMANECER ESTABLE
support_summary_small	Pregunta corta con un documento.	1 retrieval, 1 model call, contrato válido, coste bajo.
rag_long_context	Consulta con varios documentos largos.	Retrieval trazado, prefill visible, coste dentro de presupuesto.
tool_required	Run que necesita una tool concreta.	Tool llamada una vez, permiso correcto, salida validada.
contract_failure	Salida intencionalmente inválida.	El validador bloquea y no se publica.
fallback_route	Ruta degradada cuando el modelo principal no cumple.	Estado y motivo de fallback visibles.

Estas trazas sirven para dos cosas: enseñar a nuevos miembros cómo se ve una run sana, y detectar regresiones de observabilidad. Si actualizas el runtime y la golden trace ya no tiene `model_id` o `contract_version` , has perdido una señal crítica aunque la respuesta final parezca correcta.

Observabilidad específica de RAG y tools

RAG y tools son las dos zonas donde más se suele perder información útil. El modelo responde al final, pero la causa del problema puede estar antes.

Para RAG, yo exigiría:

SEÑAL	POR QUÉ IMPORTA
<code>retrieval_index_version</code>	Permite saber qué índice respondió.
<code>embedding_model_id</code>	Cambia el espacio semántico de búsqueda.
<code>query_tokens</code>	Una query enorme puede explicar coste y ruido.
<code>top_k</code> y <code>reranker_version</code>	Cambian cobertura y precisión.
<code>source_ids</code> y <code>chunk_ids</code>	Permiten reconstruir evidencia sin guardar texto completo.
<code>retrieval_latency_ms</code>	Separa lentitud de búsqueda de lentitud de modelo.
<code>empty_retrieval</code>	Detecta respuestas sin contexto real.

Para tools:

SEÑAL	POR QUÉ IMPORTA
<code>tool_name</code> y <code>tool_version</code>	Una tool puede cambiar aunque el modelo no cambie.
<code>permission_scope</code>	Permite revisar si la acción estaba permitida.
<code>args_schema_version</code>	Los argumentos también tienen contrato.
<code>duration_ms</code>	Separa tools lentas de modelo lento.
<code>result_state</code>	Distingue éxito, timeout, cancelación o respuesta parcial.
<code>retry_count</code>	Explica coste y latencia.
<code>result_summary</code>	Resume sin guardar toda la salida.

La regla: si RAG o tools influyen en la respuesta, deben aparecer en la traza. Si no aparecen, el modelo cargará con culpas que quizá no son suyas.

Coste de observar

Observar también cuesta. Cuesta almacenamiento, ingestión, consultas, retención, mantenimiento y atención humana. La observabilidad mal diseñada puede volverse otro sistema que nadie entiende.

Costes típicos:

COSTE	CÓMO APARECE	CONTROL
Ingesta	Demasiados logs o spans por token.	Agregar eventos, muestrear y evitar contenido repetido.
Cardinalidad	Etiquetas con <code>run_id</code> , <code>user_id</code> o IDs infinitos.	Mover IDs a trazas/logs, no a métricas.
Retención	Guardar trazas completas demasiado tiempo.	Retención por tipo de señal y criticidad.
Consulta	Dashboards lentos o caros.	Paneles mínimos y agregaciones previas.
Privacidad	Contenido sensible en sistemas de observabilidad.	Redacción, hashes, permisos y export control.
Atención	Alertas sin acción.	Owner, runbook y criterio de silencio.

Una buena pregunta antes de añadir señal es: “¿Quién la usará, para qué decisión, durante cuánto tiempo y con qué nivel de acceso?”. Si no sabemos responder, quizá la señal todavía no está madura.

Herramientas que se usan para observar sistemas de IA

Una herramienta de observabilidad no sustituye al contrato de instrumentación. Si tu run no emite `run_id`, `trace_id`, `model_id`, `prompt_version`, `contract_version`, `tokens`, `coste` y `estado final`, ninguna plataforma lo va a adivinar de forma fiable. Lo que sí hace una buena herramienta es recoger esas señales, conservarlas, cruzarlas y convertirlas en consultas, alertas y evaluaciones.

Yo lo separaría en tres capas: estándar de instrumentación, backend de observabilidad general y capa específica de IA.

Estándar e instrumentación

HERRAMIENTA	QUÉ RESUELVE	PROS	CONTRAS
OpenTelemetry	APIs, SDKs, Collector, protocolo OTLP, trazas, métricas, logs y convenciones semánticas.	Evita acoplar el código a un único proveedor; propaga contexto entre API, cola, worker, retrieval y modelo; tiene convenciones para sistemas generativos.	No es “un dashboard”; hay que elegir backend, diseñar atributos y vigilar volumen. La auto-instrumentación no entiende del todo tu dominio.
OpenInference	Convenciones e instrumentación pensadas para IA sobre la idea de OpenTelemetry.	Útil para trazas de LLM, RAG, embeddings y agentes; ayuda a que distintas herramientas entiendan atributos parecidos.	Si tu arquitectura tiene contratos propios, tendrás que mapear campos y completar atributos manualmente.
Traceloop / OpenLLMetry	Auto-instrumentación para frameworks y proveedores de LLM exportando a OpenTelemetry.	Acelera el primer paso: ves llamadas a modelos, prompts, latencias y coste sin escribir todos los spans a mano.	Puede quedarse corta en decisiones de producto: permisos, estado final, versión de contrato o aceptación real suelen requerir spans propios.

La decisión práctica: usa OpenTelemetry como idioma común siempre que puedas. Después decide si quieres que Langfuse, Phoenix, Grafana, Datadog u otra plataforma lea ese idioma.

Backends generales de métricas, logs y trazas

HERRAMIENTA	QUÉ RESUELVE	PROS	CONTRAS
<u>Prometheus</u>	Métricas de series temporales y consultas PromQL.	Muy bueno para SLIs, SLOs, alertas, histogramas, colas, workers y capacidad. Encaja bien con <code>ai_run_latency_seconds</code> o <code>ai_contract_failure_total</code> .	No es una base de trazas. Si metes <code>run_id</code> , <code>user_id</code> o IDs infinitos como labels, rompes cardinalidad y coste.
<u>Grafana</u>	Visualización y cuadros de mando sobre varias fuentes.	Permite construir un dashboard operativo con latencia, coste, errores de contrato, cola, tokens y burn rate.	No corrige una mala instrumentación. Si nombres y etiquetas son pobres, el panel solo lo hace más visible.
<u>Grafana Tempo</u>	Backend de trazas distribuido.	Útil para reconstruir una run completa por <code>trace_id</code> , saltando de API a worker, retrieval, modelo y validador.	Requiere muestreo, retención y atributos bien pensados. Guardar todo al 100% puede salir caro.
<u>Grafana Loki</u>	Logs indexados por etiquetas.	Bueno para logs estructurados cuando quieres correlacionar eventos con una traza sin indexar todo el contenido.	Si conviertes cada campo en etiqueta, vuelve el problema de cardinalidad. Hay que separar labels estables de contenido consultable.
<u>Grafana Mimir</u>	Almacenamiento escalable de métricas compatible con Prometheus.	Útil cuando Prometheus local se queda pequeño o necesitas retención y consulta de métricas a mayor escala.	Añade operación y coste. Para proyectos pequeños puede ser demasiado pronto.
<u>Datadog LLM Observability</u>	APM gestionado con vistas específicas para flujos LLM.	Interesa si la empresa ya usa Datadog: una infraestructura, servicios, trazas LLM, costes y errores en una misma plataforma.	Coste y dependencia de proveedor. Las decisiones de dominio siguen siendo tuyas: contrato, aceptación, versionado y datasets.

No hay una herramienta “correcta” universal. Hay una pila que encaja con tu tamaño, tus restricciones y tu forma de operar. Para un equipo pequeño, Prometheus + Grafana + OpenTelemetry puede ser suficiente. Para un equipo con producto en producción, un APM gestionado puede ahorrar mucho trabajo operativo. Para un curso o laboratorio, Phoenix o Langfuse dan mucha visibilidad sin montar una plataforma enorme.

Herramientas específicas de LLM, RAG y agentes

HERRAMIENTA	QUÉ RESUELVE	PROS	CONTRAS
<u>Langfuse</u>	Trazas LLM, prompts, sesiones, evaluaciones, datasets y coste.	Muy útil para ver conversaciones, versiones de prompt, llamadas a modelo, scoring y regresiones de producto. Tiene opción cloud y open source.	Hay que cuidar qué contenido se envía. Si ya tienes OpenTelemetry, decide qué vive en Langfuse y qué vive en el backend general.
<u>LangSmith</u>	Trazas, datasets, experimentos y evaluación de aplicaciones LLM, especialmente en LangChain/LangGraph.	Fuerte para depurar cadenas, agentes, RAG y comparar ejecuciones con datasets.	Si no usas LangChain, puede seguir sirviendo, pero su ergonomía brilla más dentro de ese ecosistema.
<u>Arize Phoenix</u>	Observabilidad open source para LLM, RAG, embeddings, datasets y evaluaciones.	Muy buena para enseñanza y equipos que quieren ver trazas, spans y evals de RAG sin empezar con una plataforma cerrada.	Requiere despliegue, gobierno de datos y disciplina de versionado igual que cualquier herramienta.
<u>Helicone</u>	Proxy y observabilidad para llamadas a APIs LLM.	Fácil para empezar: centraliza coste, latencia, usuario, proveedor y logs de llamadas sin tocar demasiado el código.	Al ir por proxy, hay que revisar privacidad, disponibilidad, región y qué ocurre si ese proxy cae. No sustituye trazas internas de cola, retrieval o validador.
<u>Braintrust</u>	Evals, experimentos, logging, datasets y comparación de prompts/modelos.	Muy útil cuando el cuello de botella es medir calidad y comparar cambios, no solo ver latencia.	No reemplaza Prometheus, Grafana u OpenTelemetry para operar infraestructura, colas y workers.

Estas herramientas suelen ser más cercanas al trabajo de IA: muestran prompt, respuesta, spans de RAG, scoring, datasets y versiones de experimento. Eso no las convierte en reemplazo del SRE clásico. En una arquitectura sería conviven: una herramienta LLM para entender calidad y una pila general para operar servicio.

Cómo elegir sin perderse

SITUACIÓN	ELECCIÓN RAZONABLE	POR QUÉ
Estás aprendiendo o montando un prototipo serio.	OpenTelemetry en código + Phoenix o Langfuse.	Ves trazas de IA pronto y aprendes el contrato sin casarte con una sola plataforma.
Ya tienes plataforma SRE en la empresa.	OpenTelemetry hacia Datadog, Grafana, Honeycomb o similar + herramienta de evals.	Aprovechas alertas, dashboards y permisos existentes, y añades evaluación LLM donde aporta.
Tu problema principal es coste de API.	Helicone o trazas propias con coste por <code>run_id</code> y <code>tenant_tier</code> .	Necesitas saber quién gasta, en qué tarea, con qué modelo y con qué resultado.
Tu problema principal es calidad.	Braintrust, LangSmith, Phoenix o Langfuse con datasets versionados.	Necesitas comparar prompts, modelos, RAG y contratos con casos repetibles.
Tu problema principal es latencia.	OpenTelemetry + backend de trazas + métricas de TTFT, TPOT, cola y batch.	La causa puede estar en prefill, decode, cola, retrieval o red, no solo en el modelo.
Tu problema principal es privacidad.	Instrumentación propia con hashes/IDs + backend controlado + muestreo estricto.	Conviene enviar solo metadatos, resúmenes y muestras autorizadas.

Mi regla para ingeniería sería esta: primero define **qué decisión quieres poder tomar**. Después eliges herramienta. Si empiezas por la herramienta, acabas midiendo lo que la interfaz trae por defecto, no lo que tu sistema necesita.

Lo que ninguna herramienta arregla por ti

DEUDA	POR QUÉ SIGUE SIENDO TUYA
No tener <code>run_id</code> estable.	No podrás unir API, cola, modelo, tools y cierre de producto.
No versionar modelo, prompt, contrato e índice.	No podrás explicar regresiones después de un cambio.
No separar métrica, log y traza.	Acabarás metiendo IDs infinitos en métricas o texto sensible en logs.
No definir SLO.	Tendrás datos, pero no criterio operativo.
No tener runbook.	La alerta llegará, pero nadie sabrá qué hacer primero.
No tener datasets ni golden traces.	Verás síntomas, pero no podrás comparar cambios de forma repetible.
No acordar retención y permisos.	La observabilidad se convierte en otro lugar donde hay datos que nadie gobierna.

En el día a día

En un proyecto real, la observabilidad empieza en el diseño del contrato, no al final. Si la API no tiene `run_id` , no podrás correlacionar. Si el worker no propaga `trace_id` , tendrás dos historias. Si la llamada al modelo no guarda `model_id` , no sabrás qué versión falló. Si no guardas `contract_version` , no sabrás si falló el modelo o cambió el schema.

Una checklist mínima para producción:

CAPA	DEBE EMITIR
API	<code>request_id</code> , <code>run_id</code> , <code>trace_id</code> , tenant, ruta, contrato, tamaño.
Cola	tiempo de espera, prioridad, TTL, reintentos, descarte.
Retrieval	índice, versión, top_k, documentos, latencia, cobertura.
Tools	nombre, argumentos resumidos, permisos, duración, salida resumida, error tipado.
Modelo	proveedor, modelo, tokens, TTFT, TPOT, coste, estado.
Validador	schema, versión, campos ausentes, campos extra, resultado.
Cierre	estado final, aceptada/no aceptada, coste total, latencia total, owner.
Herramientas	OpenTelemetry, backend de métricas/trazas/logs y, si aporta valor, herramienta específica para LLM/RAG/evals.

Por qué debería importarte

Porque sin observabilidad todo se convierte en opinión. Producto cree que el modelo falla. Ingeniería cree que la cola está saturada. Soporte cree que el usuario escribió mal. Finanzas cree que el proveedor subió coste. Nadie tiene una historia compartida.

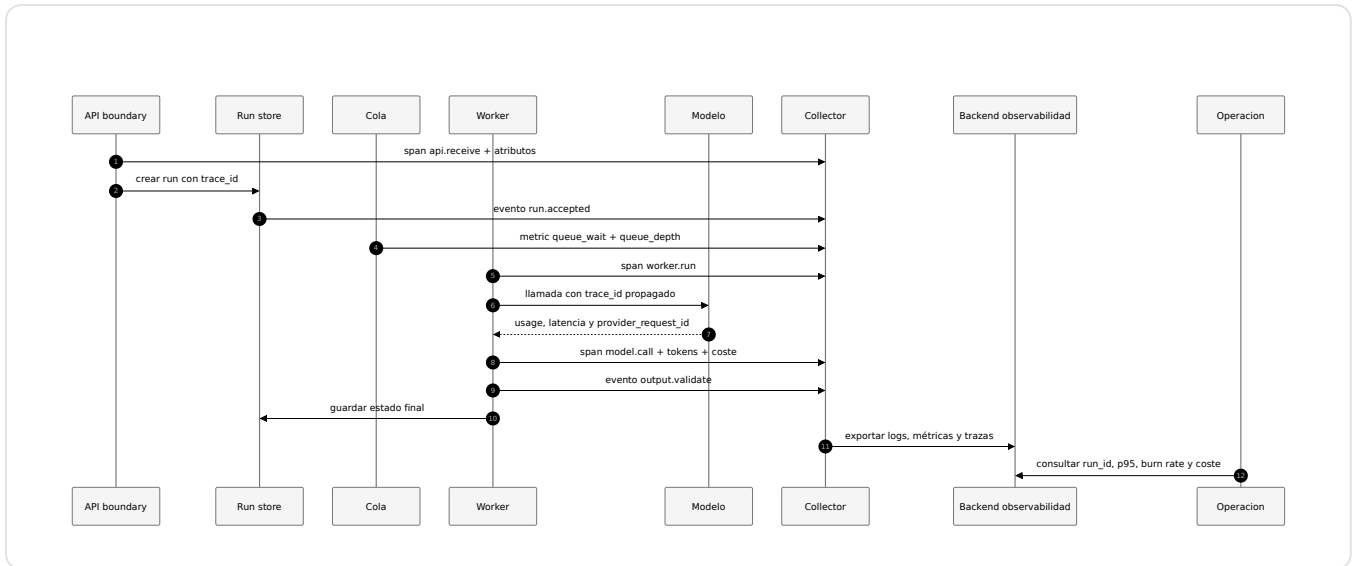
Con observabilidad, las conversaciones se vuelven verificables: “el 70% de las runs caras vienen de `task=legal_summary` con contexto p95 de 42.000 tokens”, “la versión `prompt@1.8` duplicó `contract_failed`”, “el p99 subió por cola de serving, no por retrieval”, “el coste por aceptada bajó aunque subió el coste por llamada”.

Dónde solía tropezar yo

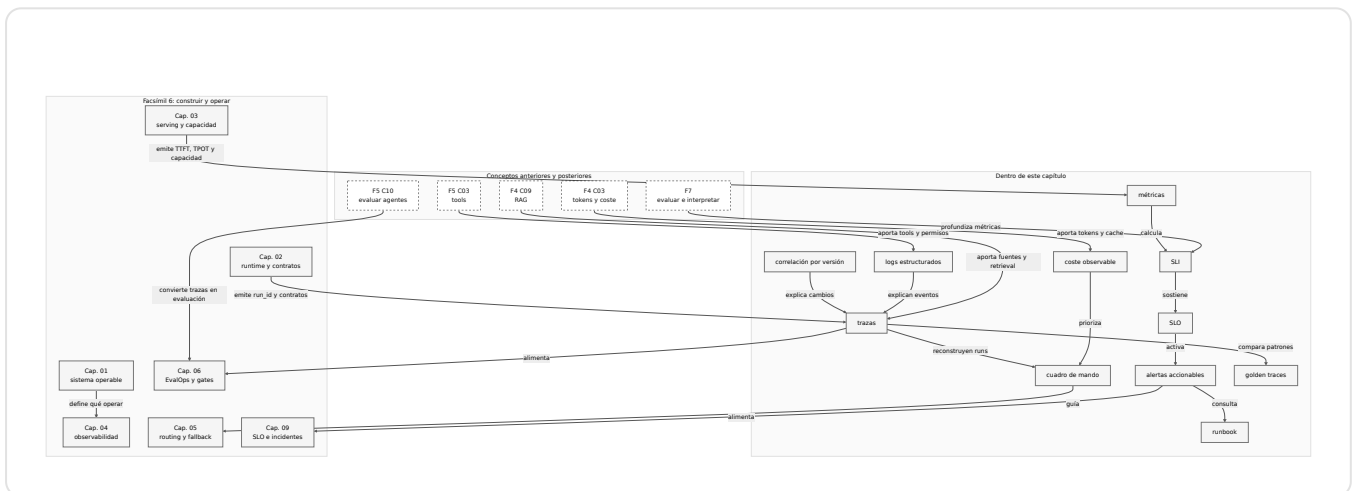
TROPIEZO	POR QUÉ ES UN PROBLEMA	ANTÍDOTO
Guardar demasiado contenido	Complica privacidad, coste y lectura.	Guardar IDs, hashes, tamaños, versiones y muestras controladas.
Medir solo la media	La media oculta la cola de usuarios que peor lo pasan.	Usar histogramas, p95, p99 y trazas lentas.
Meter <code>run_id</code> como etiqueta de métrica	Explota la cardinalidad y encarece el sistema.	<code>run_id</code> vive en trazas y logs, no en series agregadas.
Alertar por todo	El equipo deja de confiar en las alertas.	Alertar solo por síntomas accionables y SLO burn rate.
No propagar <code>trace_id</code>	Cada servicio cuenta una historia distinta.	Propagar contexto desde API hasta worker y modelo.
Separar coste de calidad	Parece barato lo que luego se descarta.	Medir coste por run aceptada.
No versionar prompts y contratos	No sabes qué cambio rompió qué.	Emitir <code>prompt_version</code> , <code>model_id</code> y <code>contract_version</code> .
Pensar que observabilidad se añade al final	Luego no existen los puntos de instrumentación.	Diseñar señales junto al contrato de runtime.

Cómo encaja todo

Primero, el flujo de telemetría de una run:



Y el mapa conceptual:



La observabilidad es el puente entre construir y mejorar. Sin señales, un cambio de prompt, modelo, índice o runtime se evalúa por sensaciones. Con señales, se evalúa por calidad, coste, latencia y trazabilidad.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Observabilidad	Capacidad de entender el estado interno de un sistema a partir de señales emitidas.
Log	Evento estructurado que registra algo ocurrido.
Métrica	Medida numérica agregable en el tiempo.
Traza	Historia completa de una petición o run, formada por spans.
Span	Unidad de trabajo dentro de una traza.
Atributo	Dato clave-valor que describe un span, log o métrica.
Evento de span	Marca puntual dentro de un span, como <code>contract.failed</code> .
Trace context	Información que permite propagar una traza entre servicios.
Cardinalidad	Número de combinaciones posibles de etiquetas en una métrica.
Histograma	Métrica que agrupa observaciones por rangos para calcular percentiles.
SLI	Indicador medible del comportamiento de un servicio.
SLO	Objetivo interno medible basado en un SLI.
Error budget	Margen permitido de incumplimiento del SLO.
Burn rate	Velocidad a la que se consume el presupuesto de error.
Sampling	Selección parcial de señales para controlar coste y volumen.
Retención	Tiempo durante el que se conserva una señal.
Coste por run aceptada	Coste total dividido entre runs que realmente sirven para producto.
Correlación por versión	Capacidad de filtrar señales por modelo, prompt, contrato, índice, runtime y release.
Runbook	Guía operativa que indica qué mirar y qué hacer cuando aparece un síntoma.
Golden trace	Traza representativa y versionada que enseña cómo debería verse una run sana o controlada.

TÉRMINO	DEFINICIÓN
Tail sampling	Muestreo que decide conservar una traza después de ver que fue lenta, cara o fallida.
Coste de observabilidad	Coste de ingesta, almacenamiento, consulta, retención y atención humana asociado a observar el sistema.
OpenTelemetry	Estándar abierto para instrumentar y exportar logs, métricas y trazas.
Prometheus	Sistema de métricas de series temporales muy usado para SLIs, SLOs y alertas.
Grafana	Plataforma de visualización para dashboards y exploración de señales.
APM	Application Performance Monitoring: herramientas para observar servicios, latencia, errores, dependencias y recursos.
Observabilidad LLM	Capa de herramientas centrada en prompts, respuestas, coste, trazas de RAG, agentes, datasets y evaluaciones.

Antes de pasar página

- ☐ ¿Puedes explicar con tus palabras la diferencia entre log, métrica y traza?
- ☐ ¿Puedes diseñar una traza mínima para una run de IA?
- ☐ ¿Puedes decir qué atributos debe tener `model.call` ?
- ☐ ¿Puedes explicar por qué `run_id` no debería ser etiqueta de métrica?
- ☐ ¿Puedes definir un SLI de salida aceptada?
- ☐ ¿Puedes calcular burn rate si el SLO es 99% y el error observado es 5%?
- ☐ ¿Puedes explicar por qué HTTP 200 no siempre significa éxito de producto?
- ☐ ¿Puedes diseñar una métrica de coste por run aceptada?
- ☐ ¿Puedes decidir qué señales guardarías al 100% y cuáles muestrearías?
- ☐ ¿Puedes nombrar una alerta accionable y una alerta que eliminarías?
- ☐ ¿Puedes explicar por qué p95/p99 importan en IA interactiva?
- ☐ ¿Puedes conectar observabilidad con EvalOps y gates?
- ☐ ¿Puedes decir qué versiones debe emitir una run para investigar una regresión?
- ☐ ¿Puedes diseñar un dashboard mínimo de seis paneles sin llenarlo de ruido?
- ☐ ¿Puedes escribir una alerta con condición, owner y runbook?

- ☐ ¿Puedes seguir el árbol de diagnóstico para separar TTFT, TPOT, contrato, coste y retrieval?
- ☐ ¿Puedes explicar para qué sirve una golden trace?
- ☐ ¿Puedes decir qué señales de RAG y tools no deberían faltar?
- ☐ ¿Puedes justificar qué campos prohibirías como etiquetas de métricas?
- ☐ ¿Puedes elegir entre OpenTelemetry, Prometheus/Grafana y una herramienta LLM específica según el problema?
- ☐ ¿Puedes explicar por qué una herramienta no reemplaza `run_id` , versionado, SLO y runbook?
- ☐ ¿Puedes comparar instrumentación por SDK, auto-instrumentación y proxy?

En resumen

IDEA	QUÉ DEBES LLEVARTE
Observabilidad es reconstrucción, no acumulación.	Debes poder explicar una run concreta sin guardar contenido de más.
Logs, métricas y trazas cumplen trabajos distintos.	Logs explican eventos, métricas agregan comportamiento y trazas muestran causalidad.
En IA el éxito no es solo HTTP 200.	Hay que medir contrato, latencia, coste, calidad, revisión y estado final.
La cardinalidad se diseña.	Algunas cosas van en métricas; otras en trazas o logs.
Coste y calidad deben leerse juntos.	El coste útil es coste por run aceptada, no solo precio por token.
Las alertas deben ser accionables.	Un buen SLO y burn rate ayudan a alertar menos y mejor.
Las versiones son parte de la señal.	Sin <code>model_id</code> , <code>prompt_version</code> , <code>contract_version</code> e índice no sabes qué cambio explicar.
El runbook convierte una alerta en trabajo.	Una alerta útil trae owner, primera comprobación y acción inmediata.
RAG y tools deben dejar huella.	Si influyen en la respuesta, deben aparecer en la traza con versión, duración y resultado resumido.
Las herramientas amplifican una buena instrumentación.	OpenTelemetry, Grafana, Langfuse, Phoenix o Datadog ayudan, pero no inventan contratos, versiones ni SLOs.

Para saber más

- Dean, J. y Barroso, L. A. (2013). *The Tail at Scale*. Communications of the ACM, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>
- Ewaschuk, R. (2016). *Monitoring Distributed Systems*. En B. Beyer, C. Jones, J. Petoff y N. R. Murphy (eds.), *Site Reliability Engineering*. <https://sre.google/sre-book/monitoring-distributed-systems/>
- Little, J. D. C. (1961). *A Proof for the Queuing Formula: $L = \lambda W$* . Operations Research, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>
- OpenTelemetry. (2026). *Logs*. <https://opentelemetry.io/docs/concepts/signals/logs/>
- OpenTelemetry. (2026). *Metrics*. <https://opentelemetry.io/docs/concepts/signals/metrics/>
- OpenTelemetry. (2026). *Semantic Conventions for Generative AI Systems*. <https://opentelemetry.io/docs/specs/semconv/gen-ai/>
- OpenTelemetry. (2026). *Traces*. <https://opentelemetry.io/docs/concepts/signals/traces/>
- OpenTelemetry. (2026). *Tracing API*. <https://opentelemetry.io/docs/specs/otel/trace/api/>
- Arize. (2026). *Phoenix*. <https://arize.com/docs/phoenix>
- Arize. (2026). *OpenInference*. <https://arize-ai.github.io/openinference/>
- Braintrust. (2026). *Documentation*. <https://www.braintrust.dev/docs>
- Datadog. (2026). *LLM Observability*. https://docs.datadoghq.com/llm_observability/
- Grafana Labs. (2026). *Grafana documentation*. <https://grafana.com/docs/>
- Grafana Labs. (2026). *Grafana Loki documentation*. <https://grafana.com/docs/loki/latest/>
- Grafana Labs. (2026). *Grafana Mimir documentation*. <https://grafana.com/docs/mimir/latest/>
- Grafana Labs. (2026). *Grafana Tempo documentation*. <https://grafana.com/docs/tempo/latest/>
- Helicone. (2026). *Documentation*. <https://docs.helicone.ai/>
- LangChain. (2026). *LangSmith documentation*. <https://docs.langchain.com/langsmith/home>
- Langfuse. (2026). *Documentation*. <https://langfuse.com/docs>
- Traceloop. (2026). *OpenLLMetry documentation*. <https://www.traceloop.com/docs/openllmetry>
- Prometheus. (2026). *Metric and label naming*. <https://prometheus.io/docs/practices/naming/>
- Wilkinson, J. (2018). *Alerting on SLOs*. En B. Beyer, N. R. Murphy, D. Rensin, K. Kawahara y S. Thorne (eds.), *The Site Reliability Workbook*. <https://sre.google/workbook/alerting-on-slos/>
- World Wide Web Consortium. (2021). *Trace Context Level 2*. <https://www.w3.org/TR/trace-context-2/>

Notas

1. Ewaschuk, R. (2016). *Monitoring Distributed Systems*. En B. Beyer, C. Jones, J. Petoff y N. R. Murphy (eds.), *Site Reliability Engineering*. <https://sre.google/sre-book/monitoring-distributed-systems/>. Consultado el 27 de mayo de 2026. ↵
2. OpenTelemetry. (2026). *Traces*. <https://opentelemetry.io/docs/concepts/signals/traces/>. Consultado el 27 de mayo de 2026. ↵
3. OpenTelemetry. (2026). *Tracing API*. <https://opentelemetry.io/docs/specs/otel/trace/api/>. Consultado el 27 de mayo de 2026. ↵
4. OpenTelemetry. (2026). *Semantic Conventions for Generative AI Systems*. <https://opentelemetry.io/docs/specs/semconv/gen-ai/>. Consultado el 27 de mayo de 2026. ↵
5. OpenTelemetry. (2026). *Logs*. <https://opentelemetry.io/docs/concepts/signals/logs/>. Consultado el 27 de mayo de 2026. ↵
6. OpenTelemetry. (2026). *Metrics*. <https://opentelemetry.io/docs/concepts/signals/metrics/>. Consultado el 27 de mayo de 2026. ↵
7. World Wide Web Consortium. (2021). *Trace Context Level 2*. <https://www.w3.org/TR/trace-context-2/>. Consultado el 27 de mayo de 2026. ↵
8. Ewaschuk, 2016. ↵
9. Wilkinson, J. (2018). *Alerting on SLOs*. En B. Beyer, N. R. Murphy, D. Rensin, K. Kawahara y S. Thorne (eds.), *The Site Reliability Workbook*. <https://sre.google/workbook/alerting-on-slos/>. Consultado el 27 de mayo de 2026. ↵
10. Prometheus. (2026). *Metric and label naming*. <https://prometheus.io/docs/practices/naming/>. Consultado el 27 de mayo de 2026. ↵
11. Dean, J. y Barroso, L. A. (2013). *The Tail at Scale*. Communications of the ACM, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>. ↵
12. Little, J. D. C. (1961). *A Proof for the Queuing Formula: $L = \lambda W$* . Operations Research, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>. ↵
13. Apdex Alliance; Sevcik, P. (2005). *Defining the Application Performance Index*. El índice Apdex es un estándar abierto para resumir la satisfacción de rendimiento en un valor entre 0 y 1. ↵
14. Cochran, W. G. (1977). *Sampling Techniques* (3.^a ed.). Wiley. Texto de referencia sobre el error estándar de una proporción estimada a partir de una muestra. ↵

- .5. Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8), 861-874. <https://doi.org/10.1016/j.patrec.2005.10.010>. Relaciona sensibilidad, tasa de falsos positivos y prevalencia mediante el teorema de Bayes. ↵
- .6. Dunning, T. (2021). The t-digest: Efficient estimates of distributions. *Software Impacts*, 7, 100049. <https://doi.org/10.1016/j.simpa.2020.100049>. Describe el t-digest para estimar cuantiles en flujos de datos de forma agregable y precisa en las colas. ↵
-

Cuadernos para practicar

Escanea el código con el móvil para abrir el cuaderno en Google Colab y ejecutarlo sin instalar nada.



Observabilidad: la caja negra que sí puedes mirar

<https://colab.research.google.com/github/686f6c61/iaparagentecuriosa-notebooks/blob/main/Facs%C3%ADmil%2006/01-observabilidad-servicio.ipynb>

Capítulo 05: Routing, fallback y presupuestos por tarea

Qué deberías poder hacer al terminar

En el capítulo 02 convertimos una petición en una run con estado y contrato. En el capítulo 03 vimos que servir modelos significa convivir con memoria, colas, prefill, decode y capacidad real. En el capítulo 04 aprendimos a observar una run para reconstruir qué ocurrió.

Ahora toca decidir **por dónde debe ir cada run antes de ejecutarla**.

No todas las tareas merecen el mismo modelo, la misma latencia, el mismo coste ni la misma estrategia de recuperación. Una pregunta corta de soporte no debería consumir el mismo presupuesto que un análisis legal largo. Una extracción JSON estricta no debería ir por una ruta que no soporta salida estructurada. Una tarea interactiva no debería quedarse esperando detrás de un lote nocturno. Y una dependencia lenta no debería provocar diez reintentos que empeoran el sistema.

Al terminar, deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Diseñar un router operativo.	Separas reglas, señales, catálogo de rutas, presupuesto y trazabilidad.
Definir presupuestos por tarea.	Pones límites de latencia, coste, tokens, retries y salida esperada.
Explicar fallback sin vender atajos.	Distingues degradar, reintentar, cambiar proveedor, cambiar modelo o parar con estado claro.
Calcular coste esperado de una ruta.	Tienes en cuenta entrada, salida, probabilidad de reintento y aceptación final.
Evitar tormentas de reintentos.	Usas timeout, backoff, jitter, circuit breaker y límites de concurrencia.
Instrumentar la decisión.	Registras ruta elegida, alternativas descartadas, motivo y presupuesto consumido.

La idea central: **un sistema serio no llama al modelo “a ver qué pasa”; decide una ruta bajo contrato, presupuesto y señales observables.**

La petición que parece igual pero no lo es

Imagina dos usuarios escribiendo “resume este documento”. La frase parece la misma, pero las condiciones no lo son.

En el primer caso, el documento son cinco párrafos de una política interna y el usuario quiere una respuesta rápida. En el segundo, el documento son setenta páginas, la respuesta debe citar fuentes, el contrato exige JSON, el coste máximo es bajo y la tarea no es interactiva. Si mandamos ambas por la misma ruta, estamos fingiendo que el sistema no entiende su propio trabajo.

El router es la pieza que evita esa ficción. Mira la tarea, el contrato, el presupuesto, la salud de las rutas, la capacidad disponible, la sensibilidad de los datos, el histórico de calidad y decide: local o cloud, modelo pequeño o grande, streaming o batch, RAG o no RAG, salida breve o larga, fallback permitido o no.

Qué no es routing

Routing no es “si falla A, prueba B”. Eso es solo una parte pequeña, y a veces peligrosa, de la historia.

Tampoco es elegir siempre el modelo más potente. El modelo más potente puede ser demasiado caro, lento, innecesario o incompatible con el contrato de salida. En operación, “mejor” no significa “más grande”; significa **suficiente para esta tarea bajo estas restricciones**.

Y tampoco es esconder la decisión dentro de un prompt: “elige tú el mejor modelo”. Algunas decisiones pueden usar un clasificador o un modelo auxiliar, pero la política operativa no debería quedar enterrada en texto. Si cambia el presupuesto, si sube la cola, si una ruta no soporta JSON o si el usuario pertenece a un tenant con región obligatoria, eso debe vivir en una capa explícita.

CONFUSIÓN	QUÉ FALTA
“Router es fallback”	Falta selección inicial, política, coste, capacidad y trazabilidad.
“Uso siempre el modelo grande”	Falta distinguir calidad suficiente de exceso de coste.
“Reintento hasta que salga”	Falta presupuesto, backoff, límite y motivo de parada.
“Si el proveedor va lento, pruebo otro”	Falta saber si el segundo cumple contrato, datos, región y parámetros.
“La ruta la decide el prompt”	Falta una política auditable y versionada.

Inteligencia artificial para gente curiosa · 686f6c61 · CC BY 4.0

Qué sí es un router operativo

Un router es, formalmente, una **función de decisión** (un dispatcher) que mapea las señales de una petición a una ruta:

$$\rho(t, c, b, h, o) \rightarrow r$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
ρ	Función de routing.	Código, política o servicio que elige ruta.
t	Tipo de tarea.	support_summary , legal_rag , json_extract , batch_labeling .
c	Contrato de salida y capacidades requeridas.	JSON estricto, tools, visión, RAG, región, streaming.
b	Presupuesto disponible.	5 segundos, 0,08 EUR, 8000 tokens, 1 retry.
h	Señales de salud y capacidad.	p95, cola, tasa de timeouts, coste p95, proveedor disponible.
o	Observabilidad histórica y evaluación.	aceptación, calidad por ruta, fallos de contrato, golden traces.
r	Ruta elegida.	local_qwen_fast , cloud_frontier_json , rag_batch_worker .

La función puede ser simple o compleja. En un producto pequeño puede ser una tabla YAML con reglas. En un producto grande puede ser un servicio con feature flags, estadísticas por tenant, circuit breakers, evaluaciones offline, rutas canary y aprendizaje sobre resultados. Lo importante no es que sea sofisticado: es que sea **explícito**.

Fecha de corte del estado del arte

Fecha de corte: 28 de mayo de 2026.

Fuentes consultadas: Google SRE sobre sobrecarga y monitorización, AWS Builders Library sobre timeouts, retries, backoff y jitter, OpenRouter Provider Routing, Prompt Caching y Latency and Performance, LiteLLM Router, OpenAI Rate Limits, Latency Optimization, Prompt Caching, Batch API y Flex Processing, Anthropic Rate Limits, Prompt Caching y Cache Diagnostics, Prometheus Metric Naming y OpenTelemetry Semantic Conventions for Generative AI Systems.

Lo estable es el patrón de ingeniería: presupuestos, colas, retries limitados, backoff, jitter, degradación controlada, health signals, circuit breakers, observabilidad y SLOs. Lo cambiante son modelos disponibles, proveedores, precios, límites de cuota, parámetros soportados y sintaxis concreta de routers comerciales.

Google SRE describe la sobrecarga como un problema que exige decidir qué trabajo aceptar, retrasar o rechazar para proteger el servicio.¹ AWS insiste en que timeouts, retries, backoff y jitter deben diseñarse juntos; reintentar sin control puede amplificar el problema que intentas resolver.²

En herramientas de IA, OpenRouter documenta selección de proveedor por precio, latencia, throughput, fallbacks, requisitos de parámetros y políticas de datos.³ LiteLLM documenta un router para balanceo, fallbacks, retries, cooldowns y selección entre despliegues.⁴ OpenAI y Anthropic documentan límites de tasa y cuota que obligan a tratar capacidad y presupuesto como parte de la integración, no como detalle posterior.⁵⁶

La política de ruta por dentro

Un router maduro no decide con una sola señal. Combina restricciones duras y preferencias.

Primero filtra lo que no cumple:

RESTRICCIÓN DURA	PREGUNTA	EJEMPLO
Contrato	¿La ruta soporta lo que pide la salida?	JSON estricto, tool calling, visión, streaming.
Presupuesto	¿Cabe en coste, tiempo y tokens?	<code>max_cost_eur <= 0.08</code> , <code>max_latency_ms <= 6000</code> .
Datos	¿La ruta cumple región, retención y política interna?	Local, UE, no almacenar contenido, solo IDs.
Capacidad	¿La ruta puede aceptar trabajo ahora?	Cola, cuota, workers, rate limit, cooldown.
Compatibilidad	¿Soporta los parámetros necesarios?	<code>temperature</code> , <code>response_format</code> , <code>max_output_tokens</code> .

Después ordena lo que sí cumple:

PREFERENCIA	QUÉ OPTIMIZA	CUÁNDO DOMINA
Menor latencia	Tiempo hasta respuesta útil.	Chat interactivo, copilotos, soporte.
Menor coste	Coste por run aceptada.	Clasificación masiva, tareas de bajo margen.
Mayor calidad	Precisión, razonamiento, formato, evaluación.	Tareas complejas, legales, técnicas o de decisión.
Mayor estabilidad	Menos variación p95/p99.	Productos con SLO estricto.
Menor exposición de datos	Menos salida de datos del entorno controlado.	Documentos internos o datos sensibles.

Para elegir ruta podemos puntuar cada candidata con una función de valor aditiva, la decisión multicriterio de siempre: sumar lo que aporta y restar lo que cuesta, cada término con su peso.⁷

$$S(r) = w_qQ(r) - w_cC(r) - w_lL(r) - w_eE(r) - w_s\sigma(r)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$S(r)$	Puntuación de la ruta r .	0,71 para <code>local_fast</code> ; 0,83 para <code>cloud_json</code> .
$Q(r)$	Calidad esperada normalizada.	0,92 si pasa evals de extracción.
$C(r)$	Coste esperado normalizado.	0,30 si cuesta poco, 0,90 si es caro.
$L(r)$	Latencia esperada normalizada.	p95 de 2,5 s frente a un límite de 6 s.
$E(r)$	Penalización por exposición de datos o restricciones.	0 si todo queda local; 0,6 si sale a cloud sin necesidad.
$\sigma(r)$	Variabilidad o cola de cola larga.	Penaliza rutas con p99 malo aunque p50 parezca bien.
w_q, w_c, w_l, w_e, w_s	Pesos de decisión.	En soporte pesa latencia; en extracción pesa contrato.

Esto no obliga a convertir el router en una fórmula rígida. La fórmula sirve para una idea: una ruta no se elige solo porque “responde bien”, sino porque equilibra calidad, coste, latencia, política y estabilidad.

Catálogo de rutas: la tabla que sostiene al router

Un router sin catálogo acaba siendo un `if` enorme. Al principio parece suficiente, pero pronto nadie sabe qué rutas existen, qué contrato soportan, qué coste tienen, qué dueño las mantiene o qué limitación ocultan.

El catálogo de rutas es un artefacto versionado. Puede vivir en YAML, base de datos, configuración de plataforma o servicio interno, pero debe responder preguntas concretas:

PREGUNTA	CAMPO DEL CATÁLOGO
¿Qué ruta puedo usar para esta tarea?	<code>allowed_tasks</code> , <code>capabilities</code> , <code>contracts</code> .
¿Qué ruta no debo usar con estos datos?	<code>data_policy</code> , <code>region</code> , <code>retention</code> .
¿Cuánto cuesta antes de ejecutarla?	<code>price.input_1k</code> , <code>price.output_1k</code> , <code>tool_costs</code> .
¿Qué SLO suele cumplir?	<code>latency.p50</code> , <code>latency.p95</code> , <code>ttft.p95</code> , <code>tpot.p95</code> .
¿Qué límites tiene ahora mismo?	<code>rpm</code> , <code>tpm</code> , <code>concurrency</code> , <code>context_window</code> .
¿Quién responde si se rompe?	<code>owner</code> , <code>runbook</code> , <code>dashboard</code> .
¿Cómo se degrada?	<code>fallbacks</code> , <code>degraded_mode</code> , <code>stop_state</code> .

Una versión mínima:

version: route-catalog@2026-05-27
owner: ai-runtime

routes:

local_fast:
 provider: local
 model: qwen3-8b-instruct
 runtime: vllm
 owner: ai-platform
 allowed_tasks: [fast_triage, support_fast_answer]
 capabilities: [text, json]
 contracts: [triage_v2, short_answer_v1]
 context_window: 32768
 data_policy:
 region: local
 retention: none
 content_logging: ids_only
 price:
 input_1k_eur: 0.0002
 output_1k_eur: 0.0006
 observed:
 latency_p95_ms: 1800
 timeout_ratio_5m: 0.02
 contract_failure_ratio_24h: 0.012
 limits:
 max_concurrency: 24
 max_output_tokens: 700
 fallback:
 chain: [support_brief_static]
 stop_state: capacity_unavailable

rag_medium_json:
 provider: cloud
 model: medium-json-2026-05
 owner: ai-runtime
 allowed_tasks: [support_summary, policy_qa]
 capabilities: [text, json, rag, citations]
 contracts: [answer_with_sources_v3]
 context_window: 131072
 data_policy:
 region: eu
 retention: none
 content_logging: hash_and_ids
 price:
 input_1k_eur: 0.002
 output_1k_eur: 0.006
 observed:
 latency_p95_ms: 4200
 timeout_ratio_5m: 0.05
 contract_failure_ratio_24h: 0.018
 limits:
 rpm: 600

```
tpm: 1200000
max_output_tokens: 1200
fallback:
  chain: [rag_small_brief, batch_review]
  stop_state: no_route_under_budget
```

El catálogo también evita una trampa frecuente: pensar que “modelo” y “ruta” son lo mismo. Una ruta incluye modelo, proveedor, runtime, región, parámetros permitidos, contrato, límites, observabilidad, owner y fallback. Dos rutas pueden usar el mismo modelo y comportarse distinto porque cambian proveedor, cuantización, template, latencia, región o versión del gateway.

Presupuesto por tarea

Un presupuesto de tarea es un contrato operativo, un vector de límites. No dice «usa poco»: dice cuánto se puede gastar, esperar y reintentar.

$$B_t = \langle T_{max}, C_{max}, I_{max}, O_{max}, R_{max}, A_{max} \rangle$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
B_t	Presupuesto para la tarea t .	Presupuesto de <code>support_summary</code> .
T_{max}	Latencia máxima.	6000 ms.
C_{max}	Coste máximo.	0,08 EUR.
I_{max}	Tokens máximos de entrada.	12.000 tokens.
O_{max}	Tokens máximos de salida.	600 tokens.
R_{max}	Reintentos máximos.	1 retry.
A_{max}	Acciones o llamadas externas máximas.	2 tools, 1 retrieval, 0 escrituras.

Ejemplo de presupuestos:

TAREA	LATENCIA	COSTE	TOKENS	REINTENTOS	ruta preferida
support_fast_answer	2500 ms	0,02 EUR	4000 entrada / 300 salida	0	modelo pequeño o caché.
support_summary	6000 ms	0,08 EUR	12000 entrada / 600 salida	1	RAG + modelo medio.
legal_rag_review	30000 ms	0,60 EUR	60000 entrada / 2000 salida	1	modelo fuerte + citas + revisión.
batch_labeling	120000 ms	0,01 EUR por ítem	2000 entrada / 50 salida	2	lote barato, sin streaming.
json_extract_strict	8000 ms	0,05 EUR	8000 entrada / 800 salida	1	ruta con salida estructurada fiable.

La clave está en el adjetivo “por tarea”. Un presupuesto global tipo “máximo 20.000 tokens” no dice casi nada. La operación necesita saber si esos tokens pertenecen a una conversación interactiva, a un lote, a una extracción o a un resumen con citas.

Cuotas y rate limits: capacidad externa también es arquitectura

Cuando usas proveedores externos, la capacidad no depende solo de tu código. También depende de cuotas de peticiones, tokens, concurrencia, región, tier, modelo y organización. OpenAI y Anthropic documentan límites de tasa que pueden expresarse en peticiones, tokens u otros contadores según producto, modelo o plan.⁸⁹

Para un router, esos límites no son un error inesperado: son una entrada de decisión.

Para no saturar un proveedor, la regla de capacidad es una desigualdad de tasa: las llegadas por los tokens esperados de cada una no deben superar el límite de la ruta, dejando un margen de seguridad.

$$\lambda_t \cdot E[I_t + O_t] \leq TPM_{ruta} \cdot \alpha$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
λ_t	Runs por minuto de la tarea t .	120 resúmenes por minuto.
$E[I_t + O_t]$	Tokens esperados por run.	7500 entrada + 500 salida.
TPM_{ruta}	Tokens por minuto permitidos en la ruta.	1.200.000 TPM.
α	Margen operativo que no queremos superar.	0,75 para dejar colchón.

Si $120 \cdot 8000 = 960000$ tokens por minuto y la ruta tiene 1.200.000 TPM, parece caber. Pero con $\alpha = 0,75$, el límite operativo sería 900.000 TPM. El router debería empezar a derivar parte del tráfico, pasar una clase a batch o bajar tokens de entrada.

Lo mismo para peticiones por minuto:

$$\lambda_t \leq RPM_{ruta} \cdot \alpha$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
RPM_{ruta}	Peticiones por minuto permitidas.	600 RPM.
λ_t	Llegada esperada.	520 RPM.
α	Margen de seguridad.	0,80.

Con $\alpha = 0,80$, 600 RPM se convierten en 480 RPM operativos. Si llegan 520, no conviene esperar al error de cuota; conviene actuar antes.

SEÑAL	ACCIÓN RAZONABLE DEL ROUTER
<code>quota_remaining_tpm</code> baja rápido.	Reducir contexto, cambiar ruta o mandar tareas no urgentes a batch.
<code>rpm_utilization</code> supera margen.	Activar backpressure o repartir entre rutas compatibles.
<code>retry_after_ms</code> cabe en presupuesto.	Esperar con backoff y jitter.
<code>retry_after_ms</code> no cabe.	Cambiar ruta o cerrar con estado claro.
<code>context_window</code> insuficiente.	Comprimir, hacer RAG, dividir tarea o rechazar con explicación.

El punto de ingeniería: no esperes a que el proveedor te diga “no puedo”. Tu router debería saber antes si la ruta está cerca del límite.

Coste esperado y coste útil

El coste visible de una llamada no es el coste de producto. Si una ruta falla contrato, requiere reintento o produce una respuesta que se descarta, ha consumido dinero sin producir utilidad.

El coste esperado de una run es la suma de sus partes, con la probabilidad de reintento incluida. Es una esperanza matemática, no un teorema:

$$E[C_{run}] = C_{in} + C_{out} + p_{retry}(C_{in}^{retry} + C_{out}^{retry}) + C_{tool} + C_{retrieval}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$E[C_{run}]$	Coste esperado de una run.	0,047 EUR.
C_{in}	Coste de tokens de entrada.	10.000 tokens de contexto.
C_{out}	Coste de tokens de salida.	500 tokens generados.
p_{retry}	Probabilidad de necesitar reintento.	0,08.
$C_{in}^{retry}, C_{out}^{retry}$	Coste de un intento adicional.	Puede ser menor si recortamos contexto.
C_{tool}	Coste de tools o servicios externos.	Búsqueda, OCR, base vectorial, API externa.
$C_{retrieval}$	Coste de recuperación y reranking.	Embeddings, vector DB, reranker.

Y el coste que de verdad importa es el **coste por tarea aceptada** (cost per successful task): el coste total repartido entre las ejecuciones que pasan el control.

$$C_{util} = \frac{\sum C_{run}}{N_{aceptadas}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
C_{util}	Coste por run aceptada.	0,11 EUR por respuesta útil.
$\sum C_{run}$	Coste total de runs intentadas.	11 EUR.
$N_{aceptadas}$	Runs que pasan contrato y sirven al usuario.	100.

Esta fórmula es incómoda porque revela rutas “baratas” que salen caras. Un modelo pequeño puede costar poco por token, pero si falla más el contrato, reintenta más o exige revisión, el coste útil puede subir.

Tipos de routing que sí aparecen en producción

No hay un único tipo de router. Hay familias.

TIPO	CÓMO DECIDE	SIRVE CUANDO	CAUTION
Reglas estáticas	<code>task == X -> ruta Y .</code>	Producto pequeño, compliance claro, pocas tareas.	Se queda rígido si crece el catálogo.
Routing por contrato	Filtra por soporte de JSON, tools, visión, región o streaming.	La salida tiene requisitos fuertes.	Hay que mantener capacidades por ruta.
Routing por coste	Ordena por coste estimado bajo una calidad mínima.	Tareas masivas o de bajo margen.	Barato no equivale a aceptable.
Routing por latencia	Prefiere p95/p99 menor o cola más corta.	Experiencia interactiva.	Puede elegir rutas más caras.
Routing por calidad	Usa evals, golden traces y resultados históricos.	Tareas donde el error cuesta más que la latencia.	Necesita datasets vivos y versionados.
Routing por salud	Evita rutas con timeouts, cuota agotada o cola alta.	Operación real con proveedores y runtimes cambiantes.	Sin observabilidad, se convierte en intuición.
Routing híbrido	Combina reglas, score, salud, coste y contrato.	Sistemas de IA en producción.	Requiere trazabilidad y ownership.

Lo importante: un router puede empezar simple y evolucionar. Lo peligroso es empezar opaco.

Compatibilidad real entre proveedores

Una API compatible no garantiza comportamiento compatible. Dos rutas pueden aceptar un cuerpo parecido y aun así diferir en lo que importa.

OpenRouter documenta opciones para exigir proveedores que soporten parámetros concretos y para filtrar por políticas de datos, proveedores, cuantización, precio, latencia o throughput.¹⁰ Esa idea es general: antes de fallback o routing cruzado, hay que comprobar compatibilidad.

SUPERFICIE	QUÉ COMPARAR	POR QUÉ IMPORTA
Salida estructurada	JSON mode, schema estricto, tool calling, validación.	Una ruta puede “intentar JSON” y otra garantizar contrato.
Streaming	Formato de eventos, fin de stream, errores parciales.	El cliente puede depender de eventos concretos.
Tools	Schema, tool choice, llamadas paralelas, argumentos.	Cambia cómo el modelo decide usar herramientas.
Contexto	Ventana, tokenizador, coste de entrada, compaction.	El mismo texto no ocupa igual ni cabe igual.
Parámetros	<code>temperature</code> , <code>top_p</code> , <code>seed</code> , <code>max_tokens</code> , razonamiento.	El nombre puede existir, pero su efecto no ser idéntico.
Seguridad y datos	Región, retención, logging, entrenamiento con datos.	No todas las rutas sirven para todos los tenants.
Errores	Códigos, <code>retry_after</code> , timeouts, cuota.	El gateway debe normalizar estados para el router.
Precios	Input, output, caché, tools, razonamiento, lote.	El coste esperado necesita la factura completa.
Versionado	Alias flotante o versión fija.	Un alias puede cambiar comportamiento sin cambiar tu código.

Por eso el catálogo debería distinguir `supports_json_schema=true` de `usually_returns_json=true` . Lo primero es una capacidad. Lo segundo es una esperanza.

Un ejemplo de matriz de compatibilidad:

RUTA	CAPACIDADES DECLARADAS	ENTORNO	FALLBACK PERMITIDO
local_fast	JSON estricto, streaming, sin tools. Contexto 32768.	Local.	Sí, a respuesta breve.
rag_medium_json	JSON estricto, tools, streaming, RAG y citas. Contexto 131072.	UE.	Sí, a rag_small_brief .
frontier_review	JSON estricto, tools, streaming y contexto largo.	Proveedor .	Solo si presupuesto lo permite.
batch_cheap	JSON estricto, sin streaming, sin tools. Contexto 32768.	Proveedor .	No interactivo.

La matriz no es burocracia. Es lo que impide que un fallback rompa el contrato.

Fallback no significa “hacer cualquier cosa”

Fallback es una ruta alternativa con contrato. No es una excusa para devolver algo distinto sin avisar.

Hay varios tipos:

TIPO DE FALLBACK	QUÉ CAMBIA	EJEMPLO
Mismo modelo, otro proveedor	Cambia endpoint o región.	Mismo modelo servido por otro proveedor compatible.
Otro modelo equivalente	Cambia modelo manteniendo contrato.	Modelo B que soporta JSON y calidad mínima.
Modelo menor	Reduce coste o latencia.	Respuesta breve, menos contexto, menos razonamiento.
Ruta local	Evita dependencia externa.	Modelo local para resumen simple cuando cloud no conviene.
Ruta batch	Cambia expectativa temporal.	“Lo dejamos procesando y avisamos cuando esté”.
Respuesta parcial	Devuelve lo seguro y marca lo pendiente.	“Puedo resumir, pero no validar citas ahora”.
Parada explícita	No ejecuta más.	<code>budget_exhausted</code> , <code>contract_not_supported</code> , <code>capacity_unavailable</code> .

La regla de oro: **fallback debe ser igual o más conservador que la ruta principal**. Si la ruta principal no puede ejecutar una acción con el contrato correcto, el fallback no debería inventar más capacidad. Debe reducir alcance, pedir confirmación, pasar a batch o cerrar con estado comprensible.

Reintentos, backoff y jitter

Un retry solo tiene sentido si el problema puede desaparecer al volver a intentar. Un error de contrato no se arregla repitiendo igual. Una cuota agotada no se arregla enviando más rápido. Una salida inválida quizá se arregla con un retry si cambiamos instrucción, temperatura o modelo, pero hay que presupuestarlo.

Política mínima:

CASO	RETRY	MOTIVO
Timeout de red puntual	Sí, con backoff y límite.	Puede ser transitorio.
Rate limit	Sí, si <code>retry_after</code> cabe en presupuesto.	Si no cabe, cambiar ruta o parar.
JSON inválido	Una vez, con reparación o ruta más fiable.	Repetir igual suele repetir el fallo.
Presupuesto agotado	No.	Ya no hay margen operativo.
Contrato no soportado	No.	La ruta era incorrecta.
Error de permisos	No.	No es un problema temporal.

Backoff exponencial simple:

$$delay_i = \min(D_{max}, D_0 \cdot 2^i) + U(0, J)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$delay_i$	Espera antes del retry i .	400 ms, 850 ms, 1700 ms.
D_0	Espera inicial.	250 ms.
D_{max}	Espera máxima.	3000 ms.
i	Número de retry.	0, 1, 2.
$U(0, J)$	Ruido aleatorio entre 0 y J .	Jitter de 0 a 150 ms.

El jitter parece un detalle menor, pero evita que muchos clientes despierten a la vez y vuelvan a presionar la misma dependencia. En sistemas con IA, donde cada intento puede ser caro, el retry debe gastar presupuesto explícito.

Circuit breaker, cooldown y load shedding

Cuando una ruta empieza a fallar o a degradarse, el router no debería seguir probándola con todo el tráfico. Necesita memoria operativa.

MECANISMO	QUÉ HACE	SEÑAL TÍPICA
Circuit breaker	Cierra temporalmente una ruta con demasiados fallos recientes.	<code>timeout_ratio_1m > 0.25</code> .
Cooldown	Espera antes de volver a probar la ruta.	30 segundos, 2 minutos, 10 minutos.
Probe	Envía pocas runs de prueba antes de reabrir.	1% del tráfico o tarea sintética.
Load shedding	Rechaza o retrasa trabajo para proteger el sistema.	Cola superior a umbral o SLO quemándose.
Backpressure	Indica al cliente o cola que reduzca entrada.	<code>retry_after</code> , cola batch, límite por tenant.

Esto conecta con Little:

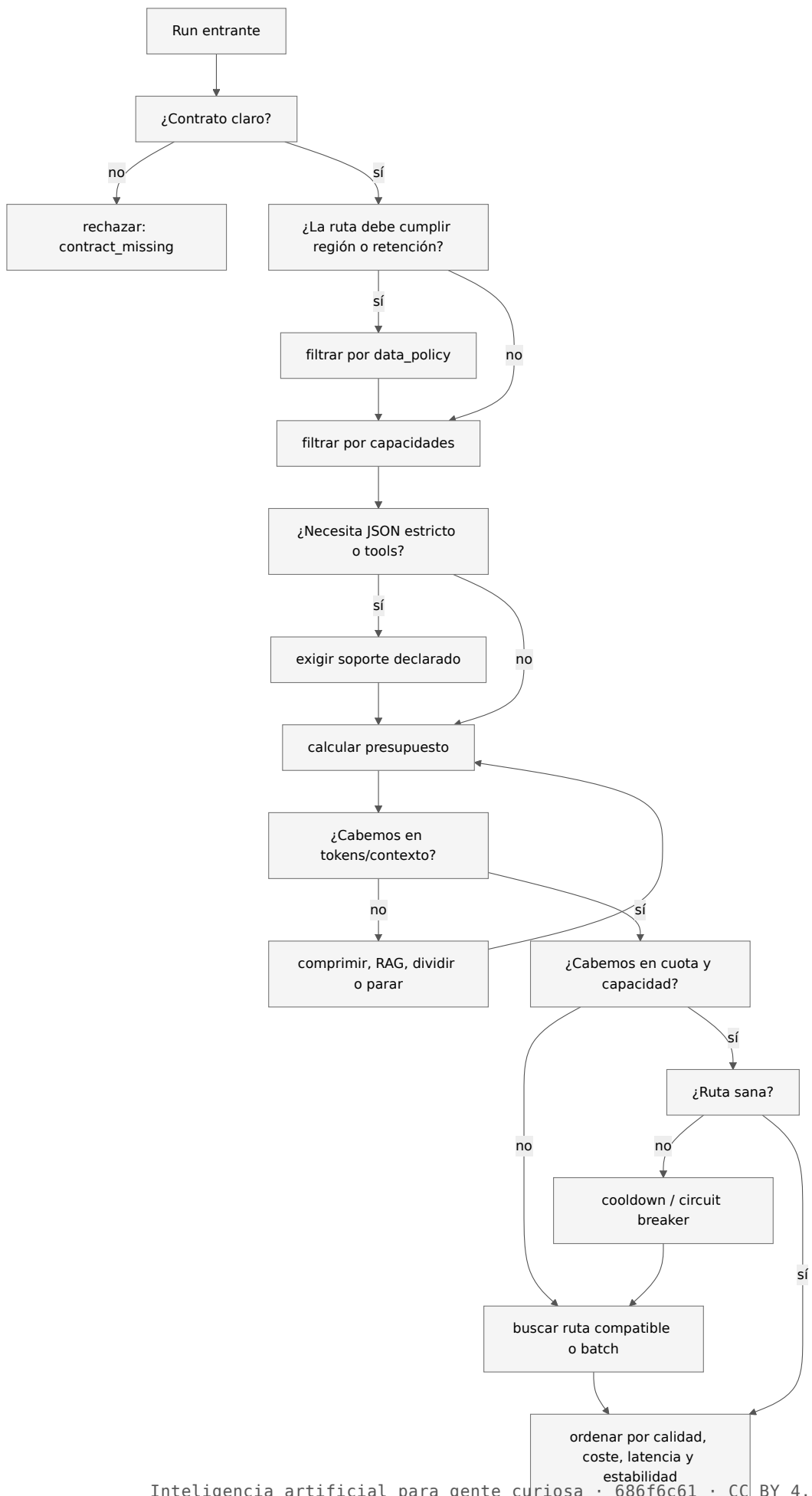
$$L = \lambda W$$

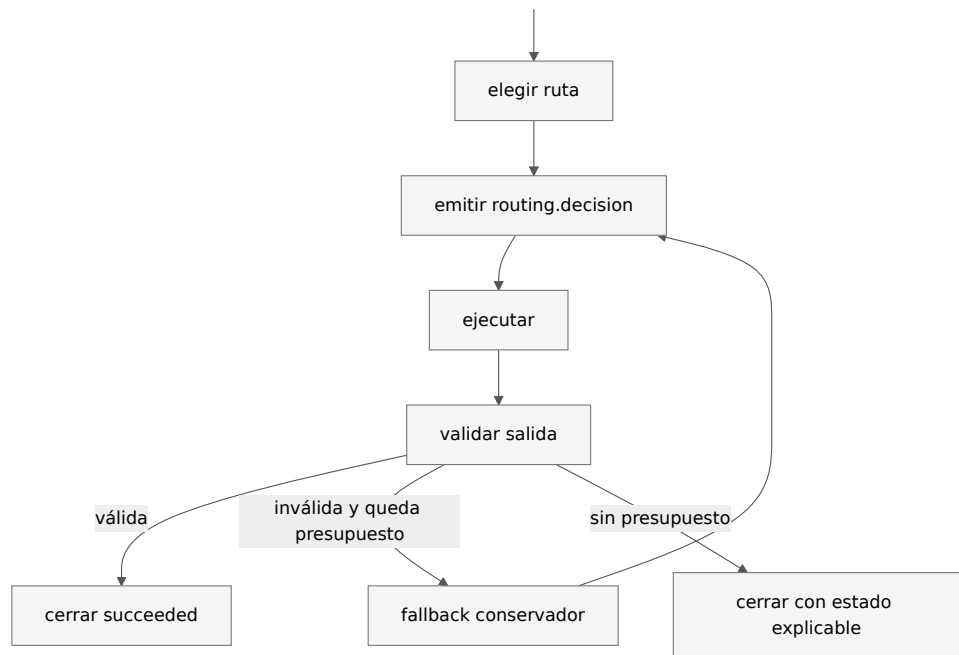
SÍMBOLO	SIGNIFICADO	EJEMPLO
L	Trabajo medio dentro del sistema.	240 runs esperando o ejecutándose.
λ	Tasa de llegada.	40 runs por minuto.
W	Tiempo medio en el sistema.	6 minutos.

Little no es una receta universal; es una alarma conceptual. Si la llegada sube y la capacidad no sube, el tiempo de espera crece. Un router que acepta todo “porque quizá sale” puede destruir el SLO. Dean y Barroso explicaron en *The Tail at Scale* que la cola larga de latencia importa porque el usuario vive los peores percentiles, no la media.¹¹ Little formuló la relación clásica entre trabajo en sistema, llegada y espera.¹²

Árbol de decisión para elegir ruta

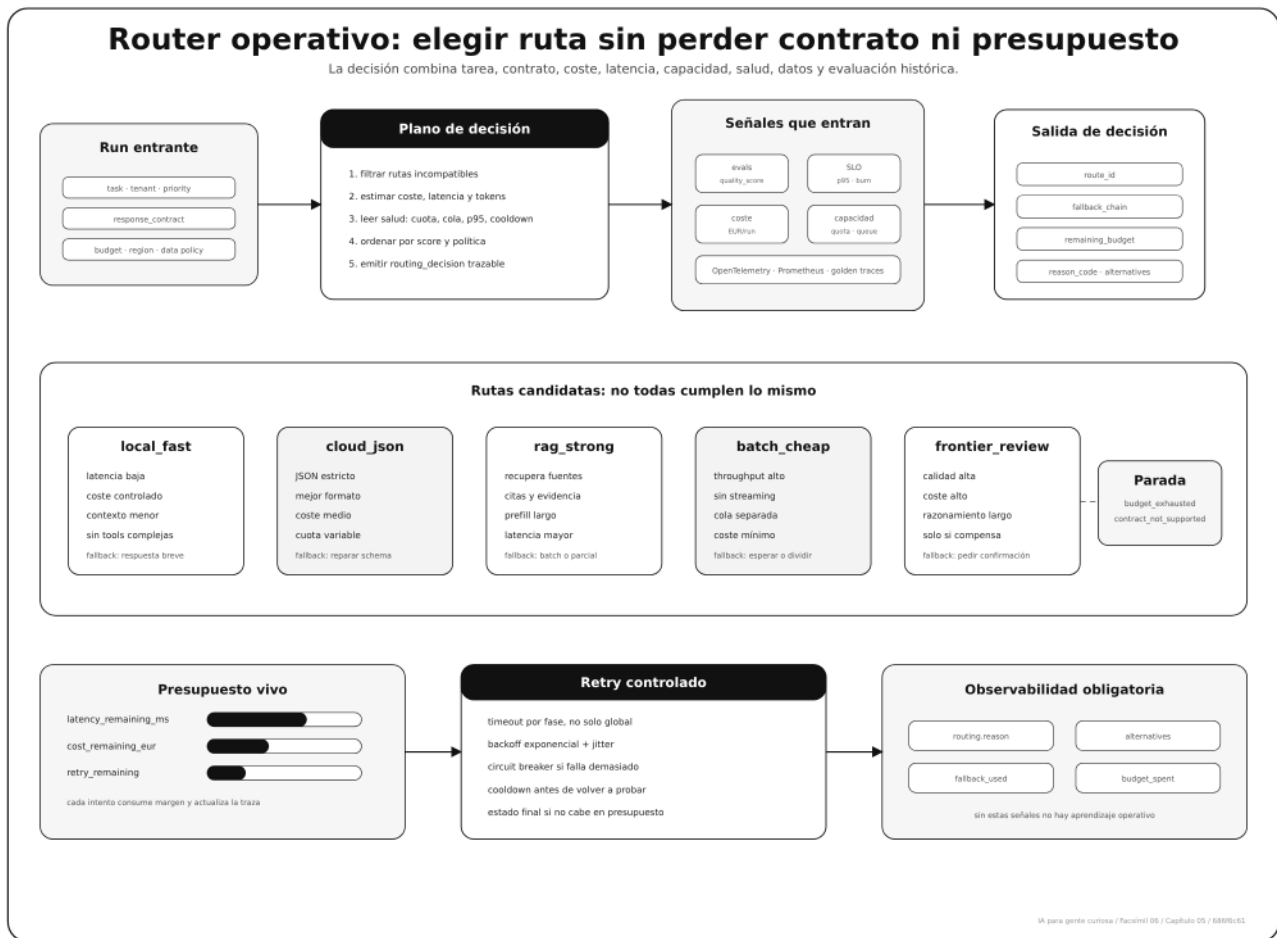
El árbol no sustituye al router, pero ayuda a revisar si la política tiene sentido. Léelo de arriba abajo:





Fíjate en dos ramas: primero contrato, después presupuesto. Si el contrato falta, no hay router inteligente; hay adivinanza. Si el presupuesto no cabe, no hay “un intento más”; hay deuda operativa.

Anatomía visual de un router de IA



El dibujo enseña la idea importante: el router no es una bifurcación simple. Es una frontera operativa. Antes de gastar tokens decide si una ruta cumple contrato, si cabe en presupuesto, si está sana y qué alternativa existe si algo no encaja.

Cómo se ve en producción

Una decisión de routing debería quedar registrada como dato estructurado. Algo así:

```

{
  "event": "routing.decision",
  "run_id": "run_20260527_0512",
  "task": "support_summary",
  "selected_route": "rag_medium_json",
  "reason": "meets_contract_under_budget",
  "budget": {
    "max_latency_ms": 6000,
    "max_cost_eur": 0.08,
    "max_retries": 1
  },
  "estimated": {
    "latency_p95_ms": 4200,
    "cost_eur": 0.041,
    "input_tokens": 7200,
    "output_tokens": 480
  },
  "alternatives": [
    {
      "route": "local_fast",
      "rejected_because": "does_not_support_required_citations"
    },
    {
      "route": "frontier_review",
      "rejected_because": "cost_above_task_budget"
    }
  ],
  "fallback_chain": ["rag_medium_json", "rag_small_brief", "batch_review"]
}

```

Esto no está para decorar logs. Sirve para responder preguntas:

PREGUNTA	CAMPO QUE LA RESPONDE
¿Por qué no usamos el modelo local?	<code>alternatives.rejected_because</code> .
¿Por qué subió el coste?	<code>estimated.cost_eur</code> , ruta seleccionada, tokens y retries.
¿Cuántas veces se activa fallback?	<code>fallback_chain</code> , <code>fallback_used</code> , <code>final_state</code> .
¿Qué ruta incumple p95?	<code>selected_route</code> + métricas de latencia.
¿Qué contrato expulsa rutas?	<code>response_contract</code> y <code>does_not_support_*</code> .

Prometheus recomienda nombres de métricas y labels que permitan entender unidades y agregación sin crear cardinalidad inútil.¹³ OpenTelemetry mantiene convenciones semánticas para sistemas generativos, incluyendo atributos de modelo, operación y uso de tokens.¹⁴ En

nuestro router, eso se traduce en no inventar métricas imposibles de mantener:
`ai_route_decision_total{task,route,reason}` , Sí; `ai_route_decision_total{run_id}` , no.

Herramientas y piezas que se suelen usar

Hay tres caminos frecuentes.

CAMINO	QUÉ USAS	PROS	CONTRAS
Router propio	Código interno, YAML, feature flags, métricas y gateway.	Control total sobre contratos, coste, datos y trazas.	Hay que mantener catálogo, salud, cuotas y fallbacks.
Router de proveedor/a gregador	OpenRouter, gateway cloud o proveedor con múltiples endpoints.	Reduce integración con muchos proveedores; aporta routing por precio, latencia o proveedor.	No sustituye tu política de producto ni tus presupuestos internos.
Router de librería	LiteLLM Router u otra capa compatible.	Acelera balanceo, retries y fallbacks entre despliegues.	Debes revisar semántica de parámetros, errores, streaming, tools y observabilidad.

Mi recomendación para ingeniería: aunque uses un router externo, conserva un **router lógico propio**. Ese router lógico decide tarea, contrato, presupuesto y política. Luego puede delegar la ejecución a OpenRouter, LiteLLM, un proveedor cloud o un runtime local. Si toda la decisión vive fuera de tu sistema, pierdes criterio de producto.

Optimizar el router: qué palancas existen

Optimizar no significa “usar siempre lo más barato”. Tampoco significa “usar siempre lo más rápido”. Optimizar significa mejorar una métrica bajo restricciones: coste por run aceptada, p95 de latencia, TTFT, tasa de contrato válido, tasa de fallback, cuota restante, calidad evaluada o experiencia percibida por el usuario.

La primera regla es elegir el objetivo:

OBJETIVO	OPTIMIZACIÓN CORRECTA	OPTIMIZACIÓN PELIGROSA
Bajar latencia interactiva	Reducir salida, usar modelo menor suficiente, streaming, caché de prefijo, ruta con p95 estable.	Recortar contexto sin medir calidad.
Bajar coste	Batch, flex, prompt caching, modelo menor, menos tokens, menos retries, cachear respuestas deterministas.	Cambiar a ruta barata que sube fallo de contrato.
Mejorar throughput	Colas por clase, batch, rutas con más TPS, bajar saturación, repartir por cuota.	Aumentar concurrencia hasta disparar p99.
Mejorar estabilidad	Circuit breaker, cooldown, margen de cuota, rutas canary, percentiles p90/p99.	Mirar solo p50 o media.
Mantener calidad	Evals por tarea, golden traces, matriz de compatibilidad, fallback conservador.	Cambiar modelo sin dataset de comparación.

OpenAI resume la optimización de latencia en principios bastante prácticos: procesar tokens más rápido, generar menos tokens, usar menos tokens de entrada, hacer menos peticiones, paralelizar, reducir la espera percibida y no usar un LLM cuando una técnica clásica basta.¹⁵ Para un router, eso se traduce en política.

PALANCA	QUÉ CAMBIA	SEÑAL QUE DEBE MEJORAR	RIESGO
Modelo menor suficiente	Ruta más rápida y barata.	TTFT, TPOT, coste.	Baja calidad si no hay evals.
Menos salida	<code>max_output_tokens</code> , contrato más compacto, respuesta breve.	Latencia de decode y coste de salida.	Respuesta incompleta.
Menos entrada	RAG más selectivo, limpiar HTML, resumir historial, deduplicar contexto.	Coste de entrada y prefill.	Perder evidencia necesaria.
Prefijo estable	Instrucciones, ejemplos y tools al principio; datos variables al final.	Cache hit rate, TTFT, coste de entrada.	Cache miss silencioso si cambian timestamps o tool order.
Menos llamadas	Unir pasos secuenciales en una respuesta estructurada.	Latencia total y coste de red.	Prompt demasiado complejo.
Paralelizar	Ejecutar pasos independientes a la vez.	Latencia de pared.	Más coste si luego cancelas trabajo.
Streaming	Primer token antes, usuario espera menos.	TTFT y experiencia percibida.	No reduce coste por sí mismo.
Batch	Llevar tareas no urgentes a procesamiento asíncrono.	Coste y cuota síncrona disponible.	No sirve para interacción inmediata.
Flex o baja prioridad	Menor coste a cambio de más espera o disponibilidad variable.	Coste por run no urgente.	Más timeouts o <code>resource_unavailable</code> .
Response cache	Reutilizar respuesta exacta si entrada y contrato son repetibles.	Latencia y coste casi cero en hits.	Solo sirve si la respuesta puede repetirse.

La latencia de una run se descompone en sus etapas, siguiendo el modelo de serving con fase de prefill y fase de decode.¹⁶

$$T_{run} \approx T_{queue} + \frac{I_{uncached}}{R_{prefill}} + \frac{O}{R_{decode}} + T_{tools} + T_{network}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
T_{run}	Latencia total aproximada.	4200 ms.
T_{queue}	Espera en cola o proveedor.	300 ms.
$I_{uncached}$	Tokens de entrada que realmente se procesan.	1200 dinámicos + 0 cacheados.
$R_{prefill}$	Velocidad de procesamiento de entrada.	tokens/s en prefill.
O	Tokens de salida generados.	480 tokens.
R_{decode}	Velocidad de generación.	tokens/s en decode.
T_{tools}	Tiempo de tools, RAG, validación o reranking.	650 ms.
$T_{network}$	Red, gateway y serialización.	120 ms.

La fórmula explica por qué reducir salida suele notarse más que recortar un poco de entrada en chats normales: el decode produce tokens de uno en uno. Pero en documentos largos, RAG grande o tools repetidas, el prefill, el caché y la recuperación sí pueden dominar.

Para coste con caché:

$$C_{in} = I_{fresh}P_{in} + I_{cache\_write}P_{write} + I_{cache\_read}P_{read}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
I_{fresh}	Tokens nuevos no cacheados.	pregunta del usuario y chunks dinámicos.
P_{in}	Precio normal de entrada.	precio por 1M tokens.
I_{cache_write}	Tokens que se escriben en caché.	system prompt largo o tools.
P_{write}	Precio de escritura de caché.	puede ser mayor que entrada normal.
I_{cache_read}	Tokens leídos desde caché.	prefijo estable reutilizado.
P_{read}	Precio de lectura de caché.	normalmente menor que entrada normal.

La conclusión: cachear compensa cuando hay prefijos largos y repetidos. No compensa si cada prompt empieza distinto.

Cuánta fiabilidad te da una cadena de fallback. Tener un modelo de respaldo no es una vaga sensación de seguridad: añade fiabilidad de forma calculable. Si la ruta y sus respaldos fallan de manera independiente, la cadena solo falla si fallan todos, y eso es un producto de probabilidades.¹⁷

$$R_{\text{cadena}} = 1 - \prod_{i=1}^n (1 - r_i)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
r_i	Fiabilidad de cada ruta (probabilidad de que responda bien).	0,95 y 0,90.
n	Número de rutas en la cadena de fallback.	2
R_{cadena}	Fiabilidad del conjunto con respaldo.	$1 - 0,05 \times 0,10 = 0,995$.

En palabras: un modelo al 95% con un respaldo al 90% da un 99,5% combinado, siempre que los fallos sean independientes. El matiz es ese «independientes»: si ambos dependen del mismo proveedor o de la misma red, los fallos se correlacionan y la cuenta optimista no se cumple. Un buen fallback usa una vía de fallo distinta, no otra copia del mismo riesgo.

Cómo explorar rutas sin malgastar: UCB1. Si no sabes qué ruta es mejor, tienes un dilema clásico: explotar la que parece buena o explorar las demás por si lo son más. El algoritmo UCB1 lo resuelve con una fórmula que premia tanto el buen rendimiento medio como la incertidumbre por haber probado poco.¹⁸

$$UCB_i = \bar{x}_i + \frac{2 \ln n}{n_i}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\bar{x}_i$	Recompensa media observada de la ruta i (calidad por coste).	0,82
n_i	Veces que se ha usado la ruta i .	40
n	Veces totales de routing.	1.000
UCB_i	Puntuación que decide la ruta del próximo caso.	Suma media más bono de exploración.

En palabras: se elige la ruta con mayor puntuación, donde el segundo término es un «bono» que crece si una ruta se ha probado poco. Así el sistema prueba lo justo las opciones dudosas y converge a la buena. Es el reemplazo honesto del «creo que el modelo caro es mejor»: que lo demuestren los datos, sin dejar de explorar.

Cuánto cuestan de verdad los reintentos. Un reintento parece gratis hasta que se cuenta. Si cada intento falla con probabilidad q de forma independiente, el número de intentos hasta acertar sigue una distribución geométrica, y su media es conocida.¹⁹

$$\mathbb{E}[N] = \frac{1}{1-q}, \quad \mathbb{E}[C] = \frac{c}{1-q}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
q	Probabilidad de que un intento falle.	0,2
$\mathbb{E}[N]$	Número medio de intentos hasta el éxito.	$1/0,8 = 1,25$.
c	Coste de un intento.	0,03 euros.
$\mathbb{E}[C]$	Coste medio por tarea resuelta, con reintentos.	0,0375 euros.

En palabras: con un 20% de fallos, resolver una tarea cuesta de media un 25% más de lo que sugiere el precio por llamada. Por eso el coste por tarea aceptada y el coste por llamada divergen, y por eso un límite de reintentos no es tacañería: sin tope, una racha de fallos multiplica la factura sin garantizar el éxito.

Cómo respetar un presupuesto sin frenar en seco: el cubo de tokens. Un límite de peticiones por minuto no debería ser un muro que rechaza todo al pasarse. El algoritmo del cubo de tokens permite ráfagas acotadas y un ritmo sostenido: el cubo se rellena a tasa constante y cada petición gasta tokens; si no hay, espera o se rechaza.²⁰

$$\text{tokens}(t) = \min(b, \text{tokens}(t^-) + r \cdot \Delta t)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
r	Tasa de relleno (ritmo sostenido permitido).	100 por minuto.
b	Tamaño del cubo (ráfaga máxima tolerada).	150
Δt	Tiempo transcurrido desde la última petición.	0,5 segundos.
$\text{tokens}(t)$	Crédito disponible ahora mismo.	Se gasta una unidad por petición.

En palabras: el cubo absorbe picos breves (hasta b) pero impone un ritmo medio (r) a largo plazo. Es la forma estándar de implementar un presupuesto por minuto que no castiga una ráfaga legítima ni deja que el sistema se desboque, y traduce directamente los límites TPM/RPM del proveedor a una política propia controlable.

Qué indican los proveedores y servicios

Los proveedores no dicen todos lo mismo, pero sus recomendaciones convergen en algo: **medir tokens, prefijos, latencia, cuota y modalidad de trabajo.**

PROVEEDOR O SERVICIO	QUÉ OFRECE O RECOMIENDA	USO EN ROUTING Y MÉTRICA CLAVE
OpenAI	Latency optimization, prompt caching automático, Batch API, Flex Processing y modelos pequeños para alto volumen.	Interactivo con streaming y caché; batch/flex para evals o enriquecimiento. Mira TTFT, tokens cacheados, output tokens, coste por aceptada y finalización batch.
Anthropic	Prompt caching automático o explícito, TTL de 5 minutos o 1 hora, cache diagnostics para misses.	Separar prefijo estable, tools e historial. Mira <code>cache_read_input_tokens</code> , <code>cache_creation_input_tokens</code> , TTFT y divergencia de prefijo.
OpenRouter	Routing por precio, throughput o latencia; percentiles recientes; <code>max_price</code> ; filtros por parámetros; sticky routing y response caching.	Elegir proveedor por p90/p99, precio máximo o soporte de JSON/tools. Mira p50/p90/p99, throughput, proveedor elegido, cache hit y fallbacks.
LiteLLM	Load balancing entre despliegues, cooldowns, fallbacks, timeouts, retries, estrategias por peso, rate-limit aware, latencia o coste.	Gateway interno para unificar proveedores manteniendo política propia encima. Mira RPM/TPM por despliegue, cooldowns, retries y latencia añadida.
Runtime propio	vLLM, SGLang, TGI u otro serving controlado por el equipo.	Ruta local para datos sensibles, coste predecible o baja latencia si tienes capacidad. Mira TTFT, TPOT, memoria GPU, cola, KV cache y goodput.

OpenAI documenta que prompt caching funciona automáticamente en modelos recientes, que los hits dependen de prefijos exactos y que colocar contenido estático al principio ayuda a reducir latencia y coste.²¹ También documenta Batch API para trabajos asíncronos con menor coste, límites separados y ventana de finalización de 24 horas; eso lo convierte en una ruta distinta, no en una optimización invisible.²² Flex processing baja coste a cambio de más lentitud y posible falta temporal de recursos, así que debe vivir en rutas no urgentes y con timeouts acordes.²³

Anthropic documenta prompt caching con caché automático o breakpoints explícitos, lectura de caché a menor coste y buenas prácticas como cachear contenido estable al principio del prompt.²⁴ También ofrece cache diagnostics para diagnosticar misses por divergencias de modelo, system prompt, tools o historial.²⁵ Esto es oro para ingeniería: si la métrica de caché cae, no quieres discutir; quieres saber qué byte cambió.

OpenRouter documenta routing por precio, throughput o latencia, preferencias por percentiles y filtros como `max_price` o soporte de parámetros.²⁶ También documenta prompt caching con sticky routing para mantener caliente el mismo proveedor cuando la caché aporta ahorro.²⁷ En su guía de latencia describe el overhead del gateway, cache warming, checks de saldo y fallback como factores operativos.²⁸

LiteLLM documenta load balancing, cooldowns, fallbacks, timeouts, retries y estrategias como weighted pick, rate-limit aware, latency-based, least-busy y lowest-cost routing; además advierte que estrategias con tracking de uso pueden añadir latencia por Redis.²⁹ Esa advertencia es importante: una optimización puede introducir su propio coste.

Orden práctico para optimizar sin romper calidad

Si yo tuviera que optimizar un router de producción, no empezaría tocando pesos a ciegas. Seguiría este orden:

PASO	QUÉ HACES	POR QUÉ
1. Baseline	Mides p50/p95/p99, TTFT, TPOT, tokens, coste, contrato y aceptación por ruta.	Sin línea base no sabes si mejoras.
2. Separar clases	Distingues interactivo, batch, largo, RAG, tools y revisión.	Cada clase optimiza otra cosa.
3. Reducir salida	Ajustas contrato, longitud, schema y <code>max_output_tokens</code> .	Suele tocar latencia y coste de forma directa.
4. Estabilizar prefijos	Mueves instrucciones, ejemplos y tools estables al principio.	Mejora cache hit y reduce prefill repetido.
5. Podar entrada	Limpias HTML, deduplicas chunks, mejoras RAG y compaction.	Evita pagar contexto que no aporta.
6. Elegir modelo suficiente	Cambias a modelo menor solo si pasa evals.	Latencia y coste bajan sin convertirlo en lotería.
7. Usar rutas asíncronas	Batch o flex para lo que no exige respuesta inmediata.	Libera cuota interactiva y baja coste.
8. Optimizar proveedor	Ordenas por p90/p99, throughput, precio máximo y soporte de contrato.	Aprovechas señales vivas del mercado.
9. Probar política	Shadow routing, golden traces, canary y rollback.	Evita que una mejora media rompa casos críticos.

El criterio final no es “ha bajado la factura”. Es:

$$\Delta C_{util} < 0 \quad \wedge \quad \Delta L_{p95} \leq 0 \quad \wedge \quad \Delta Q \geq -\epsilon \quad \wedge \quad \Delta F_{contract} \leq \tau$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
ΔC_{util}	Cambio en coste por run aceptada.	Queremos que baje.
ΔL_{p95}	Cambio de latencia p95.	No debería empeorar en rutas interactivas.
ΔQ	Cambio de calidad evaluada.	Permitimos caída mínima ϵ si el producto lo acepta.
$\Delta F_{contract}$	Cambio en fallos de contrato.	No puede superar el umbral τ .
ϵ, τ	Tolerancias explícitas.	<code>quality_drop <= 0.01</code> , <code>contract_fail_delta <= 0.002</code> .

Esa fórmula evita una trampa muy común: declarar victoria porque el coste baja mientras la calidad cae o el contrato falla más.

Patrones que ayudan a no romper el sistema

PATRÓN	DECISIÓN CONCRETA
Cola por clase de tarea	Separar interactivo, batch, largo, RAG y revisión.
Presupuesto decreciente	Cada fase resta tiempo, coste y retries disponibles.
Fallback hacia menos alcance	Si no cabe la respuesta completa, entregar resumen breve o pasar a batch.
Circuit breaker por ruta	Si una ruta degradada, dejar de enviarle casi todo el tráfico.
Canary de rutas	Probar una ruta nueva con poco tráfico y comparar contra golden traces.
Shadow routing	Calcular qué habría elegido una política nueva sin ejecutar la ruta.
Holdout de evaluación	No optimizar el router solo con los mismos casos que usas para ajustarlo.
Idempotencia	Si reintentas, que no dupliques efectos externos.

El último punto importa mucho. Reintentar una lectura es una cosa. Reintentar una acción con efecto en sistemas externos es otra. Si una tool escribe, cobra, publica, borra o modifica estado, el retry debe pasar por idempotency keys, confirmación o estado explícito.

Cómo probar una política de routing

Una política de routing también se testea. No basta con mirar una demo.

PRUEBA	QUÉ COMPRUEBA	EJEMPLO
Unit test de restricciones	Una ruta incompatible nunca se elige.	Si pide citas, <code>local_fast</code> queda descartada.
Test de presupuesto	Coste, tokens y latencia bloquean rutas caras.	<code>frontier_review</code> no entra en <code>fast_triage</code> .
Test de salud	Cooldown y timeout ratio cambian decisión.	Si <code>rag_medium_json</code> degrada, pasa a batch o ruta breve.
Golden traces	La decisión esperada se mantiene en casos clave.	<code>support_summary</code> elige <code>rag_medium_json</code> con razón conocida.
Shadow routing	Política nueva calcula decisión sin ejecutar.	Comparas <code>policy@1.7</code> contra <code>policy@1.8</code> .
Canary	Política nueva recibe poco tráfico real.	5% de <code>support_summary</code> durante una hora.
Test de coste útil	No mejora solo precio por llamada.	Baja coste por token, pero sube fallo de contrato: no se acepta.

Los tests deberían producir un informe. Algo así:

```
policy: router@1.8.0
cases: 120
changed_decisions: 14
expected_changes: 12
unexpected_changes: 2
cost_per_accepted_delta: -8.4%
contract_failure_delta: +0.3%
latency_p95_delta_ms: -410
decision: canary, not full rollout
```

Ese informe se parece más a ingeniería de software que a “probemos el prompt nuevo”. Y esa es la idea: el router es código de producción.

En el día a día

El routing aparece la primera vez que tienes más de un modelo, o más de una ruta, y alguien pregunta cuál usar. La respuesta operable no es «el mejor», sino «el mejor para esta tarea y este presupuesto». Una clasificación trivial puede ir a un modelo barato y rápido; una consulta delicada, al modelo fuerte; y si el modelo fuerte se satura o devuelve un error, el

fallback decide si el sistema se degrada con elegancia (responde con el barato, o se abstiene) o se cae entero. Diseñar esa decisión por adelantado es la diferencia entre un mal día y una incidencia.

El presupuesto por tarea aparece cuando el coste se dispara y nadie sabe por qué. En un prototipo, da igual; en producción, una tarea que se permite reintentar sin límite, ampliar contexto sin tope o saltar siempre al modelo caro puede multiplicar la factura sin que la calidad mejore. Poner un presupuesto por tarea (tokens, llamadas, coste, latencia) convierte «se nos fue de las manos» en «esta tarea se detiene cuando llega a su límite», que es una frase que se puede defender ante quien paga.

Por qué debería importarte

Porque sin routing ni presupuesto un sistema de IA tiene dos finales malos, y los dos llegan pronto. O usa el modelo caro para todo y se vuelve insostenible en cuanto crece el tráfico, o usa el barato para todo y falla justo en las tareas que importan. El routing con fallback y presupuesto es lo que te permite la única respuesta sensata: gastar donde aporta y ahorrar donde no, con una red de seguridad cuando algo se rompe.

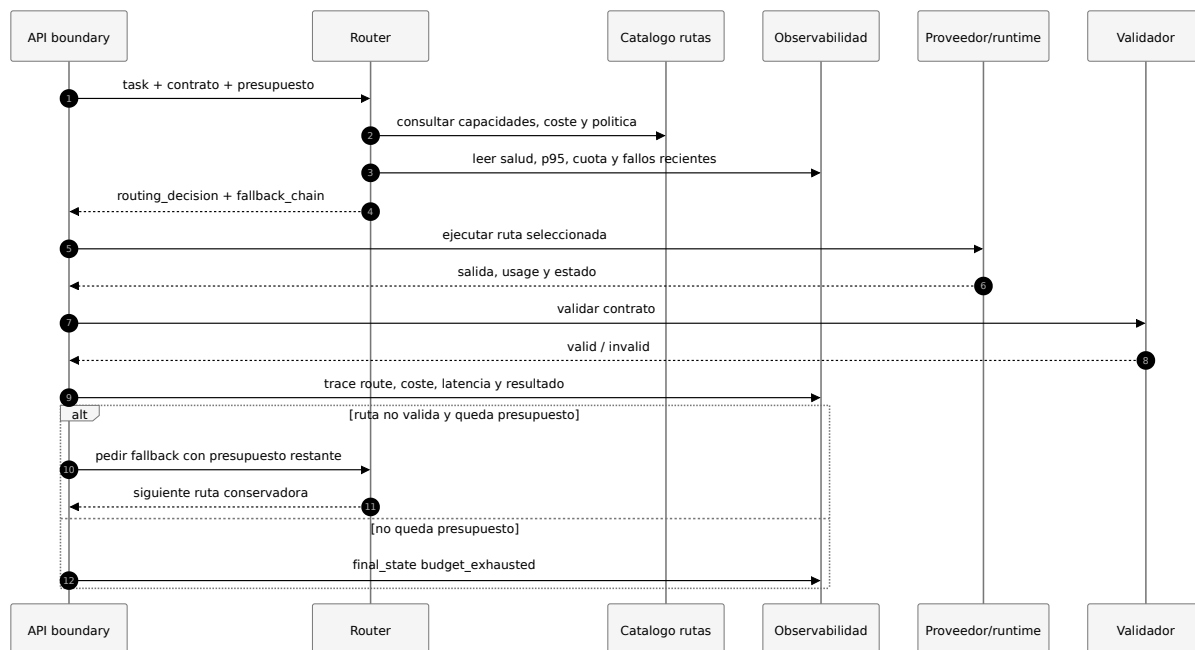
Y hay una consecuencia de oficio. Quien diseña el routing y el fallback puede prometer dos cosas que un sistema sin ellos no puede: un coste acotado y un comportamiento predecible cuando un proveedor falla. Esas dos promesas son las que distinguen un sistema que aguanta un pico de tráfico o una caída de un proveedor de uno que se desmorona con el primer imprevisto.

Dónde solía tropezar yo

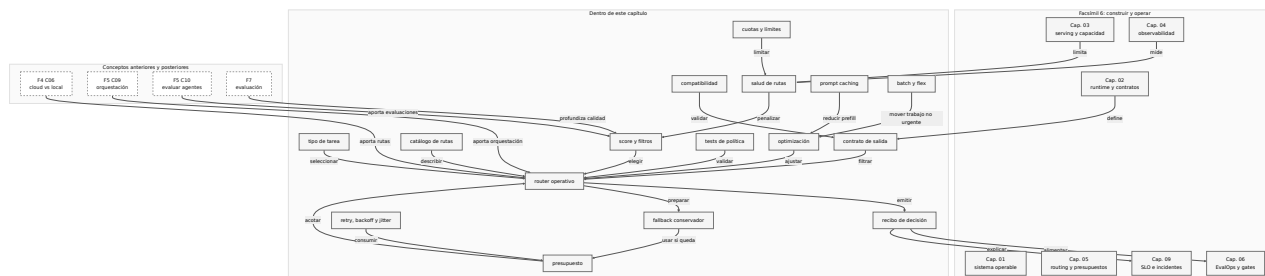
TROIEZO	POR QUÉ ES UN PROBLEMA	ANTÍDOTO
Fallback como improvisación	Cambia comportamiento sin contrato y confunde al usuario.	Definir cadena de fallback por tarea y estado final si no cabe.
Reintentar todo	Aumenta coste y presión justo cuando una ruta está mal.	Retry solo para errores recuperables, con backoff, jitter y presupuesto.
Optimizar precio por llamada	Puede subir el coste útil si fallan más respuestas.	Medir coste por run aceptada y fallo de contrato por ruta.
No registrar alternativas descartadas	Luego nadie sabe por qué el router eligió una ruta.	Emitir <code>routing.decision</code> con <code>reason</code> y <code>alternatives</code> .
Tratar todas las tareas igual	Una tarea batch y una interactiva no tienen el mismo SLO.	Presupuesto por tarea y colas separadas.
Delegar toda la política en un agregador	Pierdes criterio de producto y trazabilidad interna.	Mantener router lógico propio aunque uses herramientas externas.
No probar la política	Cambia producción sin saber qué casos se mueven.	Unit tests, golden traces, shadow routing y canary.
Optimizar una métrica aislada	Baja coste o latencia, pero sube fallo de contrato o baja calidad.	Optimizar con coste útil, p95, calidad y contrato a la vez.
Romper el prefijo cacheable	Un timestamp o cambio de orden en tools puede destruir el cache hit.	Mantener prefijos estables y medir <code>cache_read_input_tokens</code> .

Cómo encaja todo

Primero, el ciclo operativo de una run con routing:



Y el mapa conceptual del capítulo:



Routing es donde se juntan tres mundos: producto, infraestructura y evaluación. Producto define qué necesita la tarea. Infraestructura dice qué capacidad existe. Evaluación dice qué ruta funciona de verdad.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Router	Componente que selecciona una ruta de ejecución.
Ruta	Combinación de modelo, proveedor, runtime, contrato, cola y política.
Catálogo de rutas	Registro versionado de rutas, capacidades, costes, límites, owner y fallback.
Fallback	Ruta alternativa usada cuando la principal no cumple o no conviene.
Presupuesto de tarea	Límite de coste, latencia, tokens, retries y acciones para una tarea concreta.
Cuota	Capacidad asignada por proveedor o runtime, medida en tokens, peticiones o concurrencia.
Rate limit	Límite operativo de uso por unidad de tiempo.
Matriz de compatibilidad	Tabla que compara qué rutas soportan contrato, tools, streaming, región y parámetros.
Retry	Nuevo intento controlado de una operación.
Backoff	Espera creciente entre retries.
Jitter	Aleatoriedad añadida al backoff para evitar reintentos sincronizados.
Circuit breaker	Mecanismo que deja de enviar tráfico a una ruta temporalmente degradada.
Cooldown	Periodo de espera antes de volver a probar una ruta.
Load shedding	Rechazar o retrasar trabajo para proteger el servicio.
Backpressure	Señal para que clientes o colas reduzcan entrada.
Coste útil	Coste por run aceptada, no solo coste por llamada.
Prompt caching	Reutilización de un prefijo estable del prompt para reducir latencia y coste de entrada.
Cache hit	Petición que encuentra en caché el prefijo esperado.
Batch	Procesamiento asíncrono de muchas peticiones que no necesitan respuesta inmediata.

TÉRMINO	DEFINICIÓN
Flex processing	Ruta de menor coste y menor prioridad, aceptando más espera o disponibilidad variable.
Sticky routing	Mantener una conversación o prefijo en el mismo proveedor para aumentar hits de caché.
Shadow routing	Evaluar qué ruta elegiría una política nueva sin ejecutarla.
Canary	Desplegar una ruta o política a una fracción pequeña del tráfico.
Golden trace	Traza representativa que sirve como referencia para comparar decisiones de routing.

Antes de pasar página

- ☐ ¿Puedes explicar por qué routing no es solo fallback?
- ☐ ¿Puedes definir un presupuesto de tarea con latencia, coste, tokens y retries?
- ☐ ¿Puedes decidir cuándo un retry tiene sentido y cuándo no?
- ☐ ¿Puedes explicar backoff y jitter sin fórmulas opacas?
- ☐ ¿Puedes distinguir coste por llamada de coste por run aceptada?
- ☐ ¿Puedes diseñar una cadena de fallback conservadora para una tarea concreta?
- ☐ ¿Puedes decir qué campos debería tener un evento `routing.decision` ?
- ☐ ¿Puedes explicar por qué `run_id` no debería ser label de una métrica de routing?
- ☐ ¿Puedes conectar router con observabilidad, SLO y EvalOps?
- ☐ ¿Puedes justificar cuándo usar router propio, OpenRouter, LiteLLM o una mezcla?
- ☐ ¿Puedes diseñar un catálogo de rutas con capacidades, límites, owner y fallback?
- ☐ ¿Puedes calcular si una tarea cabe en una cuota TPM/RPM con margen?
- ☐ ¿Puedes explicar por qué una API compatible no garantiza comportamiento compatible?
- ☐ ¿Puedes proponer tests para una política nueva antes de desplegarla?
- ☐ ¿Puedes ordenar palancas de optimización sin tocar primero el modelo?
- ☐ ¿Puedes explicar cuándo usar prompt caching, batch, flex, streaming o response cache?
- ☐ ¿Puedes definir una condición de éxito que incluya coste, p95, calidad y contrato?

En resumen

IDEA	QUÉ DEBES LLEVARTE
Routing es una decisión operativa.	Combina tarea, contrato, presupuesto, salud, evaluación y trazabilidad.
El catálogo de rutas es el suelo del router.	Sin capacidades, owner, límites, región y fallback versionados, la decisión se vuelve opaca.
Fallback debe conservar contrato.	Una ruta alternativa no puede inventar permisos, formato o alcance.
El presupuesto vive durante la run.	Cada retry, tool, token y espera consume margen.
Las cuotas externas también son arquitectura.	RPM, TPM, concurrencia y región deben entrar antes de ejecutar.
Optimizar requiere elegir métrica objetivo.	Prompt caching, batch, flex, streaming, menor salida o modelo suficiente sirven para problemas distintos.
Reintentar mal empeora el sistema.	Timeout, backoff, jitter y circuit breaker protegen a usuarios y dependencias.
El coste útil manda más que el precio por token.	Una ruta barata puede salir cara si falla contrato o requiere revisión.
La decisión debe quedar explicada.	<code>routing.decision</code> permite aprender, depurar y mejorar la política.
Una política de routing se prueba como software.	Unit tests, shadow routing, canary y golden traces evitan cambios a ciegas.

Para saber más

- Amazon Web Services. (2026). *Timeouts, retries, and backoff with jitter*. AWS Builders Library. <https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/>
- Anthropic. (2026). *Cache diagnostics*. <https://platform.claude.com/docs/en/build-with-claude/cache-diagnostics>
- Anthropic. (2026). *Prompt caching*. <https://platform.claude.com/docs/en/build-with-claude/prompt-caching>
- Anthropic. (2026). *Rate limits*. <https://platform.claude.com/docs/en/api/rate-limits>
- Beyer, B., Jones, C., Petoff, J. y Murphy, N. R. (eds.). (2016). *Handling Overload*. En *Site Reliability Engineering*. <https://sre.google/sre-book/handling-overload/>

- Dean, J. y Barroso, L. A. (2013). *The Tail at Scale*. Communications of the ACM, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>
- Ewaschuk, R. (2016). *Monitoring Distributed Systems*. En B. Beyer, C. Jones, J. Petoff y N. R. Murphy (eds.), *Site Reliability Engineering*. <https://sre.google/sre-book/monitoring-distributed-systems/>
- LiteLLM. (2026). *Router - Load Balancing*. <https://docs.litellm.ai/docs/routing>
- Little, J. D. C. (1961). *A Proof for the Queuing Formula: $L = \lambda W$* . Operations Research, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>
- OpenAI. (2026). *Batch API*. <https://developers.openai.com/api/docs/guides/batch>
- OpenAI. (2026). *Flex processing*. <https://developers.openai.com/api/docs/guides/flex-processing>
- OpenAI. (2026). *Latency optimization*. <https://developers.openai.com/api/docs/guides/latency-optimization>
- OpenAI. (2026). *Prompt caching*. <https://developers.openai.com/api/docs/guides/prompt-caching>
- OpenAI. (2026). *Rate limits*. <https://developers.openai.com/api/docs/guides/rate-limits>
- OpenRouter. (2026). *Latency and Performance*. <https://openrouter.ai/docs/guides/best-practices/latency-and-performance>
- OpenRouter. (2026). *Prompt Caching*. <https://openrouter.ai/docs/features/prompt-caching>
- OpenRouter. (2026). *Provider routing*. <https://openrouter.ai/docs/guides/routing/provider-selection>
- OpenTelemetry. (2026). *Semantic Conventions for Generative AI Systems*. <https://opentelemetry.io/docs/specs/semconv/gen-ai/>
- Prometheus. (2026). *Metric and label naming*. <https://prometheus.io/docs/practices/naming/>

Notas

1. Beyer, B., Jones, C., Petoff, J. y Murphy, N. R. (eds.). (2016). *Handling Overload*. En *Site Reliability Engineering*. <https://sre.google/sre-book/handling-overload/>. Consultado el 27 de mayo de 2026. ↵
2. Amazon Web Services. (2026). *Timeouts, retries, and backoff with jitter*. AWS Builders Library. <https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/>. Consultado el 27 de mayo de 2026. ↵
3. OpenRouter. (2026). *Provider routing*. <https://openrouter.ai/docs/guides/routing/provider-selection>. Consultado el 27 de mayo de 2026. ↵

4. LiteLLM. (2026). *Router - Load Balancing*. <https://docs.litellm.ai/docs/routing>. Consultado el 27 de mayo de 2026. ↵
5. OpenAI. (2026). *Rate limits*. <https://developers.openai.com/api/docs/guides/rate-limits>. Consultado el 27 de mayo de 2026. ↵
6. Anthropic. (2026). *Rate limits*. <https://platform.claude.com/docs/en/api/rate-limits>. Consultado el 27 de mayo de 2026. ↵
7. Keeney, R. L. y Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley. Formaliza la función de valor aditiva ponderada para decidir entre alternativas. ↵
8. OpenAI, 2026, *Rate limits*. ↵
9. Anthropic, 2026, *Rate limits*. ↵
10. OpenRouter, 2026, *Provider routing*. ↵
11. Dean, J. y Barroso, L. A. (2013). The Tail at Scale. *Communications of the ACM*, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>. Consultado el 27 de mayo de 2026. ↵
12. Little, J. D. C. (1961). A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research*, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>. ↵
13. Prometheus. (2026). *Metric and label naming*. <https://prometheus.io/docs/practices/naming/>. Consultado el 27 de mayo de 2026. ↵
14. OpenTelemetry. (2026). *Semantic Conventions for Generative AI Systems*. <https://opentelemetry.io/docs/specs/semconv/gen-ai/>. Consultado el 27 de mayo de 2026. ↵
15. OpenAI. (2026). *Latency optimization*. <https://developers.openai.com/api/docs/guides/latency-optimization>. Consultado el 28 de mayo de 2026. ↵
16. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H. y Stoica, I. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. *Proceedings of SOSP 2023*. <https://doi.org/10.1145/3600006.3613165>. Distingue las fases de prefill y decode que dominan la latencia de inferencia. ↵
17. Trivedi, K. S. (2016). *Probability and Statistics with Reliability, Queuing, and Computer Science Applications* (2.ª ed.). Wiley. Para componentes redundantes en paralelo, la fiabilidad del conjunto es el complemento del producto de las probabilidades de fallo individuales. ↵
18. Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3), 235-256. <https://doi.org/10.1023/A:1013689704352>. Introduce UCB1 con garantías de arrepentimiento logarítmico. ↵

9. El número de ensayos de Bernoulli independientes hasta el primer éxito sigue una distribución geométrica de media $1/p$; véase Ross, S. M. (2014). *Introduction to Probability Models* (11.ª ed.). Academic Press. ↵
10. Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer Networks* (5.ª ed.). Pearson. Describe el algoritmo del cubo de tokens (token bucket) para conformar tráfico permitiendo ráfagas hasta el tamaño del cubo. ↵
11. OpenAI. (2026). *Prompt caching*. <https://developers.openai.com/api/docs/guides/prompt-caching>. Consultado el 28 de mayo de 2026. ↵
12. OpenAI. (2026). *Batch API*. <https://developers.openai.com/api/docs/guides/batch>. Consultado el 28 de mayo de 2026. ↵
13. OpenAI. (2026). *Flex processing*. <https://developers.openai.com/api/docs/guides/flex-processing>. Consultado el 28 de mayo de 2026. ↵
14. Anthropic. (2026). *Prompt caching*. <https://platform.claude.com/docs/en/build-with-claude/prompt-caching>. Consultado el 28 de mayo de 2026. ↵
15. Anthropic. (2026). *Cache diagnostics*. <https://platform.claude.com/docs/en/build-with-claude/cache-diagnostics>. Consultado el 28 de mayo de 2026. ↵
16. OpenRouter, 2026, *Provider routing*. ↵
17. OpenRouter. (2026). *Prompt Caching*. <https://openrouter.ai/docs/features/prompt-caching>. Consultado el 28 de mayo de 2026. ↵
18. OpenRouter. (2026). *Latency and Performance*. <https://openrouter.ai/docs/guides/best-practices/latency-and-performance>. Consultado el 28 de mayo de 2026. ↵
19. LiteLLM, 2026, *Router - Load Balancing*. ↵

Cuadernos para practicar

Escanea el código con el móvil para abrir el cuaderno en Google Colab y ejecutarlo sin instalar nada.



Routing, fallback y presupuesto: el modelo adecuado para cada tarea

<https://colab.research.google.com/github/686f6c61/iaparagentecuriosa-notebooks/blob/main/Facs%C3%ADmil%2006/02-routing-fallback-presupuesto.ipynb>

Inteligencia artificial para gente curiosa · 686f6c61 · CC BY 4.0

Capítulo 06: EvalOps y gates de release

Qué deberías poder hacer al terminar

En el capítulo 04 aprendimos a observar una run. En el capítulo 05 construimos un router que decide ruta, presupuesto y fallback. Ahora falta una pieza incómoda: **cómo impedimos que un cambio llegue a producción solo porque en una demo parecía funcionar.**

Un sistema de IA cambia por muchos sitios: prompt, modelo, proveedor, parámetros, RAG, router, tool, contrato de salida, dataset, evaluador, runtime, política de coste o límite de latencia. Cada cambio puede mejorar una métrica y romper otra. EvalOps es la forma disciplinada de convertir esa incertidumbre en un proceso de ingeniería.

Al terminar, deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Diseñar un pipeline EvalOps.	Separas dataset, runner, evaluadores, scorecard, gate, canary y feedback.
Elegir qué evaluar según el cambio.	No evalúas igual un prompt, un router, un RAG, una tool o un runtime.
Versionar evidencia.	Guardas dataset_id , prompt_version , model_id , route_catalog_version , eval_policy_version y trace_schema_version .
Comparar baseline contra candidate.	Miras calidad, coste, latencia, contrato, trazas y regresiones.
Definir gates discutibles.	Los umbrales están escritos, tienen dueño, tolerancia y motivo.
Evitar autoengaño estadístico.	Repites casos inestables, separas holdout y usas intervalos cuando la muestra es pequeña.
Conectar evals con producción.	Las trazas que fallan vuelven al dataset y alimentan nuevas pruebas.

La idea central: **una release de IA no debería pasar por intuición; debería pasar por evidencia versionada.**

El cambio pequeño que no era pequeño

Imagina una pull request que dice: “mejoro el prompt para que el asistente sea más claro”. Parece inocente. Se revisa el texto, se prueba con tres preguntas y todo parece más amable.

Pero al desplegarlo aparecen efectos laterales: algunas respuestas son más largas, sube el coste medio, una salida JSON empieza a traer un campo extra, la latencia p95 empeora porque el prompt metió más contexto, y el router deriva más tareas al modelo caro porque el clasificador entiende peor la intención.

La conclusión no es “no cambies prompts”. La conclusión es más profesional: un cambio de IA tiene superficie operativa. Si no lo medimos antes, lo descubrimos después con usuarios, coste y soporte.

EvalOps pone una pregunta delante de cada cambio:

¿Qué evidencia mínima necesito para decir que esta versión puede avanzar?

Qué no es EvalOps

EvalOps no es abrir un notebook, mirar diez ejemplos y escribir “parece mejor”. Eso puede servir para exploración, pero no para release.

Tampoco es delegar toda la decisión en un modelo evaluador. Un evaluador puede ayudar, igual que una métrica puede ayudar, pero los criterios deben estar escritos. Si la rúbrica cambia sin versión, si el evaluador cambia sin control o si nadie revisa casos frontera, el número resultante solo da una sensación cómoda.

Y no es tener una tabla enorme de benchmarks públicos para decorar una presentación. Un benchmark externo puede orientar, pero tu producto vive con tus usuarios, tus documentos, tus contratos, tus tools, tus costes y tus límites de latencia. La evaluación útil combina referencias externas con datos propios.

CONFUSIÓN	QUÉ FALTA
“Lo he probado y va bien”	Dataset, repetición, comparación y trazas.
“El evaluador dice 8,7”	Rúbrica, calibración, ejemplos de referencia y auditoría de errores.
“El benchmark sube”	Casos propios, contrato de salida, coste y latencia real.
“Solo cambié el prompt”	Ver si cambian tokens, formato, abstención, tools, ruta y estilo.
“Si falla, hacemos rollback”	Saber qué métrica lo detecta, cuánto tarda y qué versión restaurar.

Qué sí es EvalOps

EvalOps es el ciclo que convierte una versión candidata en una decisión operativa.

Una suite de evaluación se compone de estas piezas. No es una ecuación de la literatura, es una lista de ingeniería, así que la presentamos como tabla:

SÍMBOL O	SIGNIFICADO	EJEMPLO
<i>D</i>	Datasets de evaluación.	Golden, regresión, holdout, shadow y muestra de producción.
<i>R</i>	Runner reproducible.	Ejecuta baseline y candidate con las mismas entradas.
<i>J</i>	Evaluadores.	Código, reglas, LLM-as-judge, revisión humana, métricas RAG o métricas de tool.
<i>M</i>	Métricas.	Calidad, groundedness, JSON válido, coste, p95, tokens, trazas completas.
<i>G</i>	Gates.	Condiciones para pasar a staging, canary o producción.
<i>S</i>	Scorecard.	Documento o artefacto firmado con versiones, resultados y decisión.

Lo importante: EvalOps no vive al final. Empieza cuando se define el cambio. Si no sabemos qué dataset protege una conducta, qué métrica mide el éxito o qué gate puede bloquear, estamos cambiando software sin contrato de calidad.

Fecha de corte del estado del arte

Fecha de corte: 28 de mayo de 2026.

Fuentes consultadas: documentación oficial de OpenAI sobre evals, graders, agent evals y trace grading; documentación de Google ADK sobre evaluación de agentes; LangSmith Evaluation; Arize Phoenix Evaluation; Ragas metrics; Braintrust Evaluation; Promptfoo; OpenTelemetry; Google SRE Workbook; y trabajos clásicos de ingeniería de ML como TFX y *Software Engineering for Machine Learning*.

Lo estable es el método: datasets versionados, experimentos comparables, evaluadores con criterios explícitos, trazas, gates, CI/CD, monitorización online y feedback desde producción. Lo cambiante son productos, APIs, paneles, nombres de métricas, modelos evaluadores, precios, límites de proveedor y benchmarks de moda.

OpenAI documenta la creación y ejecución de evals mediante API y dashboard, con graders y datasets para probar criterios concretos.¹ Google ADK insiste en evaluar tanto respuesta final como trayectoria de un agente, porque los modelos introducen variabilidad y las aserciones deterministas no siempre bastan.² LangSmith separa evaluación offline, antes de publicar, y online, sobre interacciones reales muestreadas.³

Phoenix organiza evals como señales de calidad que pueden adjuntarse a trazas y datasets, con evaluadores por código, modelos y revisión humana.⁴ Braintrust presenta un ciclo de evaluación que pasa por playgrounds, experimentos, CI/CD, scoring en producción y realimentación hacia datasets.⁵ Promptfoo, por su parte, ofrece un enfoque declarativo y de CI para comparar prompts, modelos y pipelines RAG.⁶

Qué evaluar según lo que cambias

Un error habitual es tener una única evaluación para todo. No funciona. Cada cambio toca una parte distinta del sistema.

CAMBIO	QUÉ PUEDE ROMPER	EVALUACIÓN MÍNIMA
Prompt	Formato, tono, longitud, abstención, uso de tools y coste.	Golden set, contrato de salida, tokens, casos de regresión y revisión de trazas.
Modelo	Calidad, latencia, coste, soporte de parámetros, contexto y estabilidad.	Comparación baseline/candidate, p95, coste por tarea aceptada, repetición de casos dudosos.
Parámetros	Variabilidad, creatividad, longitud y cumplimiento de schema.	Repeticiones por caso, flake rate, schema pass rate y distribución de longitud.
RAG	Relevancia de chunks, groundedness, citas, cobertura y ruido.	Context precision, context recall, faithfulness, citas válidas y evaluación por documento.
Router	Rutas elegidas, coste, latencia, capacidad y degradación.	Shadow routing, matriz de tareas, presupuesto, p95 por ruta y diff de decisiones.
Tool	Argumentos, permisos, errores tipados, idempotencia y compensación.	Contract tests, trazas de tool, validación de argumentos y casos de error controlado.
Contrato JSON	Campos obligatorios, tipos, enums, extras y compatibilidad.	Validación determinista, fixtures, migración de schema y ejemplos negativos.
Runtime	Timeouts, batching, retries, colas, memoria y throughput.	Carga sintética, p95/p99, tasa de timeouts, goodput y comparación de trazas.
Evaluador	Cambia la regla con la que medimos.	Eval del evaluador, acuerdo humano, ejemplos calibrados y versionado de rúbrica.

La regla sencilla: **si no sabes qué eval corresponde a un cambio, todavía no entiendes bien su superficie de riesgo técnico.**

CI/CD real: dónde vive EvalOps

Para ingeniería informática, EvalOps no puede quedarse en una carpeta de experimentos. Tiene que entrar en el flujo normal de cambio: pull request, revisión, integración continua, release candidate, canary y producción.

GitHub Actions define workflows como procesos automatizados configurados en YAML y compuestos por jobs.⁷ GitLab CI/CD también usa un archivo YAML para declarar jobs, etapas y reglas de ejecución.⁸ La idea no depende de la plataforma: el gate de IA debe producir un estado automático, artefactos revisables y una decisión clara.

MOMENTO	JOB	QUÉ EJECUTA	QUÉ ARTEFACTO DEJA
Pull request	evalops-smoke	Schema, smoke set, contratos de tool y regresiones críticas.	smoke_report.json , logs y diff de métricas.
Pull request con cambio de IA	evalops-pr	Golden reducido, comparación baseline/candidate, coste estimado.	Comentario en PR y scorecard parcial.
Nightly	evalops-nightly	Golden completo, repeticiones, tags lentos y métricas por segmento.	Serie histórica, dataset failures y traces de muestra.
Release candidate	evalops-release	Regression completo, holdout autorizado, scorecard firmada.	release_scorecard.json y decisión.
Canary	evalops-online	SLIs online, sampling, burn rate, coste p95 y feedback.	Dashboard, alerta y casos nuevos para dataset.

Un ejemplo mínimo en GitHub Actions:

```

name: evalops

on:
  pull_request:
    paths:
      - "prompts/**"
      - "ops/ai/**"
      - "rag/**"
      - "tools/**"
  schedule:
    - cron: "17 2 * * *"
  workflow_dispatch:

jobs:
  smoke:
    runs-on: ubuntu-latest
    timeout-minutes: 10
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.12"
      - name: Install
        run: pip install -r requirements-dev.txt
      - name: Run smoke evals
        run: python ops/ai/evalops_release_gate.py --dataset evals/smoke.jsonl
      - name: Upload scorecard
        uses: actions/upload-artifact@v4
        with:
          name: evalops-scorecard
          path: output/evalops_scorecard.json

  release-gate:
    if: github.event_name == 'workflow_dispatch'
    needs: smoke
    runs-on: ubuntu-latest
    timeout-minutes: 60
    steps:
      - uses: actions/checkout@v4
      - name: Run release gate
        run: python ops/ai/evalops_release_gate.py --dataset evals/release.jsonl --strict

```

Qué debe importarte del YAML:

PIEZA	POR QUÉ IMPORTA
paths	Evita gastar evaluaciones cuando el cambio no toca IA.
timeout-minutes	Un gate que se queda colgado también rompe el flujo de ingeniería.
upload-artifact	La decisión debe dejar evidencia descargable.
workflow_dispatch	Permite ejecutar una release candidate manual y trazable.
needs	Obliga a ordenar smoke antes de release gate.

El patrón equivalente en GitLab cambia la sintaxis, pero no la arquitectura:

```
stages:
  - test
  - eval
  - release

evalops_smoke:
  stage: eval
  image: python:3.12
  rules:
    - changes:
        - prompts/**/*
        - ops/ai/**/*
        - rag/**/*
        - tools/**/*
  script:
    - pip install -r requirements-dev.txt
    - python ops/ai/evalops_release_gate.py --dataset evals/smoke.jsonl
  artifacts:
    when: always
    paths:
      - output/evalops_scorecard.json
```

La pregunta de examen no debería ser “¿sabes escribir YAML?”. Debería ser: **¿qué job bloquea qué decisión y qué evidencia deja para revisarla?**

Contrato JSONL de un caso de evaluación

Un dataset de EvalOps necesita un formato aburrido, versionable y fácil de revisar en Git. JSONL encaja bien porque cada línea es un caso independiente.

Ejemplo formateado para leer. En el archivo `eval_cases.jsonl`, cada objeto iría en una sola línea:

```
{
  "case_id": "support-json-001",
  "task": "support_triage",
  "input": {
    "message": "No puedo acceder a mi matrícula y ya pagué."
  },
  "expected": {
    "categoria": "matricula",
    "prioridad": "alta",
    "must_include": ["comprobar pago", "revisar expediente"]
  },
  "tags": ["json", "soporte", "alta_criticidad"],
  "rubric": {
    "quality_min": 0.85,
    "contract_required": true,
    "citation_required": false
  },
  "why_it_exists": "Protege clasificación urgente con salida estructurada.",
  "source": "manual",
  "source_trace_id": null,
  "owner": "ai-platform",
  "sensitivity": "low",
  "created_at": "2026-05-28"
}
{
  "case_id": "rag-policy-014",
  "task": "policy_qa",
  "input": {
    "question": "¿Puedo entregar fuera de plazo si tengo justificante?"
  },
  "expected": {
    "source_ids": ["policy-academic-2026#late-delivery"],
    "answer_must_be_grounded": true
  },
  "tags": ["rag", "policy", "citas"],
  "rubric": {
    "faithfulness_min": 0.90,
    "context_recall_min": 0.80,
    "abstain_if_missing": true
  },
  "why_it_exists": "Evita responder normativa sin fuente recuperada.",
  "source": "production_sample",
  "source_trace_id": "trace_20260527_7781",
  "owner": "academic-platform",
  "sensitivity": "medium",
  "created_at": "2026-05-28"
}
```

CAMPO	TIPO	REGLA
case_id	string	Estable, único y no reutilizable.
task	string	Debe mapear a una política de routing y gate.
input	objeto	Entrada reproducible, sin datos innecesarios.
expected	objeto	Puede ser salida exacta, propiedades, fuentes o condiciones.
tags	array	Permite análisis por contrato, idioma, ruta, dificultad o criticidad.
rubric	objeto	Umbrales por caso cuando no basta una regla global.
why_it_exists	string	Razón pedagógica u operativa del caso.
source	string	manual , production_sample , incident , benchmark , synthetic .
source_trace_id	string o null	Conecta producción con evaluación sin copiar toda la traza.
owner	string	Equipo responsable de cambiar o retirar el caso.
sensitivity	string	low , medium , high ; decide retención y permisos.

Un buen dataset no solo pregunta “qué debe contestar”. También pregunta “qué propiedad de ingeniería protege este caso”.

Los datasets no son una carpeta de ejemplos

Un dataset de evaluación es un contrato de aprendizaje. Cada caso debería explicar por qué existe.

CAMPO	QUÉ GUARDA	POR QUÉ IMPORTA
case_id	Identificador estable.	Permite seguir el caso durante meses.
input	Entrada que se ejecuta.	Debe ser realista y reproducible.
expected	Salida, propiedades o fuente esperada.	No siempre es una frase exacta; puede ser una rúbrica.
tags	Tema, dificultad, ruta, cliente, idioma, contrato.	Permite ver dónde mejora o empeora.
why_it_exists	Motivo del caso.	Evita acumular ejemplos sin intención.
source	Manual, producción, incidencia, laboratorio, benchmark.	Da contexto y trazabilidad.
sensitivity	Qué puede guardarse y quién puede verlo.	Protege datos y limita muestras.
owner	Persona o equipo responsable.	Alguien debe decidir si el caso cambia.
created_from_trace_id	Traza de origen, si viene de producción.	Cierra el bucle entre operación y evaluación.

No todos los datasets cumplen el mismo papel:

DATASET	TAMAÑO TÍPICO	USO	CAUTION
Smoke	5-20 casos	Detectar roturas obvias en segundos.	No sirve para afirmar calidad.
Golden	50-300 casos	Proteger comportamientos esenciales.	Si lo ajustas demasiado, deja de generalizar.
Regression	Crece con el producto	Evitar que vuelvan fallos corregidos.	Debe etiquetar motivo y versión que lo arregló.
Holdout	Reservado	Medir sin haber optimizado contra esos casos.	No debe mirarse a cada iteración pequeña.
Shadow	Muestra real	Ver qué pasaría con una candidata.	Necesita anonimización, muestreo y coste controlado.
Canary	Tráfico limitado	Confirmar señales reales antes de extender.	Requiere rollback y SLOs claros.

Ragas lista métricas específicas para RAG como context precision, context recall, response relevancy, faithfulness, tool call accuracy y métricas tradicionales como exact match o ROUGE.⁹ Eso no significa que haya que usarlas todas. Significa que cada dataset debe decir qué propiedad mide.

Evaluadores como software

Un evaluador no es una autoridad mística. Es software. Y si es software, se versiona, se prueba, se observa y se puede equivocar.

Tipos de evaluadores:

TIPO	CUÁNDO USARLO	EJEMPLO	RIESGO
Determinista	El contrato es objetivo.	JSON válido, enum permitido, campo obligatorio, regex, SQL ejecuta.	No mide calidad semántica.
Heurístico	Hay reglas baratas aproximadas.	Longitud máxima, presencia de cita, número de tools, coste.	Puede incentivar trucos superficiales.
Métrica clásica	Hay referencia textual o ranking.	Exact match, F1, ROUGE, Hit@k, MRR.	No siempre captura utilidad real.
LLM-as-judge	La calidad exige criterio lingüístico.	Correctitud, groundedness, tono, completitud.	Depende de rúbrica, modelo, temperatura y calibración.
Revisión humana	La decisión tiene impacto alto o ambigüedad real.	Casos límite, cambios de rúbrica, muestras de canary.	Coste y variabilidad entre personas.

Qué debe tener un evaluador serio:

PROPIEDAD	QUÉ SIGNIFICA
<code>evaluator_version</code>	Si cambia la rúbrica, cambia la versión.
<code>input_schema</code>	Qué campos necesita para puntuar.
<code>output_schema</code>	Qué devuelve: <code>score</code> , <code>label</code> , <code>explanation</code> , <code>confidence</code> , <code>evidence</code> .
<code>calibration_set</code>	Casos donde ya sabemos qué debería decidir.
<code>known_failures</code>	Casos donde el evaluador suele confundirse.
<code>cost_budget</code>	Coste máximo por lote de evaluación.
<code>owner</code>	Persona o equipo que responde por la regla.

Un patrón útil es probar al evaluador antes de usarlo para aprobar releases:

```
CALIBRATION = [  
    {  
        "case_id": "judge-cal-001",  
        "input": "Respuesta con cita correcta y fuente recuperada.",  
        "expected_label": "pass",  
    },  
    {  
        "case_id": "judge-cal-002",  
        "input": "Respuesta segura pero sin fuente recuperada.",  
        "expected_label": "fail",  
    },  
]  
  
def test_evaluator_calibration(evaluator):  
    labels = [evaluator(item["input"]).label for item in CALIBRATION]  
    expected = [item["expected_label"] for item in CALIBRATION]  
    assert labels == expected
```

La pregunta de ingeniería: si el evaluador cambia de opinión, ¿es porque la candidata mejoró, porque el evaluador cambió o porque el caso era ambiguo? Si no puedes separar esas tres cosas, tu gate tiene ruido.

La comparación baseline contra candidate

Una release seria no pregunta “¿la candidata es buena?”. Pregunta:

¿La candidata es suficientemente buena, no empeora lo importante y mejora lo que prometía mejorar?

El gate de release se escribe de forma compacta como una función indicador: vale 1, y solo entonces se publica, si se cumplen todas las condiciones a la vez.

$$G(v_c) = 1[Q_c \geq Q_{min} \wedge Q_c \geq Q_b - \delta_q \wedge L_{95,c} \leq L_{95,max} \wedge C_c \leq C_{max} \wedge K_c \geq K_{min}]$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$G(v_c)$	Gate aplicado a la versión candidata.	1 si pasa, 0 si no pasa.
Q_c	Calidad de la candidata.	0,91 de media ponderada.
Q_b	Calidad del baseline.	0,89 de la versión estable.
Q_{min}	Calidad mínima absoluta.	Nunca publicar por debajo de 0,85.
δ_q	Tolerancia de caída aceptable.	Permitir hasta 0,01 si gana mucho en coste.
$L_{95,c}$	Latencia p95 de la candidata.	3200 ms.
$L_{95,max}$	Límite p95 permitido.	4500 ms.
C_c	Coste medio o p95 por tarea aceptada.	0,021 EUR.
C_{max}	Presupuesto máximo.	0,030 EUR.
K_c	Cumplimiento de contrato.	99,4% JSON válido.
K_{min}	Cumplimiento mínimo.	99,0%.

La fórmula no reemplaza criterio. Lo fuerza a aparecer. Si decides aceptar una caída de calidad a cambio de menor coste, escríbelo. Si un contrato JSON no puede caer nunca por debajo de 99,5%, escríbelo. Si el gate cambia, versiona el cambio.

Estadística mínima para no engañarnos

En IA hay ruido. Una misma configuración puede producir respuestas distintas. Un evaluador puede variar. Un proveedor puede tener latencia distinta según hora. Una muestra pequeña puede parecer una mejora enorme por azar.

El primer indicador de inestabilidad es el flake rate:

$$F = \frac{N_{inestable}}{N_{repetido}}$$

SÍMBOLO	SIGNIFICADO
F	Proporción de casos que cambian de resultado al repetir.
$N_{inestable}$	Casos cuyo veredicto no fue consistente.
$N_{repetido}$	Casos repetidos bajo la misma configuración.

Si F sube, no basta con mirar la media. Hay que repetir, separar por tags y entender qué se mueve.

Para proporciones pequeñas, por ejemplo “pasa contrato” o “acierta caso”, conviene mirar un intervalo. Una opción práctica es el límite inferior de Wilson:

$$W_{low} = \frac{\hat{p} + \frac{z^2}{2n} - z \frac{\hat{p}(1-\hat{p})}{n} + \frac{z^2}{4n^2}}{1 + \frac{z^2}{n}}$$

SÍMBOLO	SIGNIFICADO
$\hat{p}$	Proporción observada de éxitos.
n	Número de casos evaluados.
z	Valor de confianza; 1,96 se usa a menudo para 95%.
W_{low}	Límite inferior conservador.

Ejemplo: 19 aciertos de 20 parecen un 95%. Pero con pocos casos, el límite inferior de Wilson es bastante más bajo. Eso nos recuerda que una muestra pequeña no da el mismo nivel de confianza que 950 aciertos de 1000.

¿La versión nueva es mejor, o es ruido? La prueba de McNemar. Comparar dos versiones sobre el mismo dataset es una comparación emparejada: cada caso lo resuelven ambas. Lo que importa no son los aciertos totales sino los casos donde discrepan. La prueba de McNemar mira solo esos y dice si la diferencia es significativa.¹⁰

$$\chi^2 = \frac{(b - c)^2}{b + c}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
b	Casos que la versión nueva acierta y la vieja falla.	18
c	Casos que la vieja acierta y la nueva falla.	7
χ^2	Estadístico; se compara con 3,84 para el 95%.	$(18 - 7)^2 / 25 = 4,84$.

En palabras: los casos que ambas aciertan o ambas fallan no informan sobre cuál es mejor; solo cuentan las discordancias. Con 18 a favor y 7 en contra, $\chi^2 = 4,84 > 3,84$: la mejora es significativa. Si fueran 13 y 12, no lo sería, por mucho que la media suba. Esto es lo que separa «parece mejor» de «es mejor».

¿Tengo casos suficientes? Tamaño de muestra y potencia. Un dataset de eval pequeño no puede detectar mejoras pequeñas: falta potencia. Antes de fiarte de un gate conviene saber cuántos casos necesitas para distinguir la mejora que te importa del azar.¹¹

$$n \approx \frac{(z_{\alpha/2} + z_{\beta})^2 p (1 - p)}{\delta^2}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
δ	Mejora mínima que quieres poder detectar.	0,03
p	Proporción de acierto de referencia.	0,90
$z_{\alpha/2}$	Cuantil del nivel de significación.	1,96 (95%).
z_{β}	Cuantil de la potencia deseada.	0,84 (80%).
n	Casos necesarios.	≈ 392 .

En palabras: para detectar con garantías una mejora de 3 puntos sobre un 90% de base hacen falta del orden de 400 casos por versión. Con 50 casos no puedes afirmar casi nada: por eso un gate honesto declara su poder de detección y no bloquea ni promueve con muestras que no dan para decidir.

¿Cuánto vale esa exactitud? El intervalo exacto de Clopper-Pearson. Un «92% de aciertos» en 100 casos no es un punto, es un rango. Para muestras pequeñas o proporciones extremas, el intervalo exacto de Clopper-Pearson, basado en la distribución binomial, no se pasa de confiado como hacen las aproximaciones.¹²

$$p_{\text{inf}} = B\left(\frac{\alpha}{2}; x, n - x + 1\right), \quad p_{\text{sup}} = B\left(1 - \frac{\alpha}{2}; x + 1, n - x\right)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Número de aciertos.	92
n	Número de casos.	100
$B(\cdot)$	Cuantil de la distribución Beta (inversa binomial).	Da los extremos exactos.
α	Nivel de significación.	0,05

En palabras: 92 de 100 aciertos es «entre 84,8% y 96,5% con 95% de confianza». Un gate que exige «al menos 90%» no debería aprobar con un 92% puntual cuyo intervalo baja del 90%: el método exacto evita aprobar mejoras que el azar podría desmentir mañana.

¿Cuántas «mejoras por segmento» son casualidad? Falsos descubrimientos. Un gate suele mirar muchos segmentos a la vez: idiomas, tipos de consulta, longitudes. Cuantas más comparaciones haces, más «mejoras significativas» aparecen por puro azar. El procedimiento de Benjamini-Hochberg controla la proporción de hallazgos falsos al revisar muchas hipótesis.¹³

$$\text{rechaza } H_{(i)} \iff p_{(i)} \leq \frac{i}{m} q$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$p_{(i)}$	i -ésimo p-valor ordenado de menor a mayor.	Lista de 20 segmentos.
m	Número total de comparaciones.	20
q	Tasa de falsos descubrimientos tolerada.	0,10
i	Posición en el orden.	1, 2, 3...

En palabras: con 20 segmentos y un umbral del 5%, esperarías una «mejora significativa» falsa por puro azar aunque nada haya cambiado. Benjamini-Hochberg sube el listón según el número de comparaciones, así que el gate no celebra ruido. Es lo que evita el autoengaño de «mira, mejora justo en este idioma raro».

Comparación pareada: misma entrada, dos versiones

En sistemas de IA, comparar medias agregadas puede engañar. Si la candidata mejora casos fáciles y empeora casos críticos, la media quizá sube y aun así el producto empeora.

La comparación pareada ejecuta cada caso contra baseline y candidate:

$$\Delta_i = S_c(x_i) - S_b(x_i)$$

SÍMBOLO	SIGNIFICADO
x_i	Caso evaluado.
$S_b(x_i)$	Score del baseline en ese caso.
$S_c(x_i)$	Score de la candidata en ese caso.
Δ_i	Diferencia caso a caso.

Después clasificamos cada caso:

RESULTADO PAREADO	REGLA TÍPICA	QUÉ INDICA
Win	$\Delta_i > \epsilon$	La candidata mejora de forma apreciable.
Tie	\$	Δ_i
Loss	$\Delta_i < -\epsilon$	La candidata empeora.
Contract loss	Baseline cumple contrato y candidate no.	Debe pesar más que una pérdida pequeña de estilo.

Scorecard pareada:

MÉTRICA	VALOR	CÓMO LEERLA
wins	42	Casos donde candidate mejora.
ties	118	Casos equivalentes.
losses	17	Casos donde candidate empeora.
critical_losses	3	Casos que no deberían empeorar sin revisión.
contract_losses	1	Puede bloquear aunque la media mejore.

Si quieres una prueba estadística sencilla para etiquetas binarias, McNemar suele aparecer cuando los mismos casos se evalúan con dos clasificadores y solo importan los desacuerdos. Para un facsímil como este, basta con que el alumno entienda la intuición: **los casos donde ambos aciertan o ambos fallan no distinguen versiones; la decisión vive en los desacuerdos.**

Para scores continuos, una alternativa práctica es bootstrap sobre Δ_i : remuestrear deltas, calcular la media muchas veces y mirar si el intervalo cae claramente por encima de cero. No hace falta convertir esto en ceremonia, pero sí evitar frases como “subió 0,7 puntos” sin mirar variabilidad, tags y criticidad.

Matriz de criticidad por tarea

No todas las tareas merecen el mismo gate. Una recomendación interna reversible no necesita el mismo nivel que una extracción usada para facturación o una acción que modifica un sistema.

CRITICIDAD	SEÑAL	GATE MÍNIMO
Baja	Respuesta informativa, reversible, sin contrato fuerte.	Smoke, calidad mínima, coste y latencia.
Media	Soporte, resumen, clasificación o RAG con fuente.	Golden, regression, contrato, groundedness, coste p95 y revisión de pérdidas.
Alta	Salida que alimenta procesos, usuarios o datos importantes.	Holdout, comparación pareada, contrato estricto, revisión humana de pérdidas críticas y canary.
Muy alta	Acción externa, cambio de estado o decisión sensible.	Aprobación explícita, trazas completas, doble gate, rollback probado y muestreo online.

Variables para decidir criticidad:

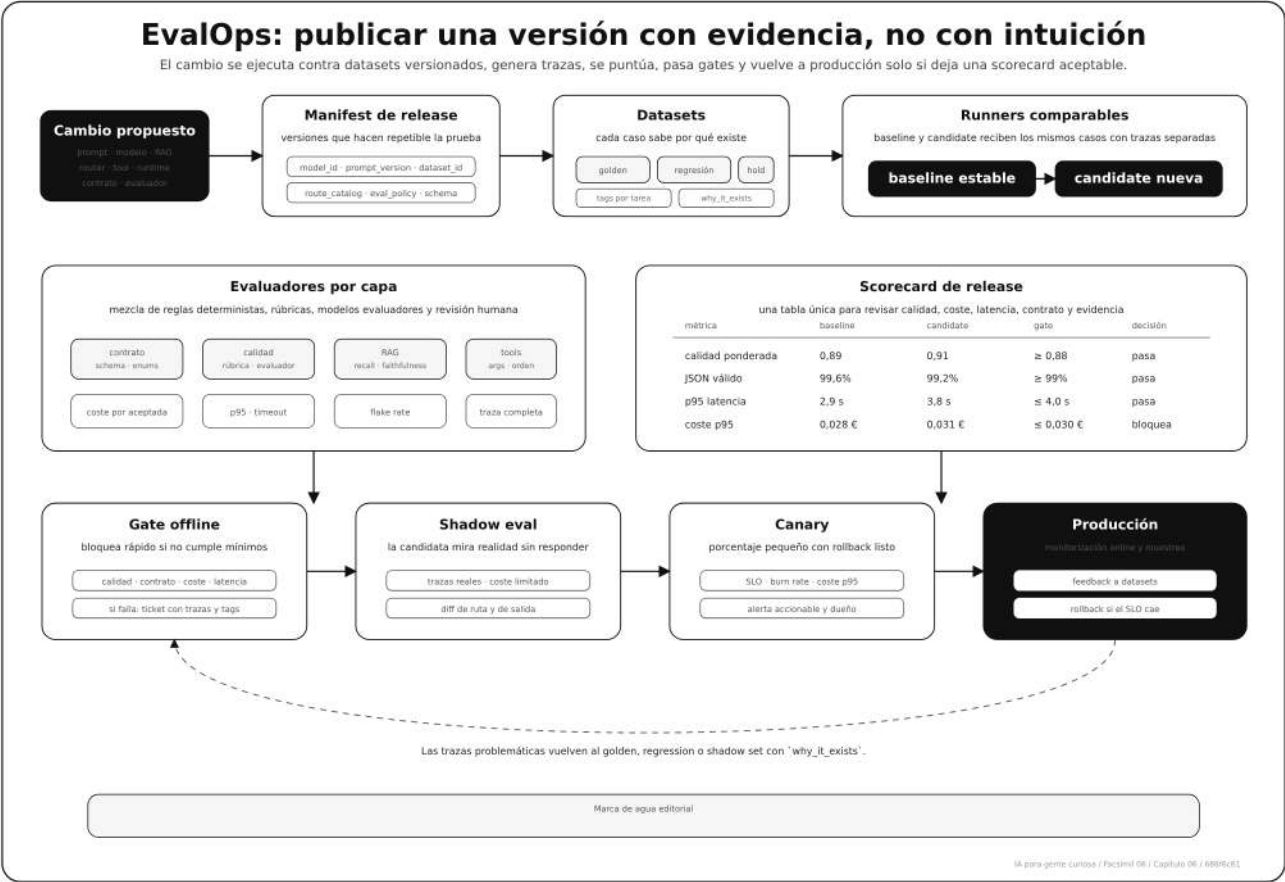
VARIABLE	PREGUNTA
Reversibilidad	¿Se puede deshacer sin daño operativo?
Dependencia downstream	¿Otra parte del sistema confía en esta salida?
Contrato	¿Hay schema, tipo, enum o cita obligatoria?
Exposición de datos	¿La run toca datos internos, personales o documentos sensibles?
Coste de error	¿Qué pasa si una respuesta incorrecta se acepta?
Frecuencia	¿Ocurre una vez al mes o miles de veces al día?
Supervisión	¿Hay revisión humana antes de ejecutar la consecuencia?

Una política simple:

```
criticality_policy:
  support_summary:
    level: medium
    required_gates: [smoke, golden, regression, cost_p95]
  billing_extraction:
    level: high
    required_gates: [schema, paired_eval, holdout, human_review_losses]
  external_action:
    level: very_high
    required_gates: [schema, trace_complete, approval, canary, rollback_drill]
```

El buen diseño no consiste en poner todos los gates a todo. Consiste en que el nivel de prueba acompañe al impacto.

Anatomía visual de un pipeline EvalOps



Gates por entorno

No todos los gates tienen el mismo coste. Un gate de pull request debe ser rápido. Un gate nocturno puede tardar más. Un gate de prepublicación puede ejecutar datasets grandes. Un gate de canary ya mira señales reales.

ENTORNO	OBJETIVO	QUÉ EJECUTA	TIEMPO RAZONABLE
PR	Detectar roturas evidentes.	Smoke, schema, tests de tool, casos de regresión críticos.	Segundos o pocos minutos.
Nightly	Ver tendencias.	Golden completo, repetición de casos inestables, comparación de costes.	Minutos u horas.
Prepublicación	Decidir si se avanza.	Golden, regression, holdout controlado, scorecard y revisión.	Lo que permita el proceso.
Shadow	Comparar con realidad sin afectar respuesta.	Muestra real, diff de rutas, costes estimados y trazas.	Horas o días.
Canary	Confirmar en uso limitado.	SLIs online, alertas, feedback y rollback preparado.	Depende del tráfico.

La clave es que cada entorno responda una pregunta distinta:

PREGUNTA	ENTORNO
¿Rompimos algo obvio?	PR.
¿La tendencia va bien?	Nightly.
¿Podemos aprobar la candidata?	Prepublicación.
¿Qué habría pasado con tráfico real?	Shadow.
¿Qué pasa con un porcentaje pequeño de usuarios?	Canary.

Herramientas: qué aporta cada una

Las herramientas no sustituyen el criterio, pero evitan que tengas que construirlo todo desde cero.

HERRAMIENTA	QUÉ APORTA	CUÁNDO ENCAJA
OpenAI Evals y Graders	Datasets, criterios, graders y ejecución de evals integradas con modelos OpenAI.	Cuando tu ciclo de mejora vive cerca de OpenAI y necesitas graders versionables.
Google ADK Evaluate	Evaluación de respuesta final y trayectoria de agentes con test files y evalsets.	Cuando construyes agentes con ADK y quieres evaluar tools y pasos intermedios.
LangSmith Evaluation	Datasets, experimentos, comparación offline, evaluadores online y feedback loop.	Cuando quieres unir trazas, datasets y experimentos en productos LangChain o LangGraph.
Phoenix Evals	Evals sobre trazas, datasets, RAG, tool use, evaluadores por código, modelo o revisión.	Cuando quieres observabilidad y evaluación con fuerte integración OpenTelemetry.
Braintrust	Experimentos, scorers, CI/CD, online scoring y datasets a partir de producción.	Cuando el equipo quiere un workflow completo de evaluación y release.
Ragas	Métricas de RAG, tool use, SQL, comparación semántica y generación de testsets.	Cuando el problema principal es retrieval, grounding o aplicaciones RAG.
Promptfoo	Configuración declarativa, matriz de prompts/modelos, CLI, CI y métricas.	Cuando quieres pruebas rápidas y reproducibles desde repositorio.
pytest + scripts propios	Control total, barato y cercano al código.	Cuando el contrato es determinista o el equipo necesita empezar sin plataforma.

Una recomendación práctica: empieza con `pytest` y datasets pequeños si aún no tienes disciplina. Cuando el equipo empiece a comparar muchas variantes, guardar trazas, revisar casos y automatizar releases, una plataforma de evals deja de ser lujo y se vuelve infraestructura.

Coste de evaluar

Evaluar también consume dinero, tiempo y capacidad. Ignorarlo lleva a dos extremos malos: no evaluar casi nada porque “sale caro”, o evaluar todo con el modelo más potente hasta que el coste hace que el equipo apague el sistema.

El coste de un lote de evaluación es la suma del coste de cada caso, contando la run, el juez y las tools:

$$C_{eval} = \sum_{i=1}^n (C_{run,i} + C_{judge,i} + C_{tools,i})$$

SÍMBOLO	SIGNIFICADO
C_{eval}	Coste total del lote de evaluación.
$C_{run,i}$	Coste de ejecutar el sistema en el caso i .
$C_{judge,i}$	Coste de evaluadores con modelo o revisión.
$C_{tools,i}$	Coste de búsquedas, bases de datos, runtimes o servicios externos.

Palancas de optimización:

PALANCA	QUÉ AHORRA	CAUIDADO
Smoke set pequeño	Tiempo en PR.	No sustituye golden completo.
Caché de outputs	Repetir evaluaciones sin recalcular todo.	Invalida si cambia modelo, prompt, datos o tool.
Evaluadores deterministas primero	Llamadas a modelos evaluadores.	No captura calidad semántica.
Muestreo por tags	Ejecutar más donde hubo cambios.	Puede dejar zonas sin cobertura.
Batch nocturno	Coste y saturación diurna.	Los fallos llegan más tarde.
Paralelismo limitado	Tiempo de ejecución.	Puede topar con rate limits o colas.
Evaluador pequeño calibrado	Coste de evaluación.	Debe compararse contra revisión o evaluador más fuerte.

Una política útil:

```
eval_budget:
  pull_request:
    max_cost_eur: 2
    max_runtime_minutes: 10
    datasets: [smoke, critical_regression]
  nightly:
    max_cost_eur: 40
    max_runtime_minutes: 120
    datasets: [golden, regression]
  release_candidate:
    max_cost_eur: 120
    max_runtime_minutes: 240
    datasets: [golden, regression, approved_holdout]
```

La frase importante: **si el coste de evaluar no está presupuestado, el equipo acabará evaluando menos de lo que cree.**

Privacidad y datasets de evaluación

Los mejores casos suelen venir de producción. También son los que más cuidado exigen. No necesitamos copiar todo lo que vio el usuario para aprender del fallo.

Una traza puede convertirse en caso de evaluación de varias formas:

TÉCNICA	QUÉ GUARDA	CUÁNDO SIRVE
ID de documento	<code>source_ids</code> , versión de índice y hash.	Cuando el contenido está en un repositorio controlado.
Resumen técnico	Propiedad del fallo, no texto completo.	Cuando basta con reproducir el patrón.
Redacción mínima	Entrada reducida que conserva el comportamiento.	Cuando se puede quitar información sensible.
Caso sintético derivado	Caso nuevo creado a partir del fallo.	Cuando la traza real no debe entrar al dataset.
Retención corta	Guardar muestra solo para diagnóstico.	Cuando hay que revisar rápido y borrar después.

Campos recomendables:

CAMPO	PARA QUÉ SIRVE
<code>sensitivity</code>	Decide quién puede ver el caso.
<code>retention_days</code>	Evita datasets eternos con datos innecesarios.
<code>redaction_status</code>	<code>raw</code> , <code>redacted</code> , <code>synthetic</code> , <code>metadata_only</code> .
<code>allowed_runners</code>	Define si puede ejecutarse local, cloud o solo entorno interno.
<code>review_required</code>	Obliga a revisión antes de entrar en golden o holdout.

La regla práctica: el dataset debe conservar **la propiedad que queremos probar**, no necesariamente el contenido original.

Scorecard: el documento que autoriza avanzar

La scorecard es el artefacto que impide discutir de memoria. Puede ser JSON, Markdown, una tabla en la plataforma o un registro en CI. Debe responder:

CAMPO	PREGUNTA QUE RESPONDE
<code>candidate_version</code>	¿Qué estamos intentando publicar?
<code>baseline_version</code>	¿Contra qué se compara?
<code>change_summary</code>	¿Qué cambió y por qué?
<code>datasets</code>	¿Con qué casos se midió?
<code>evaluators</code>	¿Quién o qué puntuó?
<code>metrics</code>	¿Qué números importan?
<code>gates</code>	¿Qué umbrales decidían?
<code>exceptions</code>	¿Qué aceptamos manualmente y con qué motivo?
<code>owner</code>	¿Quién firma la decisión?
<code>rollback_plan</code>	¿Cómo se revierte si empeora online?

Una scorecard útil no intenta demostrar que todo es perfecto. Intenta dejar una decisión auditada:

```
release_scorecard:
  candidate_version: assistant-runtime@2026.05.28-rc2
  baseline_version: assistant-runtime@2026.05.20
  datasets:
    - golden_support_es@v14
    - regression_json_contract@v8
    - rag_policy_holdout@v3
  gates:
    quality_weighted_min: 0.88
    json_valid_min: 0.99
    p95_latency_ms_max: 4500
    p95_cost_eur_max: 0.030
  decision: promote_to_canary
  owner: ai-platform
  rollback: restore assistant-runtime@2026.05.20 and route_catalog@v31
```

Runbook cuando falla un gate

Un gate que solo dice “fail” no ayuda. Debe abrir un camino de trabajo.

FASE	ACCIÓN	DUEÑO
1. Clasificar	Identificar si falló contrato, calidad, coste, latencia, trazas, RAG, tool o evaluador.	Guardia de release o responsable de plataforma.
2. Aislar	Separar casos por tags, ruta, modelo, proveedor, prompt y versión de dataset.	Equipo que propuso el cambio.
3. Reproducir	Ejecutar baseline y candidate sobre casos fallidos con trazas completas.	Ingeniería.
4. Decidir	Parar, corregir, aceptar excepción limitada o reducir alcance.	Owner del producto y owner técnico.
5. Registrar	Añadir caso de regresión si el fallo era real.	Quien corrige.
6. Reintentar	Lanzar de nuevo el gate con nueva versión.	CI/CD.
7. Vigilar	Si llegó a canary, mirar SLO, coste y feedback.	Operación.

Plantilla de incidencia de gate:

```
evalops_gate_failure:
  release_candidate: assistant-runtime@2026.05.28-rc2
  failed_gate: cost_p95_ok
  first_detected_in: evalops-release
  affected_tags: [rag, policy]
  baseline_version: assistant-runtime@2026.05.20
  candidate_version: assistant-runtime@2026.05.28-rc2
  evidence:
    scorecard: output/evalops_scorecard.json
    traces: output/failing_traces/
    cases: [rag-policy-014, rag-policy-027]
  decision: reduce_context_top_k_and_retry
  regression_cases_to_add: [rag-policy-027]
  owner: ai-platform
```

Una excepción también debe escribirse. A veces una candidata pierde en un caso poco importante pero arregla un problema mayor. Puede aceptarse, pero con alcance, fecha y dueño:


```
gate_exception:
  case_id: support-style-044
  reason: "La candidata usa un tono más breve; producto lo acepta para esta release."
  expires_at: "2026-06-15"
  owner: product-ai
  required_follow_up: "Revisar rúbrica de concisión en golden set."
```

Si una excepción no caduca, no es una excepción: es una nueva política sin admitir.

En el día a día

EvalOps aparece el día en que alguien quiere desplegar un cambio y la pregunta deja de ser «¿está mejor?» para convertirse en «¿cómo lo sabemos?». En un prototipo, se prueba a mano y se decide por sensación; en un sistema operable, hay un dataset de regresión, unas métricas y un gate que dice sí o no con criterios fijados de antemano. La escena típica: alguien afirma que el nuevo prompt mejora la calidad, ejecuta la suite y descubre que sí sube la calidad media, pero rompe dos casos críticos y dispara la latencia. Sin EvalOps, eso se descubre en producción; con EvalOps, en el pipeline.

La segunda situación es la del cambio invisible. Un proveedor actualiza un modelo, o alguien toca el retrieval, y de pronto el sistema responde distinto sin que nadie haya tocado «el código». La evaluación continua es la red que detecta esa deriva: si la regresión cae, el gate bloquea y obliga a mirar qué cambió, en vez de enterarse por una queja de un usuario tres semanas después.

Por qué debería importarte

Porque sin gates de release, cada despliegue es una apuesta y cada mejora puede ser un retroceso disfrazado. EvalOps convierte «creo que ha mejorado» en «lo he medido, aquí están los números y por eso pasa o no pasa», que es la única base honesta para decidir si un cambio sale. Y convierte la calidad en algo que se defiende con datos en una reunión, no con entusiasmo.

Hay además una razón que se nota con el tiempo. Un equipo con EvalOps puede moverse rápido sin miedo, porque cada cambio pasa por el mismo filtro objetivo; un equipo sin ello se mueve despacio y con miedo, porque cada despliegue podría romper algo que nadie está mirando. La velocidad sostenible no viene de probar menos, sino de medir mejor.

Dónde solía tropezar yo

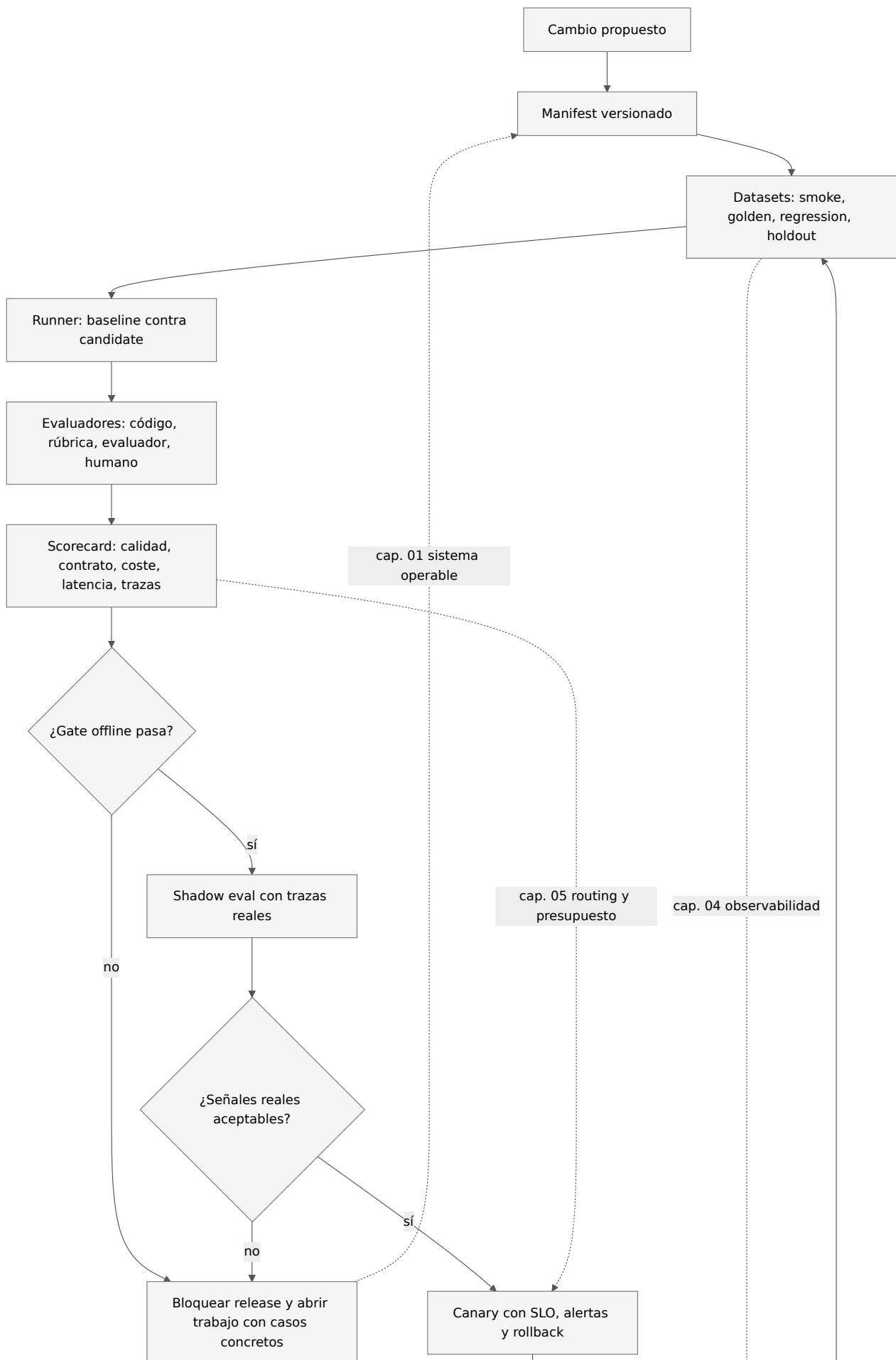
Durante mucho tiempo confundía “tener evals” con “estar evaluando bien”. Tenía un dataset, corría un script y miraba un score. Sonaba serio, pero faltaban cosas.

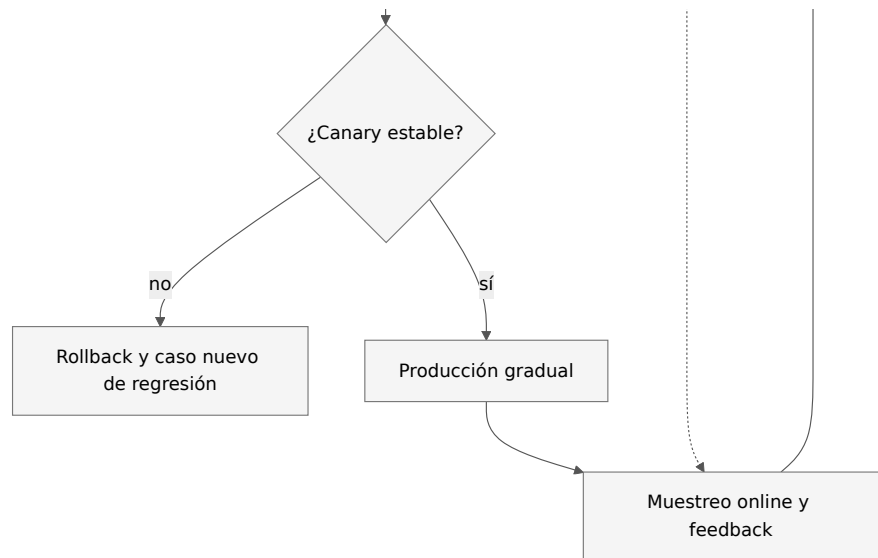
Me tropezaba en cinco sitios:

TROPIEZO	QUÉ APRENDÍ A MIRAR
Optimizar contra el golden set.	Si ajustas siempre contra los mismos casos, necesitas holdout.
Celebrar la media.	Una media puede esconder una caída fuerte en un tag crítico.
No versionar evaluadores.	Si cambia la rúbrica, cambió la regla del juego.
Ignorar coste y latencia.	Una respuesta un poco mejor puede no compensar si duplica p95 o factura.
No devolver producción al dataset.	Los casos reales que fallan son oro para la siguiente evaluación.

La frase que me sirve: **una eval sin decisión operativa es solo una medición; una eval con gate, dueño y feedback empieza a ser ingeniería.**

Cómo encaja todo





Relación con otros capítulos

EvalOps se apoya en piezas que ya hemos construido:

CAPÍTULO	QUÉ APORTA A EVALOPS
F6 · Capítulo 01	La idea de pasar de prototipo a sistema operable con contratos y evidencia.
F6 · Capítulo 02	Run, estado, contrato operativo, colas e idempotencia.
F6 · Capítulo 04	Logs, métricas, trazas, SLI, SLO y presupuesto de error.
F6 · Capítulo 05	Router, fallback, coste, latencia y presupuesto por tarea.
F5 · Capítulo 10	Evaluación de agentes por trayectoria, tools, permisos y coste.
F4 · Capítulo 10	Evaluación de RAG: retrieval, groundedness, abstención y citas.
F4 · Capítulo 13	Cierre de la caja de herramientas con evidencia, evaluación y trazas.

La diferencia es de nivel. En el facsímil 4 evaluamos una técnica concreta. En el facsímil 5 evaluamos agentes. Aquí evaluamos **el proceso de publicar cambios**.

Para entenderlo

Piensa en tres situaciones:

| Situación | Qué haría alguien improvisando | Qué hace EvalOps | |---|---| | Cambias el modelo por uno más barato. | Mira cinco respuestas y publica. | Compara calidad, p95, coste, contrato, rutas y tags críticos. | | El RAG trae más documentos. | Celebra que “hay más

contexto”. | Mide recall, precision, ruido, coste, latencia y groundedness. | | El router manda más tareas a local. | Mira que baja la factura. | Verifica que las tareas locales siguen pasando golden y que el fallback cubre límites. |

El punto no es desconfiar de cada mejora. El punto es que la mejora pueda defenderse con datos.

Vocabulario aprendido

TÉRMINO	QUÉ SIGNIFICA AQUÍ	CÓMO LO USARÍAS EN UN PROYECTO
EvalOps	Disciplina de evaluación versionada antes, durante y después de publicar cambios de IA.	Convertir cambios de prompt, modelo, RAG o tool en gates revisables.
Baseline	Versión estable contra la que se compara la candidata.	No decir “mejora” sin declarar qué versión estás superando.
Candidata	Versión nueva que quiere avanzar.	Llevar <code>model_id</code> , prompt, índice, contrato y ruta en el manifest.
Golden set	Casos importantes y revisados que protegen comportamiento esperado.	Evaluar regresiones de tareas críticas antes de canary.
Holdout	Casos reservados para evitar ajustar todo al mismo dataset.	Detectar si has optimizado contra tus propios ejemplos.
Scorecard	Documento o JSON con métricas, gates, fallos y decisión.	Adjuntarlo al PR o a la release para que otra persona pueda auditar.
Gate	Regla que bloquea, permite revisar o permite avanzar.	Traducir calidad, contrato, coste, latencia y trazas en una decisión.
Regresión	Caso que antes pasaba y ahora falla, o señal que empeora fuera de umbral.	Añadirlo al dataset para que el mismo fallo no vuelva silenciosamente.
Shadow eval	Evaluación con trazas o entradas reales sin exponer la salida candidata.	Medir comportamiento real antes de canary.
Canary	Exposición controlada de la candidata a una parte del tráfico.	Subir porcentaje solo si pasan SLO, contrato y coste.

Antes de pasar página

Comprueba que puedes responder sin mirar:

1. ¿Qué diferencia hay entre smoke, golden, regression, holdout, shadow y canary?
2. ¿Por qué cambiar un prompt puede exigir medir coste, latencia y contrato, no solo calidad?
3. ¿Qué significa comparar baseline contra candidate?
4. ¿Por qué un modelo evaluador necesita rúbrica, calibración y versión?
5. ¿Qué debería contener una scorecard de release?
6. ¿Por qué un gate de PR no debe ejecutar lo mismo que un gate de prepublicación?
7. ¿Cómo vuelve una traza de producción al dataset de evaluación?

En resumen

EvalOps es la práctica de convertir cambios de IA en decisiones de ingeniería. No pregunta si una respuesta parece bonita; pregunta si una versión candidata mejora o mantiene lo importante bajo contrato, presupuesto, trazas y umbrales explícitos.

El patrón completo es: manifest, datasets, runner, evaluadores, scorecard, gate offline, shadow, canary, producción y feedback. Cuando ese ciclo existe, cambiar prompts, modelos, rutas o tools deja de ser una apuesta y empieza a ser una operación defendible.

Para saber más

- OpenAI. *Working with evals*. <https://developers.openai.com/api/docs/guides/evals>
- Google ADK. *Why Evaluate Agents*. <https://adk.dev/evaluate/>
- LangSmith. *Evaluation*. <https://docs.langchain.com/langsmith/evaluation>
- Arize Phoenix. *Evaluation concepts*. <https://arize.com/docs/phoenix/evaluation/concepts-evals/evaluation>
- Braintrust. *Evaluate systematically*. <https://www.braintrust.dev/docs/evaluate>
- Ragas. *List of available metrics*. https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/
- Promptfoo. *Intro*. <https://www.promptfoo.dev/docs/intro/>

Notas

1. OpenAI. (2026). *Working with evals*. <https://developers.openai.com/api/docs/guides/evals>. Consultado el 28 de mayo de 2026. ↵
2. Google. (2026). *Why Evaluate Agents*. <https://adk.dev/evaluate/>. Consultado el 28 de mayo de 2026. ↵
3. LangChain. (2026). *LangSmith Evaluation*. <https://docs.langchain.com/langsmith/evaluation>. Consultado el 28 de mayo de 2026. ↵
4. Arize Phoenix. (2026). *Evaluation concepts*. <https://arize.com/docs/phoenix/evaluation/concepts-evals/evaluation>. Consultado el 28 de mayo de 2026. ↵
5. Braintrust. (2026). *Evaluate systematically*. <https://www.braintrust.dev/docs/evaluate>. Consultado el 28 de mayo de 2026. ↵
6. Promptfoo. (2026). *Intro*. <https://www.promptfoo.dev/docs/intro/>. Consultado el 28 de mayo de 2026. ↵
7. GitHub. (2026). *Workflow syntax for GitHub Actions*. <https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax>. Consultado el 28 de mayo de 2026. ↵
8. GitLab. (2026). *CI/CD YAML syntax reference*. <https://docs.gitlab.com/ci/yaml/>. Consultado el 28 de mayo de 2026. ↵
9. Ragas. (2026). *List of available metrics*. https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/. Consultado el 28 de mayo de 2026. ↵
10. McNemar, Q. (1947). Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2), 153-157. <https://doi.org/10.1007/BF02295996>. Prueba para proporciones correlacionadas en diseños emparejados. ↵
11. Cohen, J. (1988). *Statistical Power Analysis for the Behavioral Sciences* (2.^a ed.). Lawrence Erlbaum. Relaciona tamaño del efecto, nivel de significación, potencia y tamaño de muestra. ↵
12. Clopper, C. J., & Pearson, E. S. (1934). The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4), 404-413. <https://doi.org/10.1093/biomet/26.4.404>. Intervalo de confianza exacto para una proporción binomial. ↵

- .3. Benjamini, Y., & Hochberg, Y. (1995). Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society: Series B*, 57(1), 289-300. <https://doi.org/10.1111/j.2517-6161.1995.tb02031.x>. Controla la tasa de falsos descubrimientos en pruebas múltiples. ↵

Capítulo 07: Cambios progresivos: shadow, canary y rollback

Qué deberías poder hacer al terminar

En el capítulo 06 dejamos una idea clara: una versión candidata no debería llegar a producción solo porque parece mejor. Debe pasar por evidencia: dataset, evaluadores, scorecard y gates. Ahora toca la siguiente pregunta: **si el gate pasa, cómo la exponemos sin apostar todo el sistema a la primera tirada.**

Los sistemas de IA tienen una superficie de cambio más amplia que muchas aplicaciones clásicas. Puedes cambiar código, prompt, modelo, cuantización, runtime, catálogo de rutas, índice RAG, política de abstención, contrato JSON, tool schema, coste máximo o umbral de fallback. Muchos de esos cambios no se ven como una nueva pantalla, pero cambian lo que el sistema responde.

Al terminar, deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Separar deployment de release.	Puedes desplegar una versión sin exponerla todavía.
Diseñar shadow run.	Copias entradas reales sin producir efectos persistentes ni duplicar acciones.
Diseñar canary.	Repares tráfico de forma estable, medible y reversible.
Definir criterios de avance.	Cada porcentaje tiene métricas, mínimos, duración y dueño.
Preparar rollback.	Sabes volver por flag, configuración, ruta, prompt o versión.
Elegir estrategia.	Distingues rolling, blue/green, dark launch, percentage rollout, guarded rollout y canary.
Construir un controlador mínimo.	Simulas asignación por cohorte y decisión de avanzar, pausar o volver.

La idea central: **un cambio serio no se “lanza”; se expone progresivamente, se mide por segmentos y conserva una salida de vuelta.**

La release que no debería ser un salto al vacío

Imagina que el asistente de soporte usa `prompt_v12` , `model_a` , `rag_index_2026_05` y `router_policy_31` . El capítulo 06 nos ayudó a decidir que una candidata, `prompt_v13` con `router_policy_32` , pasa el gate offline. Mejora respuestas largas y reduce llamadas caras. Bien.

Pero producción no es el dataset. En producción hay usuarios con preguntas raras, tenants con documentos distintos, momentos de carga, límites de proveedor, colas, documentos recién subidos y prompts que combinan mal con ciertos casos. Si pasamos de 0% a 100%, cualquier sorpresa cae sobre todo el mundo.

Un rollout progresivo cambia la pregunta:

¿Qué porcentaje pequeño, durante cuánto tiempo y con qué métricas nos permite aprender sin comprometer todo el sistema?

Deployment no es release

En conversaciones de equipo se mezclan mucho estas dos palabras. Conviene separarlas:

CONCEPTO	QUÉ CAMBIA	EJEMPLO
Deployment	El código o configuración queda disponible.	Desplegamos <code>prompt_v13</code> y <code>router_policy_32</code> en producción, pero desactivados.
Release	La capacidad se expone a tráfico real.	El 5% de <code>support_summary</code> usa <code>prompt_v13</code> .

OpenFeature describe los feature flags como decisiones evaluadas en runtime que permiten alterar el comportamiento sin desplegar código nuevo.¹ Esa separación es la base de muchos cambios progresivos: el código puede estar desplegado y la release puede seguir controlada por una bandera, una regla de targeting o una política del router.

Kubernetes, por su parte, ofrece Deployments para gestionar Pods y ReplicaSets y actualizar el estado deseado de una carga de trabajo.² Eso resuelve una parte: cómo llegan contenedores nuevos. Pero una release de IA también puede depender de configuración viva: qué modelo se usa, qué prompt está activo, qué índice RAG se consulta y qué porcentaje de tráfico entra en la variante.

Qué no es un canary

Canary no es “subir al 10% y esperar”. Tampoco es una excusa para saltarse EvalOps. Si una candidata no pasó eval offline, canary no la convierte en segura; solo desplaza el experimento a producción.

Canary tampoco debería ser aleatorio en cada petición. Si el mismo usuario o tenant cae a veces en baseline y a veces en candidate, será difícil comparar, depurar y explicar diferencias. Para muchas tareas conviene asignación estable por cohorte: usuario, tenant, tarea, conversación o documento.

Y rollback no es “ya veremos cómo volvemos”. Rollback debe estar probado antes de exponer tráfico. En IA, volver puede significar restaurar un prompt, bajar un porcentaje, cambiar ruta, recuperar un índice, volver al contrato anterior o desactivar una tool.

CONFUSIÓN	QUÉ FALTA
“Canary es probar en producción”	Falta gate offline, porcentaje, métricas, duración y rollback.
“Deployment y release son lo mismo”	Falta flag, targeting o política que se pueda cambiar sin redespargar.
“Si falla, hacemos rollback”	Falta saber qué revertir, cuánto tarda y qué datos deja la versión candidata.
“Shadow es ejecutar dos veces y comparar”	Falta evitar efectos persistentes, duplicar coste sin límite o llamar tools de escritura.
“Subimos por usuarios al azar”	Falta cohortes estables y segmentación por tarea, tenant o criticidad.

Qué sí es progressive delivery en IA

Progressive delivery es una forma de publicar cambios reduciendo el radio de impacto. En IA generativa, el cambio puede estar en muchas capas:

CAPA	CAMBIO PROGRESIVO POSIBLE
Prompt	Activar <code>prompt_v13</code> por porcentaje, tarea o tenant.
Modelo	Comparar <code>model_a</code> y <code>model_b</code> con rutas separadas.
RAG	Probar un índice nuevo en shadow antes de usarlo como fuente.
Router	Ejecutar <code>router_policy_32</code> en modo decisión sombra.
Tool	Activar una tool primero en modo dry-run o solo lectura.
Contrato	Aceptar nuevo schema solo en consumidores preparados.
Runtime	Mover porcentaje a un serving nuevo con límites de capacidad.

Una release progresiva se describe con estas piezas. No es una ecuación de la literatura, es una lista de ingeniería:

SÍMBOLO	SIGNIFICADO	EJEMPLO
v_b	Versión baseline.	<code>support-rag@1.8.0</code> .
v_c	Versión candidata.	<code>support-rag@1.9.0-rc1</code> .
W	Secuencia de pesos de tráfico.	0%, shadow, 1%, 5%, 25%, 50%, 100%.
M	Métricas observadas por paso.	p95, coste, contrato, aceptación, fallback, calidad online.
G	Gates de avance.	Condiciones para pasar de un peso al siguiente.
B	Plan de rollback.	Flag, configuración previa, ruta anterior e índice anterior.

Argo Rollouts define canary como una estrategia donde una versión nueva recibe un porcentaje pequeño del tráfico de producción, con pasos como `setWeight` y `pause` para controlar el avance.³ LaunchDarkly separa opciones como percentage rollouts, progressive rollouts, guarded rollouts y experiments según si se quiere asignar porcentaje fijo, aumentar tráfico con el tiempo, vigilar métricas o comparar variantes.⁴ Google Cloud Deploy describe canary como un despliegue progresivo que divide tráfico entre una versión ya desplegada y una nueva antes de llegar al despliegue completo.⁵

Estrategias de cambio progresivo

No existe una única estrategia buena. Hay familias, y cada una protege algo distinto.

ESTRATEGIA	QUÉ HACE	SIRVE CUANDO	CUIDADO
Rolling update	Sustituye instancias gradualmente.	Cambio de código con compatibilidad clara.	No controla comportamiento por usuario o tarea.
Blue/green	Mantiene dos entornos y conmuta tráfico.	Necesitas vuelta rápida a entorno anterior.	Duplica infraestructura y no mide bien por segmentos.
Dark launch	Despliega sin exponer al usuario.	Quieres probar wiring, dependencias y observabilidad.	No prueba aceptación real.
Shadow run	Ejecuta candidate en paralelo sin afectar salida.	Quieres comparar con entradas reales.	Debe evitar efectos persistentes y coste ilimitado.
Percentage rollout	Envía un porcentaje fijo a candidate.	Quieres muestra controlada.	Sin métricas puede ser solo azar administrado.
Progressive rollout	Sube porcentaje por pasos.	Quieres aprender antes de ampliar.	Cada paso necesita criterio de avance.
Guarded rollout	Sube tráfico vigilando métricas.	Quieres detener si empeoran señales.	Depende de métricas fiables y umbrales claros.
Canary por cohorte	Expone tenant, tarea o segmento específico.	Quieres controlar impacto por dominio.	Puede sesgar resultados si la cohorte no representa bien.

En IA, muchas veces combinamos varias. Por ejemplo:

1. Dark launch de `model_b` en el runtime.
2. Shadow run con entradas reales de `support_summary`.
3. Canary al 1% por cohorte estable.
4. Progressive rollout 5%, 25%, 50%.
5. Guarded rollout con rollback automático si contrato, p95 o coste empeoran.

Matriz de elección de estrategia

La pregunta práctica no es “qué patrón está de moda”. La pregunta buena es: **qué necesito aprender sin exponer más superficie de la necesaria**. Esta matriz sirve para elegir.

CAMBIO	RIESGO PRINCIPAL	ESTRATEGIA PREFERENTE	POR QUÉ
Prompt de respuesta libre	Calidad, tono, longitud y coste.	Shadow y canary por tarea.	Puedes comparar salidas y después exponer solo una tarea.
Prompt con JSON estricto	Fallo de contrato.	Shadow con validador y canary pequeño.	El contrato debe romperse antes de llegar al usuario.
Modelo nuevo	Latencia, coste, calidad y límites de proveedor.	Shadow, canary por cohorte y fallback.	La misma entrada puede comportarse distinto en tokens, rutas y coste.
Índice RAG nuevo	Recuperación peor o fuentes incorrectas.	Shadow de retrieval y canary por tenant.	La salida puede parecer correcta aunque la fuente haya cambiado.
Tool de lectura	Latencia, permisos y formato de argumentos.	Dark launch, dry-run y canary por tarea.	Conviene probar wiring y trazas antes de exponer.
Tool de escritura	Efectos persistentes.	No usar shadow real con escritura; empezar con modo aprobación o dry-run.	La versión candidata no debe duplicar acciones.
Runtime de inferencia	Saturación, colas y p95.	Blue/green o canary de tráfico.	Aquí importa tanto la arquitectura como el modelo.
Schema de salida	Compatibilidad con consumidores.	Expand-contract.	Los consumidores antiguos y nuevos deben convivir durante la migración.

Un alumno debería poder justificar su elección así: “uso shadow porque necesito comparar entradas reales sin cambiar salida; uso canary por tenant porque el índice RAG depende de documentos; uso rollback por flag porque el cambio está en prompt y ruta, no en código”.

Compatibilidad: dos versiones conviven

Una release progresiva no solo reparte tráfico. Durante un rato conviven dos mundos: baseline y candidate. Esa convivencia exige compatibilidad.

SUPERFICIE	QUÉ PUEDE ROMPER	REGLA DE COMPATIBILIDAD
JSON de respuesta	Consumidores esperan campos antiguos.	Añadir campos antes de quitar campos.
Tool schema	El modelo manda argumentos nuevos.	Versionar schema y aceptar ambos durante la transición.
Índice RAG	Citas apuntan a documentos con IDs distintos.	Mantener alias o mapa de documentos.
Prompt template	Cambia el formato esperado por evaluadores.	Versionar plantilla y scorecard.
Modelo	Cambia tokenizer, límites o tool calling.	Registrar <code>model_id</code> , <code>tokenizer_id</code> y límites por release.
Base de datos	Candidate escribe datos que baseline no entiende.	Usar migración expand-contract.

La regla expand-contract es sencilla:

- 1. Expandir: añadir lo nuevo sin retirar lo antiguo.
- 2. Migrar: mover tráfico y consumidores progresivamente.
- 3. Contraer: quitar lo antiguo cuando ya no se usa y hay evidencia.

Checklist mínimo antes de canary:

PREGUNTA	RESPUESTA ESPERADA
¿Baseline puede leer lo que candidate escribe?	Sí, o candidate no escribe todavía.
¿Candidate puede leer datos antiguos?	Sí.
¿El contrato JSON tiene versión?	Sí, <code>schema_version</code> o equivalente.
¿Los dashboards separan baseline y candidate?	Sí, por <code>variant</code> y <code>release_id</code> .
¿El rollback sabe qué hacer con datos nuevos?	Sí, ignorar, transformar o bloquear.

Asignación estable por cohorte

Para repartir tráfico no basta con llamar a `random()`. Necesitamos estabilidad. Si una conversación está en candidate, debe seguir en candidate mientras dure la prueba. Si un tenant entra en baseline, no debería saltar de variante por cada request.

Una forma común es usar hash determinista:

$$b(c, f) = \frac{H(c||f) \bmod 10000}{100}$$

$$variant(c) = \begin{cases} candidate & \text{si } b(c, f) < w \\ baseline & \text{si } b(c, f) \geq w \end{cases}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
c	Clave de cohorte.	<code>tenant_42</code> , <code>user_91</code> , <code>conversation_abc</code> .
f	Nombre de la bandera o release.	<code>support_rag_v19</code> .
H	Función hash estable.	SHA-256, MurmurHash, xxHash.
$b(c, f)$	Bucket entre 0 y 100.	3,71.
w	Peso de tráfico para candidate.	5%.

Si $w = 5$ y el bucket es 3,71, ese contexto entra en candidate. Si mañana subimos a 25%, entrarán contextos nuevos, pero quienes ya estaban en candidate seguirán ahí. Esa propiedad hace que el rollout sea más depurable.

Qué medir en cada paso

El canary de una app clásica suele mirar errores HTTP, latencia, CPU y quizá conversión. En IA necesitamos más señales:

SEÑAL	QUÉ PREGUNTA RESPONDE
<code>contract_fail_rate</code>	¿La salida sigue cumpliendo schema, tipos y campos?
<code>quality_proxy</code>	¿La calidad online o muestreada no cae?
<code>acceptance_rate</code>	¿La salida se acepta, se usa o evita revisión?
<code>fallback_rate</code>	¿El sistema cae más en rutas alternativas?
<code>cost_p95_eur</code>	¿El coste por run aceptada se mantiene?
<code>latency_p95_ms</code>	¿La experiencia interactiva sigue dentro del SLO?
<code>timeout_rate</code>	¿Suben timeouts o cortes de generación?
<code>route_mix_delta</code>	¿El router deriva demasiado a rutas caras o lentas?
<code>trace_completeness</code>	¿La versión candidata deja trazas suficientes?
<code>manual_review_rate</code>	¿Aumenta la necesidad de revisión?

Estas señales no deberían vivir en un informe manual. Deben salir de métricas y trazas filtrables por `release_id`, `variant`, `rollout_step`, `task` y `tenant`. OpenTelemetry define las trazas como una forma de seguir el camino de una petición mediante spans relacionados.⁶ Sus convenciones para GenAI añaden atributos propios de sistemas generativos.⁷

Para alertar durante canary, no basta con “parece que sube algo”. El SRE Workbook de Google recomienda razonar con SLOs y presupuesto de error.⁸ El libro de SRE también recuerda que proteger un servicio bajo carga implica decidir qué trabajo aceptar, retrasar o rechazar.⁹

Consultas observables reales

Decir “mide latencia y coste” es insuficiente. Un equipo necesita consultas reproducibles. Prometheus define PromQL como un lenguaje funcional para seleccionar y agregar series temporales en tiempo real.¹⁰ Con etiquetas como `release_id`, `variant`, `task` y `tenant_tier`, puedes preguntar cosas útiles.

Tasa de fallos de contrato por variante:

```
sum(rate(ai_contract_fail_total{release_id="support-rag@1.9.0-rc1"}[10m])) by (variant)
/
sum(rate(ai_run_total{release_id="support-rag@1.9.0-rc1"}[10m])) by (variant)
```

Latencia p95 de candidate:

```
histogram_quantile(
  0.95,
  sum(rate(ai_run_latency_seconds_bucket{
    release_id="support-rag@1.9.0-rc1",
    variant="candidate"
  }[10m])) by (le, task)
)
```

Coste p95 por tarea:

```
histogram_quantile(
  0.95,
  sum(rate(ai_run_cost_eur_bucket{
    release_id="support-rag@1.9.0-rc1"
  }[30m])) by (le, task, variant)
)
```

Ratio de fallback:

```
sum(rate(ai_fallback_total{release_id="support-rag@1.9.0-rc1"}[15m])) by (variant)
/
sum(rate(ai_run_total{release_id="support-rag@1.9.0-rc1"}[15m])) by (variant)
```

Si además guardas eventos analíticos en una tabla, la pregunta de negocio se puede revisar con SQL:

```
select
  variant,
  task,
  count(*) as runs,
  avg(case when accepted_by_user then 1 else 0 end) as acceptance_rate,
  avg(cost_eur) as avg_cost_eur,
  percentile_cont(0.95) within group (order by latency_ms) as latency_p95_ms
from ai_release_runs
where release_id = 'support-rag@1.9.0-rc1'
  and created_at >= now() - interval '2 hours'
group by variant, task
order by task, variant;
```

La señal importante para un ingeniero: no basta con instrumentar. Hay que diseñar nombres, etiquetas y cardinalidad para que las consultas salgan limpias cuando la release esté viva.

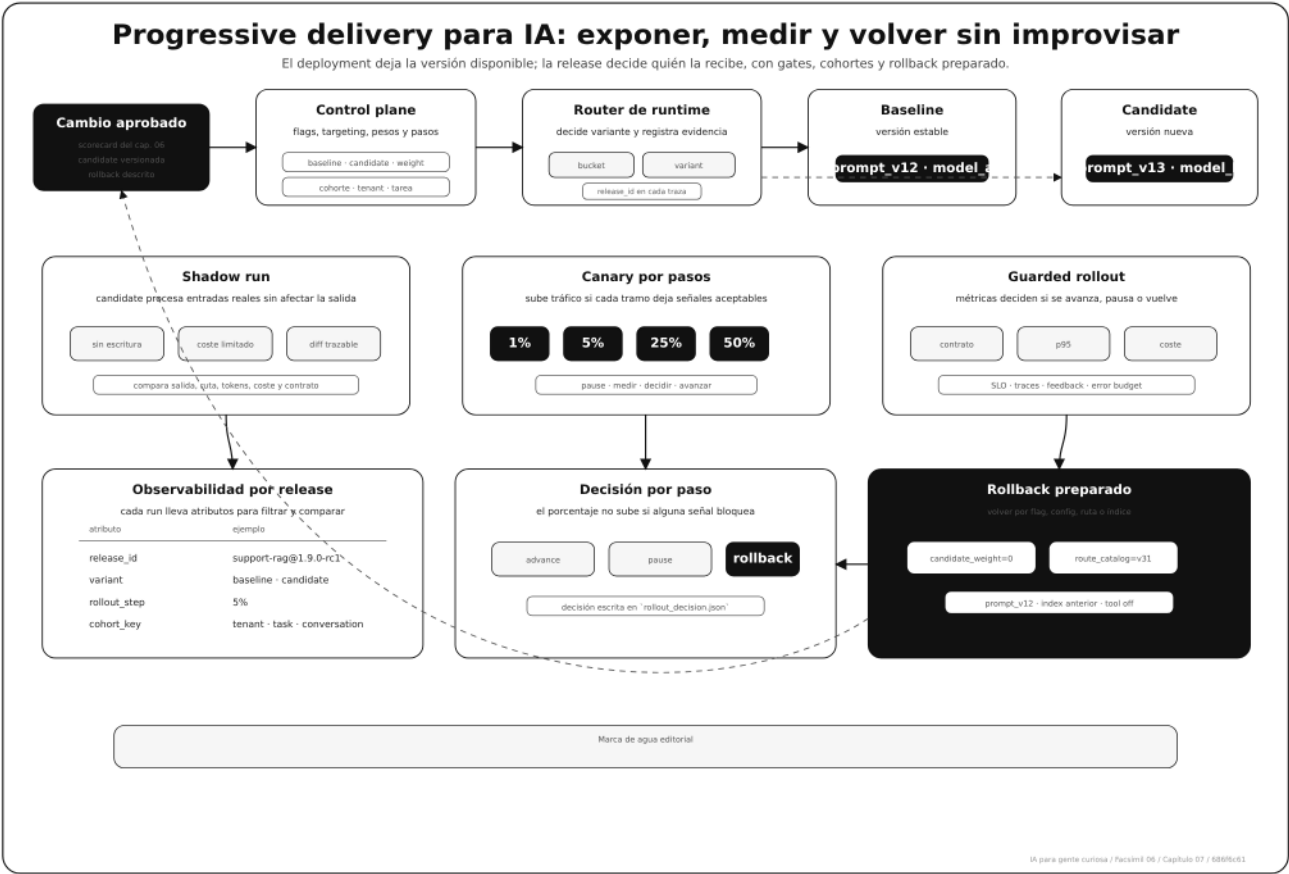
El gate de avance de cada paso es una función indicador: solo se sube al siguiente porcentaje de tráfico si se cumplen todas las condiciones:

$$A_s = 1[K_s \leq K_{max} \wedge L_{95,s} \leq L_{max} \wedge C_{95,s} \leq C_{max} \wedge Q_s \geq Q_{min} \wedge T_s \geq T_{min}]$$

SÍMBOLO	SIGNIFICADO
A_s	Decisión de avanzar en el paso s .
K_s	Tasa de fallos de contrato en el paso.
K_{max}	Máximo tolerado de fallos de contrato.
$L_{95,s}$	Latencia p95 observada.
L_{max}	Límite p95 del SLO.
$C_{95,s}$	Coste p95 por run aceptada.
C_{max}	Presupuesto máximo permitido.
Q_s	Calidad online, evaluación muestreada o proxy aceptado.
Q_{min}	Calidad mínima para avanzar.
T_s	Proporción de trazas completas.
T_{min}	Mínimo de trazabilidad aceptable.

Si $A_s = 1$, se puede avanzar al siguiente porcentaje. Si $A_s = 0$, no se avanza. Dependiendo del motivo, se pausa, se baja porcentaje o se vuelve a baseline.

Anatomía visual de un rollout progresivo de IA



Shadow run sin efectos persistentes

Shadow run significa que la candidata recibe entradas reales, pero su salida no se entrega al usuario. Esto permite comparar sin cambiar la experiencia principal. En IA hay que diseñarlo con cuidado.

Reglas para shadow:

REGLA	MOTIVO
No ejecutar tools con escritura.	Evita duplicar tickets, correos, pagos, publicaciones o cambios de estado.
Usar dry-run cuando exista.	Permite validar argumentos sin tocar sistemas externos.
Marcar cada traza como <code>shadow=true</code> .	Facilita filtrar métricas y coste.
Limitar coste por día.	Shadow puede duplicar llamadas de modelo.
No mezclar salida sombra con feedback de usuario.	El usuario no vio esa salida; no puede aceptarla ni rechazarla.
Comparar por caso pareado.	La misma entrada debe tener baseline y candidate.

La salida sombra sí sirve para medir:

COMPARACIÓN	QUÉ PUEDE REVELAR
Respuesta final	Candidate responde más largo, más corto, más caro o menos grounded.
Contrato	Candidate rompe JSON o cambia campos.
Routing	Candidate elige rutas más caras o lentas.
Tools	Candidate intenta una tool que baseline no necesitaba.
Tokens	Candidate consume más contexto o salida.
Latencia	Candidate empeora prefill, decode o validación.

Shadow no prueba aceptación real. Prueba compatibilidad con realidad. Es una fase de ingeniería, no una encuesta.

Canary para IA: qué segmentos elegir

El 1% global puede sonar prudente, pero no siempre enseña lo correcto. Si tu producto tiene tareas muy distintas, el canary debe segmentarse.

SEGMENTO	CUÁNDO USARLO	EJEMPLO
Por tarea	Cada tarea tiene riesgo y coste distintos.	<code>support_summary</code> , <code>policy_qa</code> , <code>json_extract</code> .
Por tenant	Los documentos o normas cambian por cliente.	3 tenants internos antes de externos.
Por idioma	La calidad puede variar mucho.	Español primero si el dataset está mejor cubierto.
Por canal	No es igual chat interactivo que batch.	Canary en batch antes que en chat.
Por contrato	JSON estricto merece gates más duros.	Activar solo en respuestas no estructuradas al principio.
Por criticidad	Tareas de bajo impacto primero.	Resumen interno antes que acción externa.

Un buen canary no busca la muestra más cómoda. Busca la muestra más informativa sin ampliar demasiado el impacto.

Tamaño mínimo de muestra y duración

Una release progresiva necesita suficientes observaciones para no decidir por ruido. El número exacto depende de la varianza, la criticidad y el volumen, pero conviene declarar mínimos antes de empezar.

FACTOR	QUÉ CAMBIA
Tráfico bajo	Necesitas más tiempo para llegar a un mínimo útil.
Salida muy variable	Necesitas más casos o segmentar por tarea.
Tarea crítica	Necesitas gates más estrictos y porcentaje más pequeño.
Coste alto	Shadow y canary deben tener presupuesto diario.
Contrato estricto	Un fallo puede pesar más que una media de calidad.
Tenants heterogéneos	El 1% global puede no representar a nadie.

Una pauta razonable para empezar:

PASO	MÍNIMO ORIENTATIVO	DURACIÓN MÍNIMA	QUÉ MIRARÍA ANTES DE AVANZAR
Shadow	100-300 runs pareadas.	1 ciclo de carga real.	Diffs, contrato, coste y rutas.
Canary 1%	200-500 runs.	30-120 min según tráfico.	Contrato, p95, fallback y trazas.
Canary 5%	500-1500 runs.	1-4 h.	Segmentos por tarea y tenant.
Canary 25%	1000-5000 runs.	Medio día o 1 ciclo de uso.	Presupuesto de error y coste p95.
Canary 50%	Depende del negocio.	1 ciclo relevante.	Comparación estable y sin regresiones nuevas.

Esto no sustituye estadística formal. Es una forma de no fingir precisión: si solo tienes 12 runs, una tasa del 0% no significa “cero problemas”; significa “no hemos mirado bastante”.

Cuándo parar el canary: el test secuencial de Wald. No hace falta fijar de antemano cuántas peticiones observar. El test secuencial de la razón de probabilidad acumula evidencia caso a caso y se detiene en cuanto cruza un umbral, lo que suele necesitar muchas menos muestras que un test de tamaño fijo.¹¹

$$\Lambda_n = \prod_{i=1}^n \frac{p_1(x_i)}{p_0(x_i)}, \quad B = \frac{\beta}{1-\alpha} < \Lambda_n < \frac{1-\beta}{\alpha} = A$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
Λ_n	Razón de verosimilitud acumulada tras n casos.	Sube si el canary parece malo.
p_1, p_0	Probabilidad del dato bajo «peor» y bajo «igual».	Modelos de error de cada hipótesis.
α, β	Errores tolerados de tipo I y II.	0,05 y 0,10.
A, B	Umbrales de decisión.	Parar arriba (malo) o abajo (bueno).

En palabras: mientras la evidencia esté en la zona dudosa, sigues observando; en cuanto cruza A cortas el canary, y en cuanto cruza B lo promueves. Esto es justo lo que quieres en un rollout, que un canary claramente malo se detenga pronto, sin gastar más usuarios de los necesarios para estar seguro.

¿Es el canary peor que la versión estable? Contraste de dos proporciones. La comparación natural de un canary es contra el grupo de control que sigue en la versión vieja. El contraste de dos proporciones dice si la diferencia de tasa de error entre ambos grupos es real o cabe en el azar del muestreo.¹²

$$z = \frac{\hat{p}_c - \hat{p}_v}{\hat{p}(1 - \hat{p}) \left(\frac{1}{n_c} + \frac{1}{n_v} \right)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\hat{p}_c, \hat{p}_v$	Tasa de error del canary y de la versión vieja.	0,04 y 0,03.
n_c, n_v	Tamaño de cada grupo.	2.000 y 18.000.
$\hat{p}$	Tasa combinada de ambos grupos.	0,031
z	Estadístico; \$	z

En palabras: el canary y el control comparten el mismo tráfico y el mismo momento, así que comparar sus tasas aísla el efecto del cambio mejor que comparar contra el histórico. Si $|z|$ no supera 1,96, la «subida de errores» del canary puede ser ruido; si lo supera, es señal de rollback.

¿A qué velocidad gasto el margen? La tasa de consumo del presupuesto de error. Un canary puede no fallar el contraste y aun así estar quemando el presupuesto de error demasiado rápido. La tasa de consumo (burn rate) mide a qué ritmo se agota el margen del SLO, y permite alertar antes de quedarte sin él.¹³

$$b = \frac{1 - \text{SLI}}{1 - \text{SLO}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
SLI	Indicador medido en la ventana actual.	0,94
SLO	Objetivo comprometido.	0,99
b	Tasa de consumo del presupuesto.	0,06/0,01 = 6.

En palabras: una tasa de 1 gasta el presupuesto justo a tiempo; una tasa de 6 lo agota seis veces más rápido de lo sostenible. Alertar por tasa de consumo, combinando una ventana corta y una larga, dispara el rollback cuando el daño se acumula deprisa pero ignora picos

breves e inofensivos. Es la regla que evita tanto el rollback histórico como el tardío.

¿Se ha salido la métrica de lo normal? El gráfico de control de Shewhart. En sombra comparas una métrica nueva contra su comportamiento normal. El gráfico de control separa la variación natural de la anómala con una regla simple y centenaria: marca como fuera de control lo que se aleja más de tres desviaciones de la media.¹⁴

$$LCS, LCI = \mu \pm 3 \sigma$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
μ	Media histórica de la métrica en sombra.	1,8 s de latencia.
σ	Desviación típica histórica.	0,3 s.
LCS, LCI	Límites de control superior e inferior.	0,9 s y 2,7 s.

En palabras: mientras la métrica baile entre los límites, es ruido normal y no hay que tocar nada; si los cruza, ha pasado algo que merece mirarse. Esto evita el error más común al vigilar una sombra, reaccionar a cada subida menor, y a la vez no deja pasar las desviaciones que sí importan.

Rollback no siempre es volver código atrás

En IA, rollback tiene varias capas. A veces no necesitas revertir todo el deployment. Basta con cambiar una bandera o restaurar una configuración.

CAPA	CÓMO VUELVES
Prompt	<code>prompt_version=prompt_v12</code> .
Modelo	<code>model_id=model_a</code> o ruta anterior.
RAG	<code>index_version=rag_index_2026_05</code> .
Router	<code>route_catalog=route_catalog@31</code> .
Tool	<code>tool_enabled=false</code> o <code>mode=read_only</code> .
Contrato	<code>response_schema=schema_v4</code> .
Runtime	Enviar tráfico al pool anterior.
Código	Revertir deployment o cambiar imagen.

La decisión entre rollback y roll forward depende de la causa:

SITUACIÓN	MEJOR OPCIÓN
Cambio de prompt rompe contrato.	Rollback de prompt o schema gate.
Nuevo índice RAG trae documentos incorrectos.	Rollback de índice.
Runtime nuevo tiene p95 malo.	Volver al pool anterior.
Falla una condición menor y hay fix claro.	Roll forward con parche y canary nuevo.
No sabemos la causa.	Bajar a baseline, preservar trazas y analizar.

La frase importante: **rollback no es castigo; es una capacidad de diseño.**

Kill switch: parar sin reunión

Un kill switch es una decisión operativa preparada para detener una capacidad sin desplegar código. En IA generativa puede apagar una tool, bajar una variante a 0%, forzar modelo baseline o cambiar a modo solo lectura.

CAMPO	PARA QUÉ SIRVE
<code>enabled</code>	Permite cortar una capacidad completa.
<code>candidate_weight</code>	Baja exposición sin tocar deployment.
<code>read_only</code>	Mantiene lectura y bloquea escritura.
<code>fallback_model</code>	Fuerza proveedor o modelo estable.
<code>max_cost_eur_per_run</code>	Corta rutas caras antes de agotar presupuesto.
<code>reason</code>	Deja trazabilidad de la decisión.
<code>expires_at</code>	Evita que un apagado temporal se quede olvidado.

Plantilla mínima:

```
release_id: support-rag@1.9.0-rc1
controls:
  enabled: true
  candidate_weight: 5
  read_only: false
  fallback_model: model_a
  max_cost_eur_per_run: 0.04
  kill_switch:
    enabled: false
    reason: null
    owner: ai-platform
    expires_at: null
```

Ensayo obligatorio antes del canary:

1. Activar `kill_switch.enabled=true` en staging.
2. Comprobar que todo tráfico va a baseline.
3. Comprobar que no se ejecutan tools de escritura.
4. Ver en trazas `kill_switch=true`.
5. Volver a activar candidate y verificar que el sistema recupera el comportamiento esperado.

Lifecycle de flags: limpiar también es operar

Los flags son útiles, pero también crean deuda si se quedan para siempre. LaunchDarkly documenta un ciclo de vida de flags con estados como live, ready for code removal, ready to archive, archived, deprecated y deleted, y recomienda planificar la retirada durante la creación del flag.¹⁵

Un flag de IA debería nacer con metadatos:

METADATO	EJEMPLO	MOTIVO
owner	ai-platform	Alguien responde por la bandera.
created_at	2026-05-28	Permite detectar flags antiguos.
expected_removal	2026-06-15	La retirada se planifica desde el inicio.
permanent	false	Distingue rollout temporal de interruptor permanente.
release_id	support-rag@1.9.0-rc1	Une flag, trazas, scorecard y decisión.
cleanup_issue	OPS-1842	La limpieza entra en el backlog real.

Estados recomendados:

ESTADO	QUÉ SIGNIFICA	ACCIÓN
planned	Está definido, pero no se evalúa todavía.	Revisar contrato y rollback.
shadow	Candidate se ejecuta sin salida visible.	Medir coste y compatibilidad.
canary	Candidate recibe parte del tráfico.	Vigilar gates.
launched	Candidate ya es comportamiento por defecto.	Abrir tarea de retirada si era temporal.
cleanup	El flag ya no decide nada útil.	Quitar código muerto y archivar.
permanent_control	Se conserva como interruptor.	Documentar quién puede cambiarlo.

La disciplina no termina cuando el rollout llega al 100%. Termina cuando el código, los dashboards y la documentación dejan de arrastrar caminos que nadie usa.

En el día a día

Los cambios progresivos aparecen la primera vez que algo que pasó todas las pruebas falla igualmente en producción, porque la realidad trae entradas, tráfico y casos que ningún dataset cubre del todo. La respuesta operable no es «desplegar y rezar», sino soltar el cambio por etapas: primero en sombra (procesa tráfico real pero su salida no se usa, solo se compara), luego en canary (un porcentaje pequeño de usuarios reales), y solo después al cien por cien. Cada etapa es una pregunta con menos riesgo: ¿se comporta igual que la versión vieja con tráfico real antes de que un solo usuario lo note?

La otra cara es el rollback. En un prototipo, volver atrás es `git revert` y a correr; en un sistema de IA, el comportamiento depende de modelo, prompt, política, dataset y retrieval, así que «volver» significa restaurar una combinación concreta y probada de todos ellos. El día que algo se tuerce en producción, la diferencia entre tener un rollback de un clic y reconstruir a mano qué versión funcionaba es la diferencia entre cinco minutos y una tarde con el sistema degradado.

Por qué debería importarte

Porque desplegar de golpe un sistema no determinista es una de las formas más caras de aprender. Los cambios progresivos te dan lo que ninguna batería de tests puede dar del todo: evidencia con tráfico real antes de comprometerte, y una salida de emergencia probada para cuando la evidencia sea mala. No eliminan el riesgo, lo acotan a una fracción de usuarios y a una ventana de tiempo corta.

Y hay una promesa que solo puedes hacer si esto está montado: que un cambio malo no se convierte en una incidencia grave. Con sombra, canary y rollback, lo peor que pasa es que un canary se apague antes de extenderse; sin ellos, lo peor que pasa es un fallo total en producción que tardas en diagnosticar y más aún en revertir. Esa diferencia es, casi siempre, la que separa un equipo que duerme tranquilo de uno que no.

Dónde solía tropezar yo

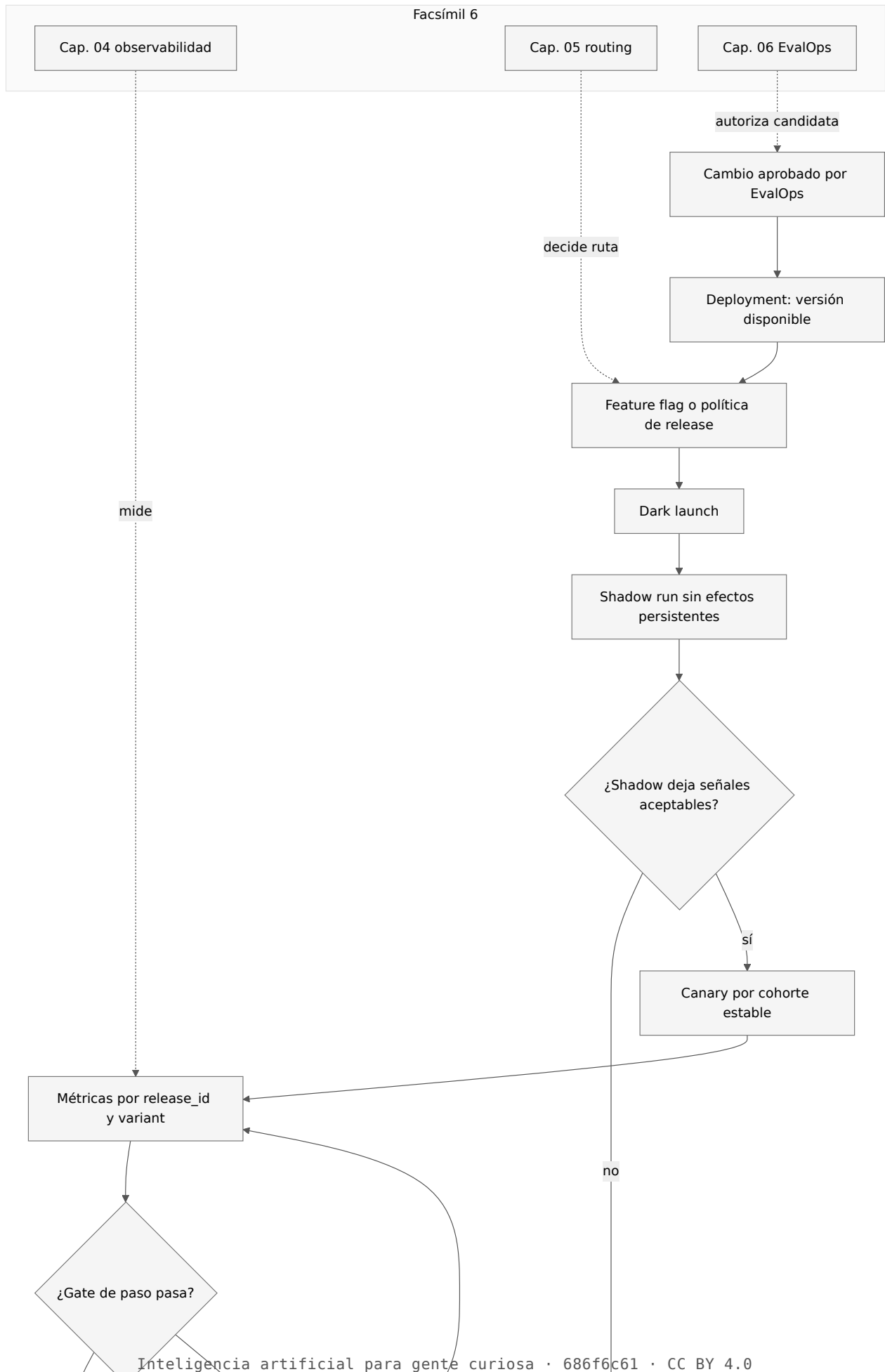
Me costó entender que canary no es una fase decorativa. Si no hay métricas, cohortes, gates y rollback, solo has cambiado el tamaño del salto.

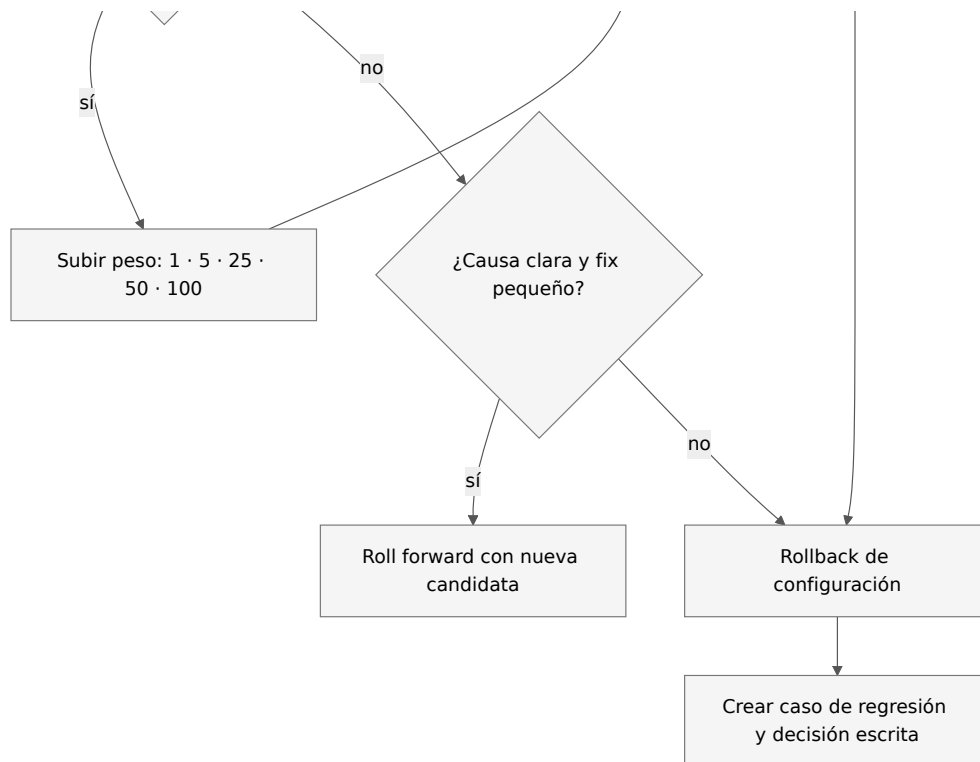
TROPIEZO	ANTÍDOTO
Confundir deployment y release.	Preguntar: ¿la versión está disponible o expuesta?
Usar <code>random()</code> por request.	Asignar por hash estable de cohorte.
Hacer shadow con tools de escritura.	Dry-run, solo lectura o bloqueo explícito.
Mirar solo errores técnicos.	Añadir contrato, coste, calidad, fallback y trazas.
No probar rollback.	Ensayar vuelta antes de subir tráfico.

La frase que me habría ahorrado muchos sustos: **un rollout no es una escalera automática; es una serie de decisiones con evidencia.**

Cómo encaja todo

Facsímil 6





Relación con otros capítulos

Este capítulo no sustituye a EvalOps. Lo continúa.

CAPÍTULO	QUÉ APORTA AQUÍ
<u>F6 · Capítulo 01</u>	Manifest, versión, release gate y rollback como idea de sistema operable.
<u>F6 · Capítulo 04</u>	SLIs, SLOs, métricas, trazas y alertas para decidir.
<u>F6 · Capítulo 05</u>	Router, rutas, presupuestos, shadow routing y canary de políticas.
<u>F6 · Capítulo 06</u>	Datasets, gates y scorecard antes de exponer tráfico.
<u>F5 · Capítulo 08</u>	Revisión humana y permisos cuando una acción requiere aprobación.

La cadena completa queda así: EvalOps decide que la candidata merece exposición; progressive delivery decide cómo se expone; observabilidad decide si sigue avanzando; rollback conserva una salida de vuelta.

Para entenderlo

Tres escenas sencillas:

SITUACIÓN	MALA DECISIÓN	DECISIÓN PROGRESIVA
Nuevo prompt más claro	100% de tráfico el lunes.	Shadow, 1%, revisar contrato y subir si p95/coste aguantan.
Nuevo índice RAG	Sustituir índice antiguo.	Candidate usa índice nuevo en shadow y compara fuentes recuperadas.
Nuevo modelo barato	Cambiar proveedor por defecto.	Canary por tareas de baja criticidad, midiendo calidad y fallbacks.

No es miedo al cambio. Es respeto por la operación.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN OPERATIVA	ERROR TÍPICO
Deployment	La versión está instalada o disponible.	Creer que ya está expuesta al usuario.
Release	La versión está recibiendo tráfico o afectando decisiones.	Publicarla al 100% sin gates intermedios.
Dark launch	La versión se despliega sin uso visible.	Confundir disponibilidad con validación.
Shadow run	La candidata procesa entradas reales sin producir efectos persistentes.	Dejar que duplique emails, tickets o escrituras.
Canary	Exposición pequeña y medible a una cohorte estable.	Usar tráfico aleatorio por request y mezclar señales.
Cohorte estable	Grupo asignado por hash de tenant, usuario, conversación o tarea.	Cambiar variante en cada llamada y romper comparaciones.
Kill switch	Control para apagar una capacidad sin redespigar.	Tener que tocar código durante una incidencia.
Rollback	Vuelta a una versión conocida y verificada.	Pensar que solo es revertir código, olvidando prompt, índice o ruta.
Roll forward	Nueva candidata que corrige el problema sin volver atrás.	Usarlo sin evidencia suficiente porque “parece pequeño”.
Blast radius	Porción de usuarios, tenants o tareas expuesta al cambio.	Medir solo porcentaje global y no criticidad.

Antes de pasar página

Comprueba que puedes responder:

1. ¿Cuál es la diferencia entre deployment y release?
2. ¿Por qué un feature flag ayuda a publicar cambios de IA?
3. ¿Qué debe evitar un shadow run para no duplicar efectos persistentes?
4. ¿Por qué conviene asignar canary por cohorte estable?
5. ¿Qué señales mirarías en un canary de RAG?

6. ¿Qué capas puede tocar un rollback en IA además del código?
7. ¿Qué archivo entregarías para demostrar que el rollback está preparado?

En resumen

IDEA	PARA LLEVARTE
Deployment y release no son lo mismo.	Puedes tener una versión desplegada pero no expuesta.
Shadow reduce incertidumbre.	Compara con entradas reales sin afectar la salida visible.
Canary debe tener gates.	Cada porcentaje necesita métricas, duración, dueño y criterio de avance.
Rollback es diseño.	Volver por flag, prompt, modelo, índice o ruta debe estar probado antes.

Para saber más

- Argo Project. *Canary Deployment Strategy*. <https://argoproj.github.io/argo-rollouts/features/canary/>
- Google Cloud. *Use a canary deployment strategy*. <https://cloud.google.com/deploy/docs/deployment-strategies/canary>
- Kubernetes. *Deployments*. <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>
- LaunchDarkly. *Reducing Technical Debt from Feature Flags*. <https://launchdarkly.com/docs/guides/flags/technical-debt>
- LaunchDarkly. *Releasing features with LaunchDarkly*. <https://launchdarkly.com/docs/home/releases/releasing>
- OpenTelemetry. *Semantic Conventions for Generative AI Systems*. <https://opentelemetry.io/docs/specs/semconv/gen-ai/>
- OpenTelemetry. *Traces*. <https://opentelemetry.io/docs/concepts/signals/traces/>
- OpenFeature. *Introduction*. <https://openfeature.dev/docs/reference/intro/>
- Prometheus. *Querying Basics*. <https://prometheus.io/docs/prometheus/latest/querying/basics/>
- Wilkinson, J. *Alerting on SLOs*. <https://sre.google/workbook/alerting-on-slos/>

Notas

1. OpenFeature. (2026). *Introduction*. <https://openfeature.dev/docs/reference/intro/>. Consultado el 28 de mayo de 2026. ↵
2. Kubernetes. (2026). *Deployments*. <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>. Consultado el 28 de mayo de 2026. ↵
3. Argo Project. (2026). *Canary Deployment Strategy*. <https://argoproj.github.io/argo-rollouts/features/canary/>. Consultado el 28 de mayo de 2026. ↵
4. LaunchDarkly. (2026). *Releasing features with LaunchDarkly*. <https://launchdarkly.com/docs/home/releases/releasing>. Consultado el 28 de mayo de 2026. ↵
5. Google Cloud. (2026). *Use a canary deployment strategy*. <https://cloud.google.com/deploy/docs/deployment-strategies/canary>. Consultado el 28 de mayo de 2026. ↵
6. OpenTelemetry. (2026). *Traces*. <https://opentelemetry.io/docs/concepts/signals/traces/>. Consultado el 27 de mayo de 2026. ↵
7. OpenTelemetry. (2026). *Semantic Conventions for Generative AI Systems*. <https://opentelemetry.io/docs/specs/semconv/gen-ai/>. Consultado el 27 de mayo de 2026. ↵
8. Wilkinson, J. (2018). *Alerting on SLOs*. En B. Beyer, N. R. Murphy, D. Rensin, K. Kawahara y S. Thorne (eds.), *The Site Reliability Workbook*. <https://sre.google/workbook/alerting-on-slos/>. Consultado el 27 de mayo de 2026. ↵
9. Beyer, B., Jones, C., Petoff, J. y Murphy, N. R. (2016). *Handling Overload*. En *Site Reliability Engineering*. <https://sre.google/sre-book/handling-overload/>. Consultado el 27 de mayo de 2026. ↵
10. Prometheus. (2026). *Querying Basics*. <https://prometheus.io/docs/prometheus/latest/querying/basics/>. Consultado el 28 de mayo de 2026. ↵
1. Wald, A. (1945). Sequential Tests of Statistical Hypotheses. *The Annals of Mathematical Statistics*, 16(2), 117-186. <https://doi.org/10.1214/aoms/1177731118>. Introduce el test secuencial de la razón de probabilidad (SPRT). ↵
2. El contraste de dos proporciones mediante el estadístico z es un resultado estándar de inferencia; véase Agresti, A. (2013). *Categorical Data Analysis* (3.ª ed.). Wiley. ↵
3. Beyer, B., Murphy, N. R., Rensin, D. K., Kawahara, K., & Thorne, S. (Eds.). (2018). *The Site Reliability Workbook*. O'Reilly. Define la alerta por tasa de consumo del presupuesto de error con ventanas múltiples. ↵

- .4. Shewhart, W. A. (1931). *Economic Control of Quality of Manufactured Product*. Van Nostrand.
Origen del gráfico de control y de los límites a tres sigma. ↩
- .5. LaunchDarkly. (2026). *Reducing Technical Debt from Feature Flags*.
<https://launchdarkly.com/docs/guides/flags/technical-debt>. Consultado el 28 de mayo de 2026.
↩

Capítulo 08: Handoffs operativos y revisión humana

Qué deberías poder hacer al terminar

En el capítulo 07 aprendimos a exponer cambios poco a poco. Ahora aparece una situación igual de importante: **qué ocurre cuando una run no debería seguir sola.**

Un sistema de IA real no vive solo en el modelo. Vive en colas, tickets, contratos, herramientas, permisos, revisiones, trazas y decisiones. A veces el sistema puede responder. A veces debe abstenerse. A veces debe preparar una acción, pero no ejecutarla. Y a veces debe entregar el caso a una persona con todo lo necesario para decidir sin reconstruir la historia desde cero.

Al terminar, deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Diseñar un handoff operativo.	Sabes qué campos mínimos debe entregar una run al pausarse.
Decidir cuándo pedir revisión.	Usas reglas por confianza, efecto, coste, contrato, evidencia y permisos.
Construir una cola de revisión.	Priorizas casos por impacto, antigüedad, SLA y falta de evidencia.
Diseñar una tarjeta de decisión.	La persona ve entrada, propuesta, fuentes, trazas, coste y acciones disponibles.
Reanudar una run.	Sabes continuar con <code>approve</code> , <code>reject</code> o <code>needs_more_info</code> sin perder estado.
Medir la cola.	Defines SLI, SLO, backlog, aging, tasa de acuerdo y coste humano.
Convertir revisión en aprendizaje.	Los casos revisados vuelven a evals, prompts, contratos y runbooks.

La idea central: **pedir revisión no es fracasar; fracasar es pedirla sin contexto, sin prioridad y sin reanudación clara.**

Cuando el sistema no debe seguir solo

Imagina un asistente interno que prepara respuestas para soporte universitario. La mayoría de preguntas son sencillas: horarios, plazos, enlaces, estado de una solicitud. Pero un día llega un caso con documentos incompletos, una norma que ha cambiado, un dato que contradice el expediente y una respuesta propuesta que podría cerrar el ticket.

El sistema puede tener una respuesta plausible. Eso no basta. La pregunta operativa es otra:

¿Tenemos evidencia suficiente para que esta acción siga automáticamente, o debemos entregar el caso a revisión?

Un handoff operativo evita dos extremos malos. El primero: automatizar de más y descubrir tarde que faltaba una condición. El segundo: mandar todo a revisión y convertir el sistema en una bandeja de espera. La ingeniería está en diseñar el punto medio.

Qué no es revisión humana

Revisión humana no es añadir un botón de “aprobar” al final de una pantalla. Si la persona no ve el motivo, los datos usados, la salida propuesta, las fuentes, los límites y el efecto de cada botón, está firmando a oscuras.

Tampoco es revisar todo. Si cada respuesta trivial pide intervención, la cola crece, el tiempo de respuesta empeora y la revisión pierde valor. La persona debe entrar donde aporta criterio: falta evidencia, hay efecto persistente, la confianza no alcanza el umbral, el contrato falla, el coste se sale del presupuesto o la acción afecta a un recurso sensible.

Y no es “que lo decida soporte” como cajón de sastre. Una cola sin SLO, prioridad, dueño y estado es deuda operativa. El sistema debe explicar qué necesita y qué pasará después de la decisión.

CONFUSIÓN	QUÉ FALTA
“Lo revisa una persona y ya está”	Falta evidencia estructurada, estado y reanudación.
“Todo lo dudoso va a revisión”	Falta priorización y criterios de entrada.
“La persona corrige a mano”	Falta convertir la corrección en dataset, regla o cambio de contrato.
“Aprobar es pulsar sí”	Falta mostrar efecto, alcance y alternativa.
“La cola es una bandeja”	Falta SLI, SLO, aging, backlog y dueño.

Qué sí es un handoff operativo

Un handoff operativo es una pausa con contrato. La run entrega un paquete de evidencia, queda en un estado recuperable y espera una decisión explícita.

Un handoff se compone de estos campos. No es una ecuación de la literatura, es una lista de ingeniería:

SÍMBOLO	SIGNIFICADO	EJEMPLO
x	Entrada original.	Pregunta del usuario y documentos consultados.
y	Salida propuesta.	Respuesta, JSON, diff o acción preparada.
a	Acción pendiente.	Enviar, publicar, actualizar, derivar, cerrar ticket.
e	Evidencia disponible.	Fuentes, trazas, validaciones, métricas, contrato.
r	Razón de revisión.	Baja confianza, efecto persistente, falta de fuente, coste alto.
q	Cola o persona responsable.	<code>soporte_n2</code> , <code>legal_ops</code> , <code>ai_platform</code> .
t	Tiempo objetivo de resolución.	30 minutos, 4 horas, 1 día laboral.
s	Estado serializable de la run.	<code>run_state</code> , <code>trace_id</code> , <code>resume_token</code> .

El sistema no debería decir solo “necesita revisión”. Debería decir: “necesita revisión por estas razones, con esta evidencia, antes de este tiempo, y estas son las decisiones posibles”.

Fecha de corte del estado del arte

Fecha de corte: 28 de mayo de 2026.

Fuentes consultadas: documentación oficial de OpenAI Agents SDK sobre human-in-the-loop y handoffs; documentación de Anthropic Claude Agent SDK sobre permisos, hooks y aprobaciones; documentación de Google ADK sobre callbacks y controles para agentes; LangGraph interrupts; OpenTelemetry Traces; NIST AI RMF; y trabajos académicos sobre ingeniería de ML, model cards y datasheets.

OpenAI Agents SDK describe un flujo donde una herramienta puede marcarse como pendiente de aprobación, la ejecución se pausa con interrupciones, el estado de la run se serializa y después se reanuda con aprobación o rechazo.¹ También documenta handoffs como mecanismo para transferir control entre agentes especializados.²

Anthropic documenta permisos, reglas de allow/deny, modos de permiso, hooks y callbacks de aprobación para controlar cuándo una herramienta se ejecuta, se bloquea o pide una decisión.³⁴⁵

Google ADK presenta callbacks para observar, personalizar y controlar el comportamiento de agentes, y recomienda tratar los agentes como sistemas con controles explícitos, evaluación y límites.⁶⁷ LangGraph usa interrupts para pausar grafos, pedir decisión humana y reanudar desde un checkpoint.⁸

Lo estable no es una API concreta. Lo estable es el patrón: detectar, pausar, empaquetar evidencia, decidir, reanudar, registrar y aprender.

Cuándo disparar revisión

Un buen sistema no pregunta a una persona por vergüenza. Pregunta porque una regla dice que ahí la revisión tiene valor.

DISPARADOR	SEÑAL CONCRETA	DECISIÓN TÍPICA
Baja confianza	<code>confidence < 0.72</code> .	Revisar antes de responder.
Contrato roto	JSON inválido, campo obligatorio ausente, enum inesperado.	Bloquear y pedir corrección.
Falta de evidencia	No hay fuente, cita o documento suficiente.	Pedir más información.
Fuentes en tensión	Dos documentos sostienen respuestas distintas.	Revisar con expediente completo.
Efecto persistente	Enviar, publicar, cerrar, modificar o borrar.	Aprobar antes de ejecutar.
Coste alto	La ruta supera presupuesto por tarea.	Revisar si merece usar ruta cara.
Permiso insuficiente	La tool requiere scope que la run no tiene.	Derivar a persona responsable.
Cola saturada	El sistema entraría en demora no aceptable.	Responder con estado claro o degradar.
Canary	Candidate toca casos nuevos.	Muestrear revisión para aprender.
Usuario lo pide	La persona quiere contacto humano.	Crear handoff explícito.

La regla práctica: cada disparador debe tener una salida concreta. Si una condición solo dice “mirar”, no es una política; es una nota suelta.

Fórmula sencilla de decisión

La decisión de revisar o automatizar es una comparación de coste esperado, el criterio de la teoría de la decisión sensible al coste: se revisa cuando el coste esperado de un fallo automático (su probabilidad por su impacto) supera el coste de revisar.⁹

$$R(x) = \begin{cases} \text{review} & \text{si } p_f(x) \cdot C_f(x) > C_h(x) + C_d(x) \\ \text{auto} & \text{si } p_f(x) \cdot C_f(x) \leq C_h(x) + C_d(x) \end{cases}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$R(x)$	Decisión para el caso x .	review o auto .
$p_f(x)$	Probabilidad estimada de fallo relevante.	0,18.
$C_f(x)$	Coste si el sistema se equivoca.	80 EUR equivalentes o impacto operativo alto.
$C_h(x)$	Coste de revisión humana.	6 minutos de una persona.
$C_d(x)$	Coste de demora.	Retraso sobre SLA o experiencia.

Ejemplo numérico:

VARIABLE	VALOR
$p_f(x)$	0,18
$C_f(x)$	80
$C_h(x)$	6
$C_d(x)$	3

Calculamos:

$$p_f(x) \cdot C_f(x) = 0,18 \cdot 80 = 14,4$$

$$C_h(x) + C_d(x) = 6 + 3 = 9$$

Como $14,4 > 9$, revisamos. No porque “dé miedo”, sino porque el coste esperado de seguir automáticamente supera el coste de parar y decidir mejor.

Esta fórmula no sustituye criterio profesional. Lo hace discutible. Si alguien quiere cambiar el umbral, tiene que hablar de probabilidades, costes, demoras y evidencia.

¿Coinciden los revisores? La kappa de Cohen. Si dos personas revisan los mismos casos y no se ponen de acuerdo, tu «verdad» no es fiable. El coeficiente kappa mide el acuerdo entre revisores descontando el que ocurriría por puro azar, que es lo que la simple coincidencia porcentual oculta.¹⁰

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
p_o	Acuerdo observado entre revisores.	0,90
p_e	Acuerdo esperable por azar.	0,60
κ	Acuerdo corregido por azar.	$(0,9 - 0,6)/(1 - 0,6) = 0,75$.

En palabras: dos revisores que coinciden el 90% pueden tener una kappa modesta si ese tema es fácil y el azar ya explicaba el 60%. Una kappa baja avisa de que la rúbrica es ambigua o los revisores entienden cosas distintas: antes de fiarte de las etiquetas humanas que alimentan tus evals, conviene medir si esas etiquetas son consistentes.

¿Cuándo molestar a un humano? Riesgo y cobertura. Mandar a revisión todo no escala; no mandar nada es temerario. La predicción selectiva formaliza el equilibrio: el sistema responde solo cuando su confianza supera un umbral y delega el resto, y se mide con dos cantidades, cuánto cubre y con qué error.¹¹

$$\text{cobertura} = \frac{1}{N} \sum_{i=1}^N 1[g(x_i) \geq \tau], \quad \text{riesgo} = \frac{\sum_i 1[g(x_i) \geq \tau] \ell_i}{\sum_i 1[g(x_i) \geq \tau]}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$g(x_i)$	Confianza del sistema en el caso i .	0,2 a 0,99.
τ	Umbral por encima del cual responde solo.	0,85
cobertura	Fracción que el sistema atiende sin humano.	0,70
riesgo	Tasa de error entre los casos atendidos.	0,02

En palabras: subir el umbral baja la cobertura pero también el error de lo que se responde solo. La curva riesgo-cobertura te deja elegir el punto de operación con datos, por ejemplo «automatizo el 70% con un 2% de error y delego el 30% dudoso», en vez de decidir el umbral por intuición.

¿Significa algo la confianza del modelo? La puntuación de Brier. La confianza solo sirve para decidir handoffs si está calibrada: si el modelo dice 0,9, debería acertar el 90% de esas veces. La puntuación de Brier mide esa calibración como el error cuadrático entre la probabilidad anunciada y lo que de verdad ocurre.¹²

$$\text{BS} = \frac{1}{N} \sum_{i=1}^N (f_i - o_i)^2$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
f_i	Confianza anunciada para el caso i .	0,9
o_i	Resultado real: 1 si acertó, 0 si no.	0 o 1.
BS	Puntuación; más baja es mejor.	0 perfecto, 0,25 es azar.

En palabras: un Brier bajo significa que la confianza es de fiar y se puede usar como umbral de handoff; uno alto significa que el «0,9» del modelo no quiere decir 90%, y entonces enrutar por esa confianza es enrutar por una ilusión. Calibrar antes de delegar es lo que evita mandar a revisión los casos equivocados.

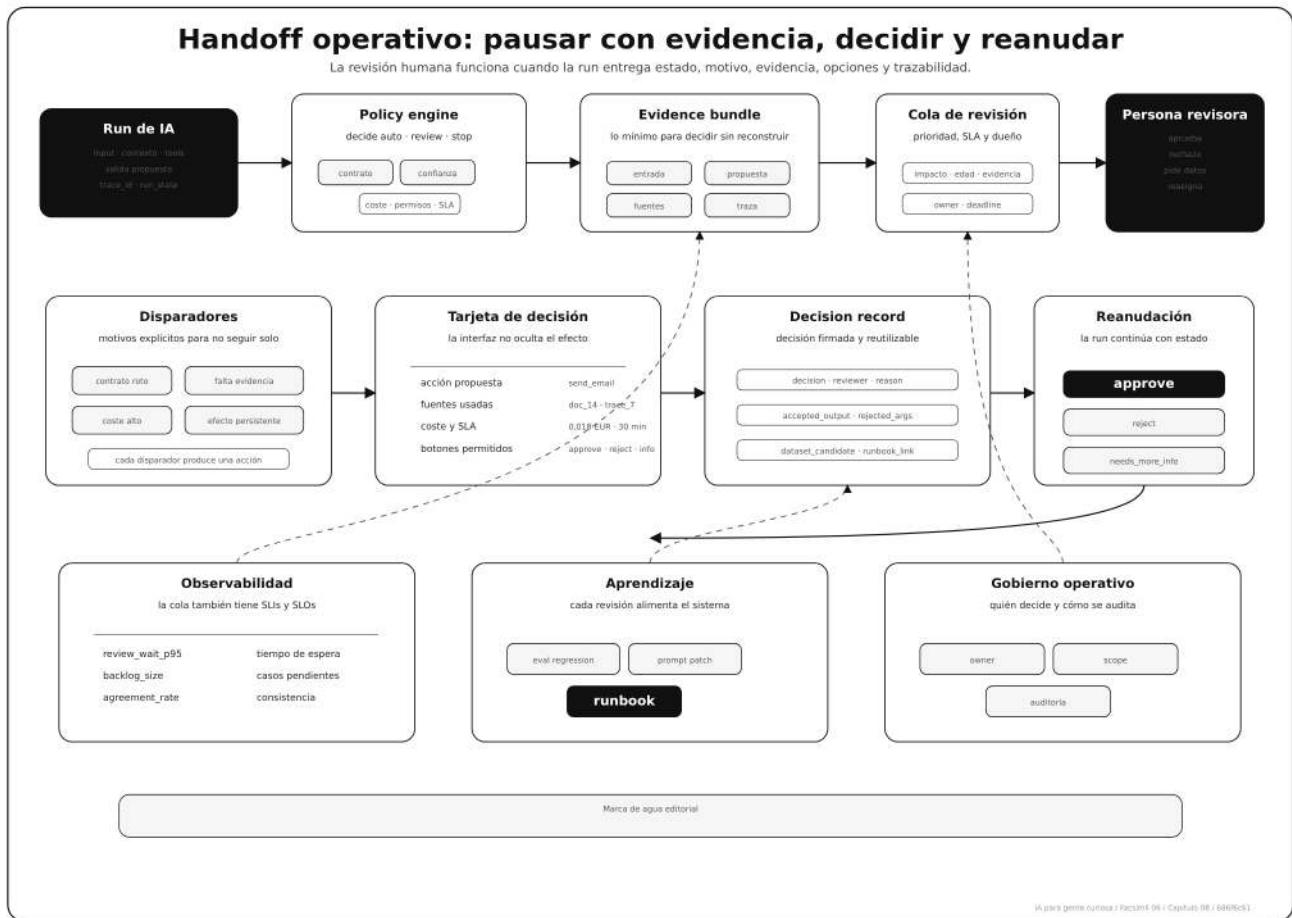
¿Se va a saturar la cola de revisión? La ley de Little. Una bandeja de revisión es una cola, y obedece la misma ley que las colas del runtime (la vimos en el capítulo 2): el trabajo acumulado es la tasa de entrada por el tiempo que cada caso pasa esperando.¹³

$$L = \lambda W$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
L	Casos esperando revisión en media.	60
λ	Casos que llegan a revisión por hora.	30 por hora.
W	Tiempo medio que un caso espera.	2 horas.

En palabras: si llegan 30 handoffs por hora y cada uno tarda 2 horas en revisarse, siempre hay unos 60 en cola. Si quieres que la espera baje a media hora con esa entrada, necesitas capacidad para que solo haya 15 acumulados, lo que fija cuántos revisores hacen falta. Sin esta cuenta, una cola de revisión sin caducidad ni dimensionado se convierte en el cuello de botella que vacía de sentido la supervisión humana.

Anatomía visual de un handoff operativo



El contrato mínimo del handoff

Un handoff serio debería poder serializarse. Si no puede guardarse, moverse de cola, auditarse y reanudarse, dependerá demasiado de la memoria de una persona.

Un contrato útil contiene:

CAMPO	MOTIVO	EJEMPLO
<code>request_id</code>	Identifica la revisión.	<code>rev_20260528_001</code> .
<code>run_id</code>	Une revisión con ejecución.	<code>run_7f31</code> .
<code>trace_id</code>	Permite abrir la traza completa.	<code>trace_abc</code> .
<code>task</code>	Agrupar métricas por caso de uso.	<code>support_reply</code> .
<code>proposed_action</code>	Dice qué se quiere hacer.	<code>send_email</code> .
<code>effect</code>	Describe impacto técnico.	<code>external_message</code> , <code>db_update</code> , <code>none</code> .
<code>reason_codes</code>	Explica por qué se revisa.	<code>low_confidence</code> , <code>missing_source</code> .
<code>evidence</code>	Resume pruebas disponibles.	Fuentes, validaciones, diffs, score.
<code>missing_evidence</code>	Dice qué falta.	<code>policy_doc</code> , <code>user_confirmation</code> .
<code>options</code>	Limita decisiones posibles.	<code>approve</code> , <code>reject</code> , <code>needs_more_info</code> .
<code>deadline_at</code>	Hace visible el SLO.	<code>2026-05-28T15:30:00Z</code> .
<code>resume_token</code>	Permite continuar la run.	Token opaco o estado serializado.

Ejemplo:

```
{
  "request_id": "rev_20260528_001",
  "run_id": "run_7f31",
  "trace_id": "trace_abc",
  "task": "support_reply",
  "proposed_action": "send_email",
  "effect": "external_message",
  "confidence": 0.68,
  "cost_eur": 0.018,
  "reason_codes": ["low_confidence", "missing_source"],
  "evidence": {
    "draft": "Hemos revisado tu solicitud...",
    "sources": ["doc_matricula_2026"],
    "schema_valid": true,
    "policy_version": "handoff_policy@3"
  },
  "missing_evidence": ["expediente_actualizado"],
  "options": ["approve", "reject", "needs_more_info"],
  "deadline_at": "2026-05-28T15:30:00Z",
  "resume_token": "resume_run_7f31_step_12"
}
```

La persona no recibe “un caso raro”. Recibe un objeto de trabajo.

Máquina de estados del handoff

Para ingeniería, un handoff no debería ser un booleano `needs_review=true`. Debe tener una máquina de estados. Si no la tiene, aparecen dudas en producción: ¿se puede aprobar dos veces?, ¿qué pasa si caduca?, ¿quién puede reasignar?, ¿qué ocurre si la run ya se reanudó?

Una máquina mínima:

ESTADO	QUÉ SIGNIFICA	TRANSICIONES VÁLIDAS
running	La run sigue trabajando.	needs_review , completed , failed .
needs_review	La política exige revisión.	queued , cancelled .
queued	El caso está en cola con prioridad y deadline.	assigned , expired , cancelled .
assigned	Una persona o equipo lo tiene asignado.	in_review , reassigned , expired .
in_review	La tarjeta está abierta o en edición.	approved , approved_with_edits , rejected , needs_more_info .
needs_more_info	Falta evidencia concreta.	queued , cancelled .
approved	La acción puede ejecutarse como venía.	resuming .
approved_with_edits	La acción puede continuar con cambios humanos.	resuming .
rejected	La acción no debe ejecutarse.	resuming .
resuming	Un worker está aplicando la decisión.	closed , resume_failed .
resume_failed	La decisión existe, pero reanudar falló.	resuming , closed_manual .
expired	Se incumplió el tiempo objetivo sin decisión.	reassigned , cancelled , fallback_response .
closed	El caso terminó con traza y decision record.	Estado terminal.

Reglas que conviene escribir:

REGLA	POR QUÉ IMPORTA
Una revisión cerrada no se modifica; se crea una revisión nueva.	Conserva auditoría y evita reescribir historia.
<code>approved</code> no ejecuta directamente; pasa por <code>resuming</code> .	Permite idempotencia y control de efectos.
<code>expired</code> no decide por sí solo.	La caducidad activa una política: reasignar, degradar o responder con estado claro.
<code>needs_more_info</code> debe nombrar qué falta.	Evita que la run vuelva a la cola sin mejora.
Cada transición emite un evento.	La cola se puede auditar y medir.

Modelo de datos persistente

El contrato JSON sirve para transportar el caso. En producción hace falta persistencia. Un modelo relacional mínimo podría ser:

```

create table handoff_requests (
  request_id text primary key,
  run_id text not null,
  trace_id text not null,
  task text not null,
  queue_name text not null,
  state text not null,
  proposed_action text not null,
  effect text not null,
  confidence numeric not null,
  cost_eur numeric not null,
  reason_codes jsonb not null,
  evidence jsonb not null,
  missing_evidence jsonb not null,
  resume_token text not null,
  idempotency_key text not null unique,
  created_at timestamptz not null,
  deadline_at timestamptz not null,
  updated_at timestamptz not null
);

create table review_decisions (
  decision_id text primary key,
  request_id text not null references handoff_requests(request_id),
  decision_version integer not null,
  reviewer text not null,
  decision text not null,
  reason text not null,
  edited_output text,
  created_at timestamptz not null,
  unique (request_id, decision_version)
);

create table review_events (
  event_id text primary key,
  request_id text not null references handoff_requests(request_id),
  event_type text not null,
  actor text not null,
  payload jsonb not null,
  created_at timestamptz not null
);

create table review_assignments (
  assignment_id text primary key,
  request_id text not null references handoff_requests(request_id),
  queue_name text not null,
  assignee text,
  assigned_at timestamptz not null,
  released_at timestamptz
);

```

Consultas que un equipo debería poder hacer:

Inteligencia artificial para gente curiosa · 686f6c61 · CC BY 4.0

```
select state, count(*) as n
from handoff_requests
where created_at >= now() - interval '24 hours'
group by state
order by n desc;
```

```
select
  queue_name,
  percentile_cont(0.95) within group (order by extract(epoch from updated_at - created_at) /
60) as age_p95_minutes,
  avg(case when now() > deadline_at and state not in ('closed', 'cancelled') then 1 else 0
end) as active_breach_rate
from handoff_requests
where created_at >= now() - interval '7 days'
group by queue_name;
```

La base de datos no es burocracia. Es lo que permite reanudar después de un crash, medir la cola, auditar decisiones y aprender de patrones repetidos.

Idempotencia: aprobar dos veces no debe ejecutar dos veces

El punto más delicado para un ingeniero de IA no es pedir revisión. Es aplicar la decisión sin duplicar efectos.

Ejemplo de fallo realista:

1. Una persona aprueba `send_email`.
2. El worker ejecuta la tool.
3. El proceso cae antes de marcar `closed`.
4. El sistema reintenta.
5. Si no hay idempotencia, envía dos correos.

Campos que evitan ese problema:

CAMPO	QUÉ PROTEGE
<code>idempotency_key</code>	La misma acción aprobada no se ejecuta dos veces.
<code>effect_id</code>	Identifica el efecto externo creado: email, ticket, cambio, publicación.
<code>decision_version</code>	Evita aplicar una decisión antigua tras una corrección.
<code>resume_token</code>	Reanuda el punto correcto de la run.
<code>state_version</code>	Bloquea carreras entre workers.
<code>executed_at</code>	Marca cuándo se produjo el efecto.

Patrón recomendado:

1. Leer request y decisión dentro de una transacción.
2. Comprobar ``state=resuming`` y ``decision_version`` vigente.
3. Reservar ``idempotency_key``.
4. Ejecutar tool con esa clave.
5. Guardar ``effect_id``.
6. Marcar ``closed``.
7. Emitir evento ``handoff.closed``.

Si la tool externa soporta claves de idempotencia, úsala. Si no, crea tu propia tabla de efectos y no ejecutes dos veces la misma combinación (`request_id`, `decision_version`, `proposed_action`).

Colas, SLI y SLO de revisión

Una cola de revisión también se opera. Si no la medimos, solo veremos su dolor cuando alguien se queje.

SLI	QUÉ MIDE	EJEMPLO
review_wait_p95	Tiempo p95 hasta primera decisión.	38 minutos.
backlog_size	Casos pendientes por cola.	71 casos.
deadline_breach_rate	Proporción fuera de SLO.	4,2%.
needs_more_info_rate	Casos que llegaron con evidencia insuficiente.	19%.
approval_rate	Porcentaje aprobado.	63%.
reversal_rate	Decisiones corregidas después.	1,1%.
review_cost_per_case	Coste humano estimado por revisión.	3,40 EUR.
agreement_rate	Consistencia entre revisores en muestra doble.	0,86.

Un SLO posible:

$$SLO_{review} = P(T_{decision} \leq 30 \text{ min}) \geq 0,95$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$T_{decision}$	Tiempo desde que se crea la revisión hasta primera decisión.	18 minutos.
30 min	Objetivo máximo para la cola.	Media hora.
0,95	Proporción mínima dentro del objetivo.	95 de cada 100 casos.

El presupuesto de revisión no es infinito. Si la cola crece, tienes tres palancas:

PALANCA	QUÉ CAMBIA	CUIDADO
Mejorar evidencia	Menos needs_more_info .	Exige tocar prompts, retrieval o contrato.
Ajustar umbrales	Menos casos llegan a cola.	Puede aumentar automatización en casos discutibles.
Cambiar capacidad	Más personas o turnos.	Cuesta dinero y puede variar criterio.

Backpressure: qué hacer si la cola se satura

Una cola saturada no es solo un problema de soporte. Cambia el comportamiento del sistema de IA. Si el sistema sigue metiendo casos en revisión cuando nadie puede atenderlos, acumula deuda, incumple expectativas y oculta el fallo real.

Política de saturación:

SEÑAL	UMBRAL EJEMPLO	ACCIÓN
backlog_size	Más de 200 casos.	Bajar automatización que genera revisiones dudosas.
review_wait_p95	Más de 45 min.	Reasignar cola o ampliar capacidad temporal.
deadline_breach_rate	Más de 5%.	Responder con estado claro en vez de prometer resolución inmediata.
needs_more_info_rate	Más de 25%.	Mejorar bundle antes de crear revisión.
agreement_rate	Menos de 0,75.	Pausar decisiones automáticas basadas en esa rúbrica.

Estrategias de degradación:

ESTRATEGIA	QUÉ HACE	CUÁNDO USARLA
queue_only_critical	Solo entran casos de alto impacto.	Backlog alto y poco personal.
auto_reject_low_evidence	No crea revisión si falta evidencia básica.	Muchos casos llegan vacíos.
fallback_response	Informa que el caso queda pendiente.	La respuesta inmediata sería engañosa.
batch_review	Agrupar casos parecidos.	Muchos casos con mismo motivo.
threshold_raise	Exige más confianza para actuar solo o más evidencia para revisar.	Canary o incidente operativo.
route_to_specialist	Cambia cola según tema.	La cola general bloquea casos especializados.

La regla: **si la cola está llena, el sistema debe cambiar de comportamiento**. No basta con quejarse del backlog.

La tarjeta de decisión

La persona revisora no debería tener que buscar en diez sistemas. La tarjeta debe responder rápido a siete preguntas:

PREGUNTA	LO QUE DEBE VER
¿Qué quiere hacer el sistema?	Acción concreta y efecto.
¿Por qué llegó a revisión?	Motivos codificados y explicación corta.
¿Qué información usó?	Fuentes, documentos, campos y trazas.
¿Qué falta?	Evidencia ausente o dato que debe comprobarse.
¿Qué pasa si apruebo?	Acción exacta y recurso afectado.
¿Qué pasa si rechazo?	Mensaje a la run y siguiente estado.
¿Qué debo dejar escrito?	Motivo, corrección, caso de regresión o enlace a runbook.

Botones sanos:

BOTÓN	QUÉ HACE
approve	Reanuda la run y ejecuta la acción propuesta.
approve_with_edits	Reanuda con una salida corregida por la persona.
reject	No ejecuta la acción y devuelve motivo al sistema.
needs_more_info	Pide datos concretos y deja la run pausada.
reassign	Cambia cola o persona responsable.

Botones peligrosos por diseño:

BOTÓN	POR QUÉ LO EVITARÍA
ok	No dice qué ocurre.
continuar	Oculto si ejecuta, responde o solo guarda.
resolver	Mezcla decisión, ejecución y cierre.
saltar	No deja claro si rechaza, pospone o ignora.

Datos visibles, datos ocultos y privacidad

La tarjeta de revisión debe mostrar lo suficiente para decidir, pero no todo lo que el sistema sabe. Minimizar datos también es ingeniería.

TIPO DE DATO	MOSTRAR POR DEFECTO	MOTIVO
Acción propuesta	Sí.	Sin esto no hay decisión.
Efecto técnico	Sí.	La persona debe saber qué cambia.
Fuentes usadas	Sí, si tiene permiso.	Permite comprobar evidencia.
Datos personales completos	No.	Mostrar solo campos necesarios o redactados.
Prompt completo	Normalmente no.	Puede contener instrucciones internas o datos no necesarios.
Traza completa	No en tarjeta; sí enlazada.	La tarjeta debe ser legible y la auditoría completa debe existir.
Coste y latencia	Sí.	Afecta decisión operativa.
Identidad de otras personas	Solo si es necesario.	Reduce exposición innecesaria.

Campos recomendados:

```
visible_to_reviewer:
  - proposed_action
  - effect
  - draft
  - sources
  - missing_evidence
  - reason_codes
  - cost_eur
  - deadline_at
redacted_by_default:
  - raw_prompt
  - full_trace_payload
  - credentials
  - unrelated_personal_data
  - provider_internal_metadata
```

Una revisión útil no necesita enseñar todo. Necesita enseñar lo necesario, con enlaces auditables para quien tenga permiso.

RACI operativo: quién puede decidir qué

La revisión humana se vuelve frágil cuando cualquiera puede aprobar cualquier cosa. Un RACI ayuda a separar responsabilidad.

CASO	RESPONSIBLE	ACCOUNTABLE	CONSULTED	INFORMED
Respuesta de soporte común	Soporte nivel 1	Responsable de soporte	Producto	Usuario final
Caso con norma ambigua	Soporte nivel 2	Responsable académico	Legal o coordinación	Usuario final
Cambio de configuración IA	Plataforma IA	Lead técnico	Producto y soporte	Equipo
Tool con efecto externo	Equipo dueño del sistema	Owner del proceso	Plataforma IA	Auditoría
Incumplimiento de SLO	Guardia operativa	Responsable de operación	Soporte	Producto

Traducción práctica:

ROL	PUEDE APROBAR	NO DEBERÍA APROBAR
Soporte nivel 1	Respuestas con fuente y efecto bajo.	Cambios persistentes o casos ambiguos.
Soporte nivel 2	Casos con expediente, excepción o criterio.	Cambios de infraestructura.
Plataforma IA	Reanudación técnica, rutas, políticas y gates.	Interpretaciones de negocio sin owner.
Producto	Cambios de experiencia y criterio funcional.	Ejecución técnica sin evidencia.
Guardia operativa	Acciones para proteger SLO y cola.	Decisiones de contenido especializado.

Handoffs en SDKs y runtimes

La forma concreta cambia según la herramienta, pero el patrón se repite.

ECOSISTEMA	MECANISMO RELEVANTE	CÓMO LO TRADUCIRÍA A OPERACIÓN
OpenAI Agents SDK	Tools con aprobación, interrupciones, <code>RunState</code> serializable y reanudación.	Guardar pending approvals como cola y reanudar desde estado versionado.
OpenAI Agents SDK	Handoffs entre agentes.	Separar especialización de revisión: no todo handoff es humano.
Claude Agent SDK	Permisos, allow/deny, modos y <code>canUseTool</code> .	Declarar qué se permite, qué se bloquea y qué pide decisión en runtime.
Claude Agent SDK	Hooks.	Interceptar tool calls para añadir política propia, logs y gates.
Google ADK	Callbacks.	Insertar controles antes/después de agente, modelo o tool.
LangGraph	Interrupts y checkpoints.	Pausar grafo, guardar estado y reanudar con decisión.
OpenTelemetry	Trazas y spans.	Unir revisión, run, tool, coste y decisión en una historia consultable.

OpenTelemetry define las trazas como una forma de seguir el camino de una petición mediante spans relacionados.¹⁴ En un handoff, la traza debería permitir responder: qué agente iba actuando, qué tool se propuso, qué política bloqueó, quién decidió y cómo se reanudó.

Calidad de la revisión

La revisión humana también se evalúa. Una persona puede tener prisa, interpretar una rúbrica de forma distinta o aprobar casos porque la interfaz no explica bien el efecto.

CONTROL	QUÉ DETECTA	CÓMO SE APLICA
Doble revisión muestral	Inconsistencia entre personas.	5% de casos van a dos revisores.
Rúbrica versionada	Cambios de criterio.	<code>review_rubric@4</code> .
Casos calibrados	Deriva del equipo.	Casos fijos con respuesta esperada.
Revisión de reversals	Decisiones corregidas después.	Analizar por motivo y persona.
Tiempo por decisión	Decisiones demasiado rápidas o lentas.	Mirar distribución, no solo media.
Feedback al sistema	Mismo motivo aparece muchas veces.	Abrir trabajo de prompt, RAG o contrato.

El acuerdo simple se calcula así:

$$A = \frac{N_{same}}{N_{double}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
A	Tasa de acuerdo simple.	0,86.
N_{same}	Casos donde dos revisores tomaron la misma decisión.	86.
N_{double}	Casos revisados por dos personas.	100.

No es una métrica perfecta, pero ayuda a detectar si la cola está midiendo el criterio del sistema o el humor del día.

Los trabajos sobre model cards y datasheets enseñan una idea que aquí encaja muy bien: documentar estructura, límites, datos, supuestos y uso previsto reduce ambigüedad operativa.¹⁵¹⁶ En revisión humana ocurre lo mismo: si la tarjeta no dice qué se está decidiendo, cada persona rellena huecos con su interpretación.

Qué te llevas para poner en práctica

Al terminar este capítulo, el lector debería poder montar una cola mínima de revisión para un sistema de IA.

ARTEFACTO	PARA QUÉ SIRVE EN UN PROYECTO REAL	QUÉ DEBERÍA CONTENER
<code>handoff_request.schema.json</code>	Contrato de entrada a la cola.	IDs, acción, efecto, evidencia, motivos, deadline y opciones.
<code>handoff_policy.yml</code>	Reglas de revisión.	Umbral de confianza, coste, efectos y evidencia obligatoria.
<code>handoff_queue.py</code>	Simulador operativo.	Prioridad, decisión, SLA y mensaje de reanudación.
<code>handoff_state_machine.md</code>	Estados y transiciones.	Qué estados existen, quién los cambia y qué eventos emiten.
<code>handoff_tables.sql</code>	Persistencia mínima.	Requests, decisiones, eventos y asignaciones.
<code>review_decision.md</code>	Registro humano.	Decisión, motivo, cambios y caso de regresión.
<code>review_dashboard.sql</code>	Consultas de cola.	Backlog, aging, breaches y tasa de aprobación.
<code>privacy_review.yml</code>	Minimización de datos.	Campos visibles, redactados y enlazados.
<code>raci_handoff.md</code>	Ownership operativo.	Quién puede aprobar cada tipo de caso.
<code>runbook_handoff.md</code>	Procedimiento del equipo.	Quién revisa, cuándo, cómo escalar y cómo cerrar.

Práctica recomendada:

1. Define dos acciones: una que pueda seguir sola y otra que requiera revisión.
2. Escribe tres motivos de revisión con códigos estables.
3. Diseña el contrato JSON.
4. Ejecuta el script de cola con casos de ejemplo.
5. Genera una decisión `approve` , una `reject` y una `needs_more_info` .
6. Añade una clave de idempotencia para que una aprobación no duplique efectos.

7. Define qué ocurre si la cola supera el SLO.
8. Convierte al menos un caso revisado en entrada de regresión.

El resultado práctico no es “tenemos humanos en el bucle”. Es poder demostrar qué entra, por qué entra, quién decide, cuánto tarda y cómo se reanuda.

Reto de capítulo: una cola de revisión en 45 minutos

Este es el ejercicio que me gustaría que alguien pudiera llevarse del capítulo y hacer de verdad. No es leer el concepto: es montar una mini cola reproducible.

Escenario: tienes un asistente de soporte que prepara respuestas. Algunas pueden guardarse como nota interna; otras quieren enviar un mensaje externo. Tu trabajo es decidir qué sigue solo, qué va a revisión y qué decisión deja preparada la reanudación.

Archivos que vas a crear:

```
mi-proyecto/  
  ops/  
    ai/  
      handoff_queue.py  
  data/  
    handoff_examples.jsonl  
  output/  
    handoff_queue_result.json
```

Datos de entrada en `data/handoff_examples.jsonl`:

```
{
  "request_id": "rev_001",
  "run_id": "run_7f31",
  "trace_id": "trace_abc",
  "task": "support_reply",
  "proposed_action": "send_email",
  "effect": "external_message",
  "confidence": 0.68,
  "cost_eur": 0.018,
  "created_at": "2026-05-28T13:45:00+00:00",
  "sla_minutes": 30,
  "evidence": {
    "draft": "Tu solicitud queda pendiente de revisar el expediente actualizado."
  },
  "sources": [
    {
      "doc_matricula_2026",
      "schema_valid": true,
      "contract_version": "support_reply_schema@4"
    }
  ],
  "missing_evidence": [
    {
      "expediente_actualizado",
      "resume_token": "resume_run_7f31_step_12"
    }
  ],
  "request_id": "rev_002",
  "run_id": "run_8a20",
  "trace_id": "trace_def",
  "task": "ticket_summary",
  "proposed_action": "save_summary",
  "effect": "internal_note",
  "confidence": 0.91,
  "cost_eur": 0.006,
  "created_at": "2026-05-28T14:05:00+00:00",
  "sla_minutes": 240,
  "evidence": {
    "draft": "Resumen interno del caso con próximos pasos."
  },
  "sources": [
    {
      "ticket_991",
      "schema_valid": true,
      "contract_version": "summary_schema@2"
    }
  ],
  "missing_evidence": [
    {
      "resume_token": "resume_run_8a20_step_05"
    }
  ],
  "request_id": "rev_003",
  "run_id": "run_9c10",
  "trace_id": "trace_xyz",
  "task": "policy_answer",
  "proposed_action": "answer_user",
  "effect": "external_message",
  "confidence": 0.74,
  "cost_eur": 0.041,
  "created_at": "2026-05-28T13:20:00+00:00",
  "sla_minutes": 60,
  "evidence": {
    "draft": "Según la política vigente, el plazo ordinario termina el viernes."
  },
  "sources": [
    {
      "fuente_normativa",
      "schema_valid": true,
      "contract_version": "policy_answer_schema@7"
    }
  ],
  "missing_evidence": [
    {
      "resume_token": "resume_run_9c10_step_09"
    }
  ]
}
```

Comandos:

```
mkdir -p ops/ai data output
python ops/ai/handoff_queue.py --write
cat output/handoff_queue_result.json
```

Qué debería ocurrir:

CASO	RESULTADO ESPERADO	POR QUÉ
rev_001	Entra en cola.	Baja confianza, efecto externo y falta expediente.
rev_002	Sigue automáticamente.	Es nota interna, tiene evidencia y supera umbrales.
rev_003	Entra primero en cola.	Efecto externo, coste alto y falta fuente normativa.

La entrega mínima del alumno:

- 1. Captura o salida de `handoff_queue_result.json`.
- 2. Una decisión escrita para `rev_003`.
- 3. Una frase explicando por qué `rev_002` no necesita revisión.
- 4. Una modificación de la política: subir o bajar `min_confidence` y justificar el efecto.
- 5. Un caso nuevo añadido al JSONL que pruebe `contract_failed`.

En el día a día

El handoff operativo aparece en cuanto un sistema de IA hace algo que un humano debe poder revisar, aprobar o retomar. La escena habitual: el agente prepara una respuesta, una acción o una propuesta, pero antes de ejecutarla con efecto se la pasa a una persona. Lo que parece un detalle se convierte en ingeniería en cuanto te preguntas lo concreto: ¿qué le llega exactamente a quien revisa? ¿Tiene el contexto, la evidencia y el diff para decidir en treinta segundos, o tiene que reconstruir media historia? ¿Qué pasa si nadie lo revisa en una hora, en un día? Un handoff sin cola, sin prioridad y sin caducidad no es supervisión humana, es un cuello de botella esperando a saturarse.

La segunda situación es la del relevo entre turnos o entre personas. Un sistema operable no para de funcionar porque cambie el turno; lo que cambia es quién atiende la cola de revisiones. Para que ese relevo no pierda casos, el handoff necesita estado: qué está pendiente, qué se aprobó, qué se rechazó y por qué. Sin ese estado, cada relevo es una pequeña amnesia, y las amnesias en operación se pagan con casos que se quedan colgados.

Por qué debería importarte

Porque la supervisión humana es la red de seguridad de casi todo sistema de IA con efecto real, y una red mal tejida da una falsa sensación de seguridad. Diseñar el handoff (qué información viaja, con qué prioridad, con qué caducidad y a quién escala si nadie responde) es lo que hace que «hay un humano en el bucle» signifique de verdad que alguien puede y va a revisar a tiempo, y no que hay un buzón donde las cosas se acumulan hasta que alguien se queja.

Y hay una razón que se entiende el día de una auditoría o de un incidente. Un handoff bien diseñado deja rastro: quién aprobó qué, con qué evidencia y cuándo. Ese rastro es lo que te permite responder con calma cuando alguien pregunta por qué el sistema hizo algo, en vez de encogerte de hombros. La revisión humana sin trazabilidad es casi tan frágil como no tener revisión.

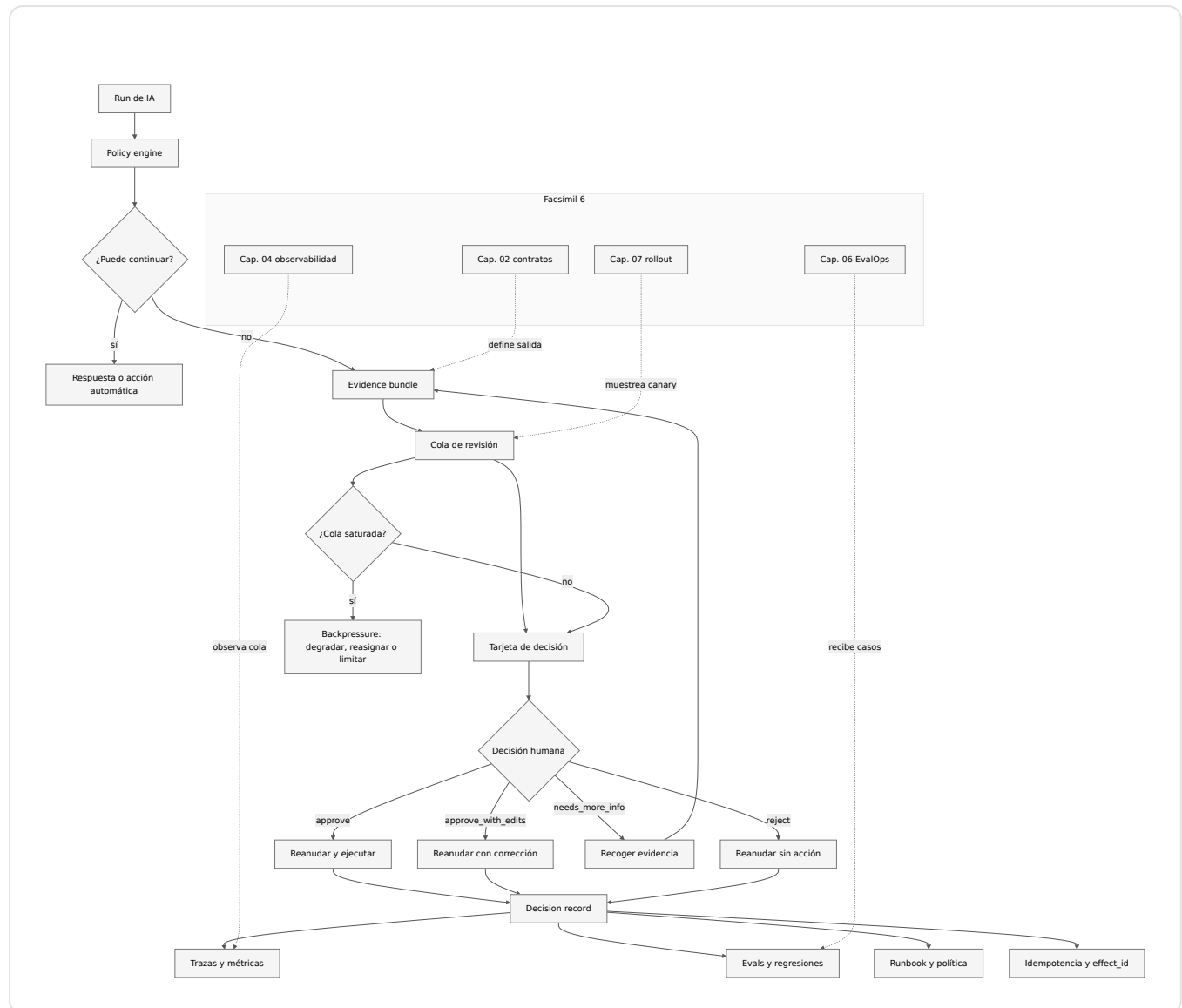
Dónde solía tropezar yo

Me costó entender que revisión humana no es una red de seguridad automática. Si llega tarde, sin contexto o sin criterio, solo traslada el problema a otra persona.

TROPIEZO	ANTÍDOTO
Mandar demasiadas cosas a revisión.	Codificar motivos y medir tasa por motivo.
Mandar casos sin evidencia suficiente.	Exigir <code>evidence</code> y <code>missing_evidence</code> .
No medir la cola.	Crear SLI/SLO de espera, backlog y breaches.
No reanudar bien.	Guardar <code>resume_token</code> , decisión y siguiente estado.
No aprender de revisiones.	Convertir casos en evals, reglas y runbooks.

La frase que me habría ahorrado muchas vueltas: **un handoff no es un mensaje; es un contrato de continuidad.**

Cómo encaja todo



Relación con otros capítulos

CAPÍTULO	QUÉ APORTA AQUÍ
F5 · Capítulo 08	Permisos, autonomía y aprobación de tools.
F6 · Capítulo 02	Estados <code>needs_review</code> , contratos de API y respuesta estructurada.
F6 · Capítulo 04	Trazas, métricas, SLIs y SLOs.
F6 · Capítulo 06	Casos revisados que vuelven a datasets y gates.
F6 · Capítulo 07	Canary y revisión muestreada antes de ampliar exposición.

Inteligencia artificial para gente curiosa · 686f6c61 · CC BY 4.0

Amershi y colaboradores mostraron que los sistemas de ML en producción traen retos propios de datos, operación, monitorización y evolución.¹⁷ El NIST AI RMF organiza la gestión de sistemas de IA alrededor de gobernar, mapear, medir y gestionar.¹⁸ Un handoff operativo es una forma concreta de llevar esas ideas al día a día: no basta con confiar en el sistema; hay que diseñar cómo se detiene, cómo pide criterio y cómo aprende.

Para entenderlo

Tres situaciones concretas:

SITUACIÓN	MALA SALIDA	HANDOFF OPERATIVO
Respuesta de soporte sin fuente.	“Creo que el plazo es el viernes”.	Pausa, pide fuente normativa y deja <code>needs_more_info</code> .
Tool quiere cerrar un ticket.	Cierra porque el resumen parece correcto.	Crea tarjeta con acción, evidencia, efecto y botón <code>approve</code> .
Canary de prompt nuevo.	Se mira el promedio y se sube al 50%.	Muestrea revisiones de casos nuevos y añade regresiones.

La diferencia no está en “meter una persona”. Está en que la persona entre justo donde el sistema necesita criterio, y salga dejando datos que mejoran el siguiente ciclo.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
Handoff operativo	Pausa estructurada de una run para que otra persona o sistema pueda decidir.
Evidence bundle	Paquete de entrada, propuesta, fuentes, trazas, motivos y estado.
Approval request	Solicitud concreta de aprobación sobre una acción.
Review queue	Cola priorizada de casos pendientes de decisión.
Decision record	Registro de qué se decidió, quién lo hizo y por qué.
Resume token	Identificador o estado que permite reanudar una run.
SLO de revisión	Objetivo medible de tiempo o calidad de la cola.
Agreement rate	Proporción de decisiones iguales entre revisores en muestra doble.

Antes de pasar página

Comprueba que puedes responder:

1. ¿Qué diferencia hay entre revisión humana y handoff operativo?
2. ¿Qué campos mínimos debe incluir un `evidence_bundle` ?
3. ¿Cuándo pedirías revisión por efecto persistente?
4. ¿Qué SLI usarías para medir una cola de revisión?
5. ¿Por qué `approve` y `approve_with_edits` no deberían ser el mismo botón?
6. ¿Cómo se reanuda una run después de una decisión?
7. ¿Qué caso revisado añadirías a un dataset de regresión?

En resumen

IDEA	PARA LLEVARTE
Revisar no es improvisar.	La revisión debe tener contrato, evidencia, SLA y reanudación.
La cola se opera.	Backlog, p95, breaches, acuerdo y coste humano son métricas del sistema.
La interfaz decide calidad.	Si la tarjeta no muestra efecto y evidencia, la persona decide a ciegas.
Cada revisión debe enseñar algo.	Los casos vuelven a evals, prompts, políticas y runbooks.

Para saber más

- Amershi, S. y otros *Software Engineering for Machine Learning: A Case Study*. <https://doi.org/10.1109/ICSE-SEIP.2019.00042>
- Anthropic. *Claude Agent SDK: Handle Approvals and User Input*. <https://code.claude.com/docs/en/agent-sdk/user-input>
- Anthropic. *Claude Agent SDK: Hooks*. <https://code.claude.com/docs/en/agent-sdk/hooks>
- Anthropic. *Claude Agent SDK: Permissions*. <https://code.claude.com/docs/en/agent-sdk/permissions>
- Gebru, T. y otros *Datasheets for Datasets*. <https://doi.org/10.1145/3458723>
- Google. *Callbacks: Observe, Customize, and Control Agent Behavior*. <https://adk.dev/callbacks/>
- Google. *Safety and Security for AI Agents*. <https://adk.dev/safety/>
- LangChain. *LangGraph Interrupts*. <https://docs.langchain.com/oss/python/langgraph/human-in-the-loop>
- Mitchell, M. y otros *Model Cards for Model Reporting*. <https://doi.org/10.1145/3287560.3287596>
- OpenAI. *Agents SDK: Handoffs*. <https://openai.github.io/openai-agents-python/handoffs/>
- OpenAI. *Agents SDK: Human-in-the-loop*. https://openai.github.io/openai-agents-python/human_in_the_loop/
- OpenTelemetry. *Traces*. <https://opentelemetry.io/docs/concepts/signals/traces/>
- Tabassi, E. *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. <https://doi.org/10.6028/NIST.AI.100-1>

Notas

1. OpenAI. (2026). *Agents SDK: Human-in-the-loop*. [https://openai.github.io/openai-agents-python/human in the loop/](https://openai.github.io/openai-agents-python/human%20in%20the%20loop/). Consultado el 28 de mayo de 2026. ↵
2. OpenAI. (2026). *Agents SDK: Handoffs*. <https://openai.github.io/openai-agents-python/handoffs/>. Consultado el 28 de mayo de 2026. ↵
3. Anthropic. (2026). *Claude Agent SDK: Permissions*. <https://code.claude.com/docs/en/agent-sdk/permissions>. Consultado el 28 de mayo de 2026. ↵
4. Anthropic. (2026). *Claude Agent SDK: Hooks*. <https://code.claude.com/docs/en/agent-sdk/hooks>. Consultado el 28 de mayo de 2026. ↵
5. Anthropic. (2026). *Claude Agent SDK: Handle Approvals and User Input*. <https://code.claude.com/docs/en/agent-sdk/user-input>. Consultado el 28 de mayo de 2026. ↵
6. Google. (2026). *Callbacks: Observe, Customize, and Control Agent Behavior*. <https://adk.dev/callbacks/>. Consultado el 28 de mayo de 2026. ↵
7. Google. (2026). *Safety and Security for AI Agents*. <https://adk.dev/safety/>. Consultado el 28 de mayo de 2026. ↵
8. LangChain. (2026). *LangGraph Interrupts*. <https://docs.langchain.com/oss/python/langgraph/human-in-the-loop>. Consultado el 28 de mayo de 2026. ↵
9. Elkan, C. (2001). The Foundations of Cost-Sensitive Learning. *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, 973-978. Define el umbral de decisión óptimo comparando el coste esperado de cada acción. ↵
10. Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1), 37-46. <https://doi.org/10.1177/001316446002000104>. Define kappa como acuerdo corregido por azar. ↵
11. Geifman, Y., & El-Yaniv, R. (2017). Selective Classification for Deep Neural Networks. *Advances in Neural Information Processing Systems*, 30. <https://papers.nips.cc/paper/2017/hash/4a8423d5e91fda00bb7e46540e2b0cf1-Abstract.html>. Formaliza la clasificación selectiva mediante la curva riesgo-cobertura. ↵
12. Brier, G. W. (1950). Verification of Forecasts Expressed in Terms of Probability. *Monthly Weather Review*, 78(1), 1-3. Define la puntuación cuadrática para pronósticos probabilísticos. ↵
13. Little, J. D. C. (1961). A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research*, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>. La relación se aplica a cualquier sistema estable, incluida una cola de revisión humana. ↵

- .4. OpenTelemetry. (2026). *Traces*. <https://opentelemetry.io/docs/concepts/signals/traces/>. Consultado el 27 de mayo de 2026. ↵
- .5. Mitchell, M. y otros (2019). *Model Cards for Model Reporting*. Proceedings of the Conference on Fairness, Accountability, and Transparency, 220-229. <https://doi.org/10.1145/3287560.3287596> ↵
- .6. Gebru, T. y otros (2021). *Datasheets for Datasets*. *Communications of the ACM*, 64(12), 86-92. <https://doi.org/10.1145/3458723> ↵
- .7. Amershi, S. y otros (2019). *Software Engineering for Machine Learning: A Case Study*. Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, 291-300. <https://doi.org/10.1109/ICSE-SEIP.2019.00042> ↵
- .8. Tabassi, E. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. NIST AI 100-1. <https://doi.org/10.6028/NIST.AI.100-1> ↵

Capítulo 09: Incidencias, postmortems y mejora continua

Qué deberías poder hacer al terminar

En el capítulo 08 diseñamos cómo pausar una run y pedir revisión. Ahora subimos un nivel: **qué ocurre cuando el sistema completo entra en una situación operativa que exige coordinación.**

Una incidencia de IA no siempre es “la API está caída”. Puede ser más sutil: sube el coste por run aceptada, el contrato JSON falla, el RAG cita documentos que no corresponden, el router manda demasiado tráfico al modelo caro, una cola de revisión se atasca, un canary degrada la calidad en un segmento o una tool empieza a devolver errores tipados.

Al terminar, deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Declarar una incidencia de IA.	Separas síntoma, impacto, severidad, servicio afectado y owner.
Priorizar respuesta.	Clasificas por usuarios, tareas, coste, contrato, SLO y reversibilidad.
Coordinar roles.	Distingues comandante, operaciones, comunicación, especialistas y documentación.
Mitigar sin perder evidencia.	Proteges el servicio y conservas trazas, eventos y decisiones.
Calcular tiempos operativos.	Mides detección, reconocimiento, mitigación y recuperación.
Escribir un postmortem útil.	Incluyes impacto, línea temporal, causas contribuyentes y acciones verificables.
Convertir aprendizaje en trabajo.	Creas casos de regresión, runbooks, alertas, cambios de política y gates.

La idea central: **una incidencia no termina cuando vuelve la gráfica; termina cuando el sistema aprende algo verificable.**

Cuando producción te enseña lo que faltaba

Imagina que un asistente de soporte llevaba dos semanas estable. De pronto, el coste p95 sube, la latencia se dispara y soporte empieza a recibir tickets de respuestas que tardan demasiado. Nadie cambió código de aplicación, pero sí se publicó un índice RAG nuevo y se abrió un canary de prompt.

Hay varias tentaciones: mirar logs al azar, culpar al modelo, revertir todo, escribir en el chat “lo estoy mirando” y esperar que alguien encuentre la causa. Eso no es operar. Operar es declarar el evento, acotar impacto, asignar roles, mitigar, preservar evidencia y dejar un registro que permita aprender.

La pregunta no es “quién lo rompió”. La pregunta útil es:

¿Qué señales vimos, qué decisiones tomamos, qué mitigó el impacto y qué cambio evitará que vuelva igual?

Qué no es gestionar una incidencia

Gestionar una incidencia no es correr más rápido que el dashboard. La velocidad sin coordinación suele crear cambios simultáneos, mensajes contradictorios y pérdida de evidencia.

Tampoco es buscar una causa única al final. En sistemas de IA, una incidencia suele tener causas contribuyentes: un cambio de datos, un umbral demasiado permisivo, un fallback que no estaba probado, una alerta tardía, un runbook incompleto o un canary que no segmentaba lo suficiente.

Y un postmortem no es un documento ceremonial. Si no produce acciones verificables, casos de regresión, alertas mejores o runbooks más claros, solo archiva una historia.

CONFUSIÓN	QUÉ FALTA
“La incidencia termina cuando baja la alerta”	Falta confirmar recuperación, cerrar comunicación y capturar aprendizaje.
“Postmortem es resumen”	Falta impacto, línea temporal, causas contribuyentes y acciones con dueño.
“Revertimos y listo”	Falta saber qué señal detectó tarde y qué dataset debe aprender.
“El modelo falló”	Falta distinguir modelo, prompt, RAG, router, runtime, contrato y proveedor.
“Ya sabemos lo que pasó”	Falta evidencia reproducible: trazas, métricas, eventos y decisiones.

Qué sí es una incidencia de IA

Para este facsímil, una incidencia de IA es:

Un evento operativo que degrada una propiedad importante del sistema: disponibilidad, latencia, coste, calidad, contrato, trazabilidad, permisos o confianza del flujo.

Una incidencia se modela con estos campos. No es una ecuación de la literatura, es una lista de ingeniería:

SÍMBOLO	SIGNIFICADO	EJEMPLO
<i>S</i>	Servicio o capacidad afectada.	support_rag , tool_gateway , review_queue .
<i>P</i>	Población afectada.	18% de tenants, canal chat, tarea policy_answer .
<i>T</i>	Ventana temporal.	13:40-14:35 UTC.
<i>E</i>	Evidencia observada.	Métricas, logs, trazas, tickets, decision records.
<i>M</i>	Mitigaciones aplicadas.	Rollback, fallback, bajar canary, desactivar tool.
<i>D</i>	Decisiones tomadas.	Quién decidió, cuándo y por qué.
<i>A</i>	Acciones de mejora.	Evals, alertas, runbooks, cambios de política.

Si falta *D*, no sabremos por qué se actuó. Si falta *E*, discutiremos recuerdos. Si falta *A*, la incidencia puede repetirse con otro nombre.

Fecha de corte del estado del arte

Fecha de corte: 28 de mayo de 2026.

Fuentes consultadas: capítulos de Google SRE sobre monitorización, respuesta de emergencia, gestión de incidencias, postmortems, alerting on SLOs y sobrecarga; OpenTelemetry sobre trazas, logs, métricas y convenciones GenAI; NIST AI RMF; y trabajos de ingeniería de ML.

El libro de SRE de Google recomienda que las alertas que interrumpen a personas sean simples, accionables y orientadas a síntomas relevantes para usuarios.¹ El SRE Workbook explica cómo alertar sobre SLOs usando consumo de presupuesto de error, no solo umbrales aislados.²

Google SRE también insiste en preparar la respuesta a emergencias antes de necesitarlas, practicarla y pedir ayuda cuando la situación supera a una persona.³ En gestión de incidencias, separa roles como comando, trabajo operativo, comunicación y planificación para evitar cambios descoordinados.⁴

Sobre postmortems, Google SRE los define como registros escritos de impacto, acciones tomadas, causas contribuyentes y seguimiento para evitar recurrencia, con una cultura centrada en aprender y mejorar sistemas.⁵ Lo estable es el método: declarar, coordinar, mitigar, documentar, aprender y verificar. Lo cambiante son herramientas concretas de guardia, chat, tickets, dashboards y proveedores.

Severidad: no todo merece el mismo ruido

La severidad debe estar definida antes de la incidencia. Si se decide en caliente, cada persona trae su propio umbral.

SEVERIDAD	IMPACTO	EJEMPLO EN IA	RESPUESTA
SEV1	Servicio principal no cumple función crítica o hay impacto amplio.	El asistente no responde o rompe contrato en producción para muchas tareas.	Comando formal, mitigación inmediata, comunicación periódica.
SEV2	Degradación importante, acotada o con workaround.	p95 fuera de SLO en una tarea clave; coste p95 se duplica.	Equipo operativo, owner, update regular, postmortem si supera umbral.
SEV3	Problema limitado, sin impacto amplio.	Una ruta RAG falla en un tenant o canal.	Ticket prioritario, runbook, revisión posterior.
SEV4	Anomalía o mejora preventiva.	Alerta de tendencia, flake rate subiendo, cola creciendo.	Trabajo planificado antes de que escale.

Para IA conviene clasificar por más dimensiones que “está caído”:

DIMENSIÓN	PREGUNTA
Disponibilidad	¿El usuario puede completar la tarea?
Latencia	¿La experiencia sigue dentro del SLO?
Contrato	¿La salida sigue siendo parseable y compatible?
Calidad	¿Aumentan rechazos, revisiones o quejas verificables?
Coste	¿Se consume presupuesto de forma anómala?
Trazabilidad	¿Podemos reconstruir lo ocurrido?
Reversibilidad	¿Podemos volver a estado conocido sin pérdida?
Alcance	¿Cuántos tenants, tareas, idiomas o canales afecta?

Fórmulas operativas que sí usaría

El primer grupo mide tiempos:

$$MTTA = \frac{1}{n} \sum_{i=1}^n (t_i^{ack} - t_i^{detect})$$

$$MTTR = \frac{1}{n} \sum_{i=1}^n (t_i^{recover} - t_i^{start})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$MTTA$	Tiempo medio hasta reconocer la incidencia.	7 minutos.
$MTTR$	Tiempo medio hasta recuperar.	42 minutos.
t_i^{detect}	Momento en que el sistema detecta el evento.	13:43 UTC.
t_i^{ack}	Momento en que una persona o proceso lo reconoce.	13:48 UTC.
t_i^{start}	Inicio de la incidencia.	13:40 UTC.
$t_i^{recover}$	Recuperación confirmada.	14:22 UTC.
n	Número de incidencias medidas.	12.

La prioridad de una incidencia es una puntuación ponderada de varios factores, una decisión multicriterio.⁶

$$P = 4U + 3C + 2L + 2K + R$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
P	Puntuación de prioridad.	23.
U	Alcance de usuarios o tenants afectados, de 0 a 5.	3.
C	Criticidad de la tarea, de 0 a 5.	4.
L	Degradación de latencia o disponibilidad, de 0 a 5.	2.
K	Degradación de contrato/calidad, de 0 a 5.	3.
R	Penalización por baja reversibilidad, de 0 a 5.	1.

Ejemplo:

$$P = 4 \cdot 3 + 3 \cdot 4 + 2 \cdot 2 + 2 \cdot 3 + 1 = 35$$

Con $P = 35$, probablemente no estamos ante un ticket normal. Necesitamos coordinación, mitigación y seguimiento.

También conviene recordar que las colas obedecen matemáticas. La ley de Little relaciona trabajos en sistema, tasa de llegada y tiempo medio.⁷ Si la cola de revisión o de serving crece, no basta con “esperar un poco”: o baja llegada, o sube capacidad, o aumenta tiempo de espera.

Cuándo esperar el próximo fallo: fiabilidad exponencial. Si los fallos ocurren al azar a tasa constante, el tiempo hasta el siguiente sigue una distribución exponencial, y la probabilidad de aguantar sin incidencias decae con el tiempo de forma conocida.⁸

$$R(t) = e^{-\lambda t}, \qquad \text{MTBF} = \frac{1}{\lambda}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
λ	Tasa de incidencias por unidad de tiempo.	0,5 por semana.
t	Horizonte de tiempo considerado.	2 semanas.
$R(t)$	Probabilidad de no tener incidencias hasta t .	$e^{-1} \approx 0,37$.
MTBF	Tiempo medio entre incidencias.	2 semanas.

En palabras: con media incidencia por semana, la probabilidad de pasar dos semanas limpias es solo del 37%. Esto desmonta el «llevamos una semana sin incidencias, vamos bien»: una racha corta es ruido esperable, no evidencia de mejora. Para afirmar que mejoras necesitas ver bajar λ sostenidamente, no una buena semana suelta.

Qué causas atacar primero: el principio de Pareto. No todas las causas de incidencia pesan igual. La ley de Pareto, observada una y otra vez en sistemas operativos, dice que una minoría de causas explica la mayoría de los fallos, lo que da una regla de priorización para los postmortems.⁹

$$P(X > x) = \left(\frac{x_m}{x}\right)^\alpha$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
x_m	Valor mínimo del fenómeno.	1 incidencia.
α	Parámetro de forma (concentración).	$\approx 1,16$ para el 80/20.
$P(X > x)$	Probabilidad de superar x .	Cola pesada: pocas causas, mucho efecto.

En palabras: si el 20% de las causas provoca el 80% de las incidencias, ordenar los postmortems por frecuencia y atacar primero esa minoría rinde mucho más que repartir esfuerzo por igual. La mejora continua no es arreglar todo, es arreglar lo que más pesa, y Pareto dice cuánto pesa cada cosa.

¿Es buena tu operación? Las métricas DORA. Para saber si operas bien necesitas un marco comparable, no una sensación. La investigación DORA identificó cuatro métricas que distinguen a los equipos de alto rendimiento; dos son directamente de incidencias: cada cuánto fallan los cambios y cuánto tardas en recuperarte.¹⁰

$$CFR = \frac{\text{despliegues con incidencia}}{\text{despliegues totales}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
CFR	<i>Change Failure Rate</i> : fracción de cambios que provocan incidencia.	0,10
MTTR	Tiempo medio de recuperación de un cambio fallido.	menos de 1 hora en equipos de élite.
despliegues totales	Cambios desplegados en la ventana.	200

En palabras: un CFR del 10% y un MTTR de minutos caracterizan a un equipo que entrega rápido y se recupera antes; un CFR alto con MTTR de días, lo contrario. Lo valioso es que son comparables entre equipos y a lo largo del tiempo: te dicen si tu mejora continua se nota en los números o solo en las reuniones.

¿Estás mejorando de verdad? Crecimiento de fiabilidad de Crow-AMSAA. Si cada postmortem corrige causas, el sistema debería fallar cada vez menos. El modelo de Crow-AMSAA cuantifica esa mejora: el número acumulado de fallos sigue una ley de potencia cuyo exponente dice si estás mejorando, estancado o empeorando.¹¹

$$N(t) = \lambda t^\beta$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$N(t)$	Número acumulado de incidencias hasta t .	Curva que se aplanas si mejoras.
β	Exponente de crecimiento.	< 1 mejora; = 1 estable; > 1 empeora.
λ	Escala inicial.	Ajustada a los datos.

En palabras: un exponente menor que 1 significa que el tiempo entre incidencias crece, es decir, que los postmortems funcionan; mayor que 1 significa que el sistema se degrada más rápido de lo que lo arreglas. Es la forma honesta de responder «¿la mejora continua está funcionando?» con la curva real de incidencias, no con la intención.

Tipos de incidencia propios de IA

TIPO	SÍNTOMA	PRIMERAS COMPROBACIONES
Contrato	JSON inválido, campos extra, enum incorrecto.	<code>contract_version</code> , salida raw, validador, cambio de prompt/modelo.
Calidad	Más rechazos, revisiones o casos corregidos.	Muestras, judge calibrado, feedback, canary, dataset afectado.
RAG	Fuentes pobres, citas ausentes, documentos obsoletos.	<code>index_version</code> , retrieval trace, <code>top_k</code> , reranker, fechas de documento.
Routing	Ruta cara o lenta sube de golpe.	<code>route_catalog</code> , distribución de tareas, fallback, proveedor.
Coste	Coste p95 o total diario sube.	Tokens, reintentos, contexto, modelo, cache hit rate.
Serving	Saturación de workers, cola, KV cache, timeouts.	Goodput, queue depth, GPU/CPU, batch size, p95/p99.
Handoff	Cola de revisión crece o caduca.	Backlog, <code>needs_more_info_rate</code> , owners, SLO de cola.
Trazabilidad	No se puede reconstruir la run.	<code>trace_id</code> , spans, sampling, logs, atributos obligatorios.

Dean y Barroso mostraron que la cola de la latencia importa especialmente en sistemas a gran escala: una parte pequeña de operaciones lentas puede dominar la experiencia.¹² En IA esto se agrava por contexto largo, herramientas, colas de serving, reintentos y validación.

Taxonomía por capa: dónde mirar primero

Un ingeniero de IA necesita separar capas. Si todo se llama “fallo del modelo”, el postmortem no produce buenas acciones.

CAPA	QUÉ PUEDE DEGRADARSE	SEÑALES	MITIGACIÓN TÍPICA
Producto	Usuario no completa tarea.	Tickets, abandono, baja aceptación.	Respuesta de estado, handoff, rollback de experiencia.
Contrato	Consumidor no parsea salida.	<code>contract_fail_rate</code> , errores de schema.	Schema anterior, validador estricto, prompt rollback.
Prompt	Cambia formato, tono, abstención o longitud.	Tokens, longitud, fallos de JSON, quejas.	Prompt anterior, límite de salida, eval de regresión.
Modelo	Latencia, coste, calidad o tool calling cambian.	<code>model_id</code> , p95, coste, flake rate.	Cambiar ruta, bajar esfuerzo, fallback de modelo.
RAG	Recuperación trae ruido o documentos obsoletos.	<code>index_version</code> , chunks, citas, recall manual.	Índice anterior, bajar <code>top_k</code> , desactivar reranker.
Router	Demasiado tráfico va a ruta cara/lenta.	<code>route_mix</code> , fallback, coste por tarea.	Catálogo anterior, override por tarea.
Tool gateway	Args inválidos, permisos o errores externos.	<code>tool_error_rate</code> , args, scope, retries.	Modo lectura, bloquear tool, handoff.
Serving	Saturación, colas, timeouts, memoria.	queue depth, p95/p99, goodput, GPU/CPU.	Rate limit, autoscaling, reducir contexto.
Observabilidad	Faltan trazas o atributos.	<code>trace_completeness</code> , sampling, logs.	Subir sampling temporal, añadir atributos obligatorios.

Checklist de primera hora:

- 1. ¿Qué cambió en las últimas 24 horas: prompt, modelo, índice, router, proveedor, política, código o tráfico?
- 2. ¿El síntoma aparece por `task` , `tenant` , `model_id` , `prompt_version` , `index_version` o `release_id` ?
- 3. ¿La mitigación más reversible está clara?
- 4. ¿Tenemos suficiente evidencia antes de borrar o sobrescribir estado?
- 5. ¿Qué caso concreto debería entrar en regresión?

Queries reales durante la incidencia

Las incidencias se operan mejor con preguntas concretas. Ejemplos con PromQL:

```
histogram_quantile(
  0.95,
  sum(rate(ai_run_latency_seconds_bucket{task="support_reply"}[10m])) by (le, release_id,
variant)
)
```

```
sum(rate(ai_contract_fail_total{task="support_reply"}[10m])) by (release_id, variant)
/
sum(rate(ai_run_total{task="support_reply"}[10m])) by (release_id, variant)
```

```
histogram_quantile(
  0.95,
  sum(rate(ai_run_cost_eur_bucket{task="support_reply"}[30m])) by (le, model_id, route_id)
)
```

```
sum(rate(ai_tool_error_total{tool_name="crm_lookup"}[10m])) by (error_type, release_id)
```

Y con SQL para eventos de producto:

```
select
  task,
  release_id,
  variant,
  count(*) as runs,
  avg(case when contract_valid then 0 else 1 end) as contract_fail_rate,
  avg(case when accepted_by_user then 1 else 0 end) as acceptance_rate,
  percentile_cont(0.95) within group (order by latency_ms) as latency_p95_ms
from ai_run_events
where created_at >= now() - interval '2 hours'
group by task, release_id, variant
order by contract_fail_rate desc, latency_p95_ms desc;
```

Estas consultas deberían vivir en el runbook. Si las inventas durante la incidencia, perderás minutos y quizá harás preguntas distintas cada vez.

Replay: reproducir antes de prometer causa

Una incidencia de IA no queda entendida hasta que podemos reproducir al menos una parte. No siempre se puede reproducir todo, pero sí guardar un paquete mínimo.

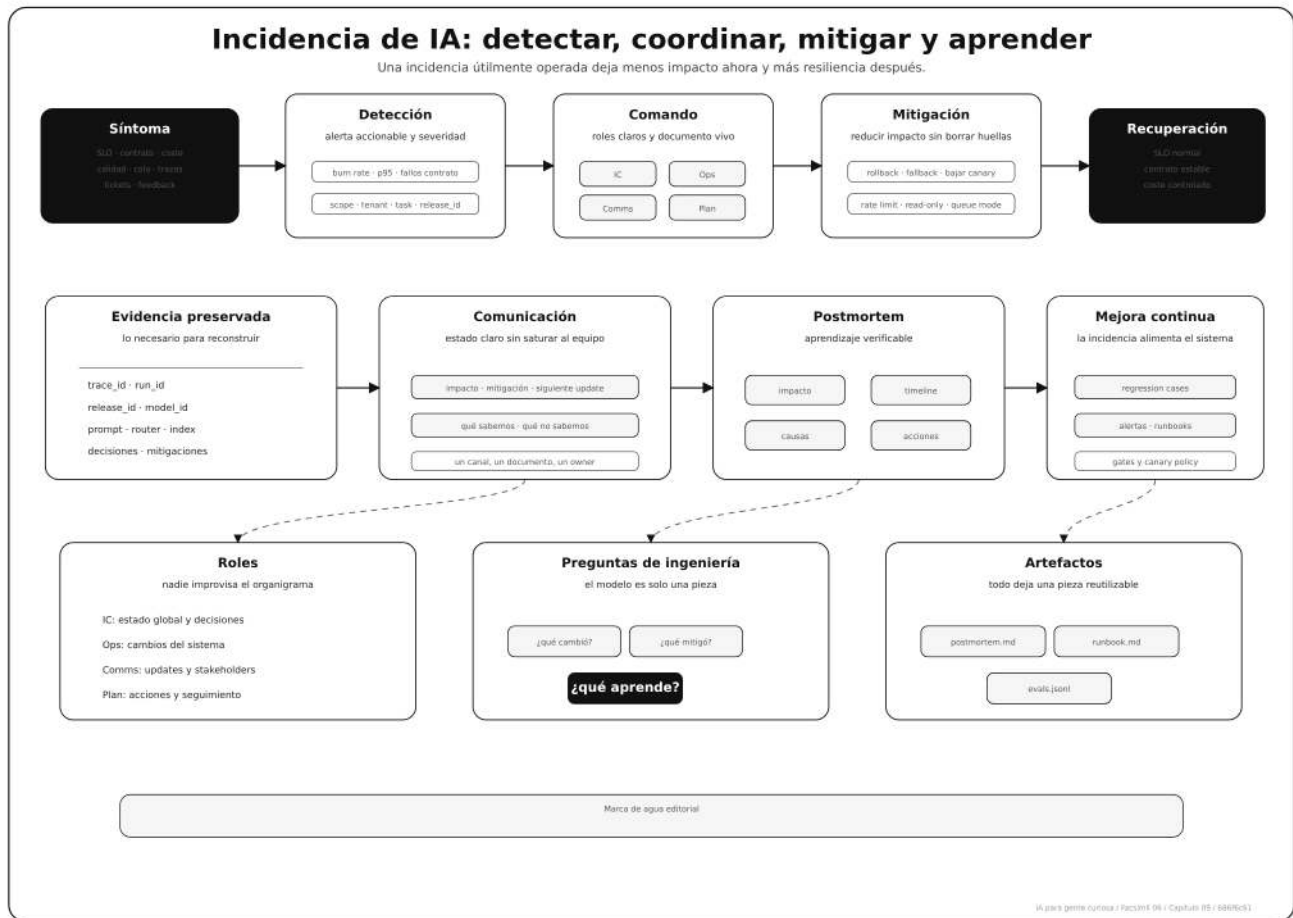
PIEZA	QUÉ GUARDAR	POR QUÉ
Entrada	Prompt de usuario o payload redactado.	Permite repetir el caso.
Contexto	IDs de chunks, documentos, versiones y hashes.	RAG cambia con el tiempo.
Configuración	Modelo, parámetros, prompt, router, tools.	Sin versiones no hay comparación.
Salida	Respuesta raw, salida validada y errores.	Permite medir contrato y calidad.
Traza	<code>trace_id</code> , spans y eventos.	Reconstruye latencia, retries y tools.
Decisión	Mitigación, rollback o handoff aplicado.	Explica por qué terminó así.

Plantilla `replay_case.json` :

```
{
  "case_id": "inc_support_rag_2026_05_28_case_001",
  "source": "incident",
  "task": "support_reply",
  "release_id": "support-rag@1.9.0-rc1",
  "input_hash": "sha256:...",
  "prompt_version": "prompt_v13",
  "model_id": "model_b",
  "route_catalog": "route_catalog@32",
  "rag_index": "rag_index_2026_06",
  "expected": {
    "contract_valid": true,
    "must_cite_source": true,
    "max_latency_ms": 4200
  },
  "observed": {
    "contract_valid": false,
    "latency_ms": 6200,
    "missing_source": true
  }
}
```

Este archivo es el puente entre postmortem y EvalOps. Si no hay replay, el aprendizaje se queda en memoria oral.

Anatomía visual de una incidencia de IA



Roles: separar trabajo para pensar mejor

Durante una incidencia, mezclar todos los roles crea ruido. Google SRE describe roles como comando, trabajo operativo, comunicación y planificación.¹³ En IA los adaptaría así:

ROL	RESPONSABILIDAD	NO DEBERÍA HACER
Comando de incidencia	Mantener estado global, severidad, decisiones y prioridades.	Tocar todos los mandos técnicos a la vez.
Operaciones	Ejecutar mitigaciones: rollback, fallback, rate limit, desactivar ruta.	Comunicar estimaciones no validadas.
Especialista IA	Analizar prompt, modelo, RAG, evals, router o tool.	Cambiar producción sin coordinar.
Comunicación	Explicar impacto, mitigación y próximo update.	Especular causas antes de evidencia.
Planificación	Registrar acciones, owners, follow-ups y postmortem.	Perderse en debugging de bajo nivel.

La regla de oro: **una persona coordina, pocas personas cambian producción, todas las decisiones quedan escritas.**

Mitigación: proteger primero, explicar después

La mitigación reduce impacto. No siempre corrige la causa profunda. Está bien. Durante una incidencia, primero evitamos que el daño operativo crezca.

SÍNTOMA	MITIGACIÓN INMEDIATA	EVIDENCIA A PRESERVAR
Contrato JSON falla.	Volver a prompt/schema anterior o activar validador estricto.	Salidas raw, <code>schema_version</code> , modelo, prompt.
Coste sube.	Bajar <code>max_tokens</code> , cambiar ruta, activar cache, pausar canary.	Tokens, route mix, provider, retries.
Latencia p95 sube.	Fallback a ruta ligera, limitar contexto, bajar batch, rate limit.	Queue depth, spans, p95 por fase.
RAG trae fuentes pobres.	Volver a índice anterior o bajar <code>top_k</code> .	Queries, chunks, index_version, reranker.
Cola de revisión caduca.	<code>queue_only_critical</code> , reasignar, respuesta de estado.	Backlog, owners, <code>needs_more_info_rate</code> .
Tool externa falla.	Modo solo lectura, fallback sin tool, abrir handoff.	Tool args, error tipado, trace_id.

El libro de SRE sobre sobrecarga recuerda que un sistema debe decidir qué trabajo aceptar, retrasar o rechazar para protegerse.¹⁴ En IA esto incluye rechazar generación cara, pausar rutas, pedir revisión o responder con estado claro.

Rollback, roll forward o contención

No toda incidencia pide la misma respuesta. Para un ingeniero de IA, la decisión importante es elegir la acción menos destructiva que reduzca impacto.

SITUACIÓN	ACCIÓN PREFERENTE	MOTIVO
Prompt nuevo rompe contrato.	Rollback de prompt.	Cambio reversible y localizado.
Índice RAG nuevo trae fuentes malas.	Rollback de índice o shadow de retrieval.	No hace falta tocar todo el servicio.
Modelo nuevo degrada p95.	Cambiar ruta o bajar porcentaje.	Mantienes producto mientras analizas.
Tool externa falla.	Contención: modo lectura o fallback sin tool.	Evitas efectos parciales.
Coste sube por contexto largo.	Límite de contexto, cache o ruta barata.	Reduce sangrado mientras investigas.
Causa no entendida y alcance crece.	Contención amplia y preservar evidencia.	Prioriza impacto y trazabilidad.
Fix pequeño probado.	Roll forward con canary nuevo.	Evita volver a estado antiguo si el parche es más seguro.

Regla práctica:

si el cambio causante es conocido y reversible -> rollback específico

si el causante es desconocido y el impacto crece -> contención

si el fix es pequeño, probado y observable -> roll forward controlado

La peor respuesta suele ser cambiar tres cosas a la vez y perder la capacidad de saber cuál ayudó.

Guardia operativa y handoff de turno

Una incidencia puede durar más que una persona. El cambio de turno también necesita contrato.

`oncall_handoff.md` debería incluir:

```
# Handoff de guardia

## Estado actual

- Incidencia:
- Severidad:
- Servicio:
- Comando:
- Último update:
- Próximo update:

## Qué sabemos

- Síntomas confirmados:
- Cambios recientes:
- Mitigaciones aplicadas:
- Estado de SL0:

## Qué no sabemos todavía

- Hipótesis abiertas:
- Evidencia pendiente:

## No tocar sin coordinar

- Rutas:
- Índices:
- Prompts:
- Proveedores:

## Siguiendo acción recomendada

Paso concreto para los próximos 15-30 minutos.
```

Esto parece pequeño, pero evita uno de los fallos más caros: que el turno nuevo repita investigación o revierta una mitigación que estaba funcionando.

Qué te llevas para poner en práctica

El alumno debería salir con un kit mínimo de incidencia para IA:

Inteligencia artificial para gente curiosa · 686f6c61 · CC BY 4.0

ARTEFACTO	PARA QUÉ SIRVE	QUÉ CONTIENE
<code>incident_events.jsonl</code>	Entrada reproducible.	Alertas, mitigaciones, cambios, recuperación y evidencias.
<code>incident_review.py</code>	Análisis ejecutable.	Severidad, timeline, MTTA, MTTR, mitigaciones y acciones.
<code>severity_matrix.yaml</code>	Criterio previo.	SEV1-SEV4 por disponibilidad, contrato, coste, calidad y alcance.
<code>incident_state.md</code>	Documento vivo.	Estado actual, owner, impacto, acciones y próximo update.
<code>postmortem.md</code>	Cierre de aprendizaje.	Impacto, línea temporal, causas contribuyentes y acciones.
<code>action_items.csv</code>	Seguimiento.	Acción, owner, fecha, verificación y estado.
<code>regression_cases.jsonl</code>	Vuelta a EvalOps.	Casos derivados de la incidencia.
<code>replay_case.json</code>	Reproducción mínima.	Entrada redactada, versiones, esperado y observado.
<code>oncall_handoff.md</code>	Cambio de turno.	Estado, hipótesis, mitigaciones y siguiente paso.
<code>incident_queries.promql</code>	Consultas listas.	Latencia, contrato, coste y errores por release.

Qué capítulos necesitas tener frescos

La práctica es usable al terminar este capítulo, pero no aparece de la nada. Cada pieza viene de algo trabajado antes:

PARTE DE LA PRÁCTICA	CAPÍTULO CONECTADO	QUÉ RECUPERA
<code>incident_events.jsonl</code> con eventos trazables.	<u>F6 · Capítulo 04</u>	Logs, métricas, trazas, <code>run_id</code> , <code>trace_id</code> , SLI y SLO.
Mitigaciones como bajar canary o volver índice.	<u>F6 · Capítulo 07</u>	Rollback, canary, kill switch y release progresiva.
Casos que vuelven a <code>regression_cases.jsonl</code> .	<u>F6 · Capítulo 06</u>	EvalOps, regresiones, gates y scorecard.
Handoff de guardia y revisión humana.	<u>F6 · Capítulo 08</u>	Colas, decisiones, evidence bundle y reanudación.
Routing, fallback y contención.	<u>F6 · Capítulo 05</u>	Rutas, presupuestos, fallback y control de coste.

Si alguien termina el capítulo 09 sin dominar todo lo anterior, aún puede ejecutar el script. Pero para defender la decisión técnica (por qué SEV2, por qué rollback de índice, por qué crear regresión) necesita volver a esas piezas.

Reto de capítulo: analizar una incidencia de IA

Escenario: un canary de prompt y un índice RAG nuevo coinciden con subida de latencia, coste y fallos de contrato. Tu tarea es reconstruir la incidencia desde eventos, calcular tiempos, proponer severidad y generar acciones.

Archivos:

```
mi-proyecto/
  ops/
    ai/
      incident_review.py
  data/
    incident_events.jsonl
  output/
    incident_report.json
```

Datos de entrada en `data/incident_events.jsonl` :

```
{
  "ts": "2026-05-28T13:40:00+00:00",
  "type": "change",
  "message": "canary prompt_v13 sube al 25%",
  "release_id": "support-rag@1.9.0-rc1",
  "task": "support_reply",
  "impact": 0
},
{
  "ts": "2026-05-28T13:43:00+00:00",
  "type": "alert",
  "message": "latency_p95_ms supera SL0",
  "metric": "latency_p95_ms",
  "value": 6200,
  "threshold": 4200,
  "task": "support_reply",
  "impact": 3
},
{
  "ts": "2026-05-28T13:47:00+00:00",
  "type": "alert",
  "message": "contract_fail_rate supera umbral",
  "metric": "contract_fail_rate",
  "value": 0.031,
  "threshold": 0.006,
  "task": "support_reply",
  "impact": 4
},
{
  "ts": "2026-05-28T13:49:00+00:00",
  "type": "ack",
  "message": "incidencia reconocida por ai-platform",
  "actor": "ai-platform-oncall",
  "impact": 0
},
{
  "ts": "2026-05-28T13:54:00+00:00",
  "type": "mitigation",
  "message": "candidate_weight baja de 25 a 5",
  "action": "reduce_canary",
  "impact": 0
},
{
  "ts": "2026-05-28T14:02:00+00:00",
  "type": "mitigation",
  "message": "rag_index vuelve a rag_index_2026_05",
  "action": "rollback_index",
  "impact": 0
},
{
  "ts": "2026-05-28T14:18:00+00:00",
  "type": "recovery",
  "message": "latency y contrato vuelven a SL0",
  "metric": "all",
  "value": 1,
  "threshold": 1,
  "impact": 0
},
{
  "ts": "2026-05-28T14:25:00+00:00",
  "type": "followup",
  "message": "crear regresión con salida JSON rota y chunk obsoleto",
  "owner": "evalops",
  "impact": 0
}
```

Comandos:

```
mkdir -p ops/ai data output
python ops/ai/incident_review.py --write
cat output/incident_report.json
```

Salida esperada:

CAMPO	VALOR ESPERADO
severity	SEV2
mtta_minutes	6
mttr_minutes	38
primary_symptoms	Latencia p95 y fallos de contrato.
mitigations	Bajar canary y volver índice RAG.
postmortem_required	true

Entrega mínima:

1. incident_report.json .
- Inteligencia artificial para gente curiosa · 686f6c61 · CC BY 4.0

2. Una propuesta de `postmortem.md`.
3. Dos acciones correctivas con owner y verificación.
4. Un caso nuevo para `regression_cases.jsonl`.
5. Una frase que explique qué señal debería haber alertado antes.
6. Un `replay_case.json` con versiones de prompt, modelo, router e índice.
7. Una decisión escrita: rollback, roll forward o contención, con motivo.

Comprobación de que la práctica está bien hecha:

COMPROBACIÓN	RESULTADO ESPERADO
Ejecuta sin dependencias externas.	Solo necesita Python estándar.
Lee <code>data/incident_events.jsonl</code> .	Si existe, usa esos eventos; si no existe, usa los eventos de ejemplo.
Escribe salida con <code>--write</code> .	Crea <code>output/incident_report.json</code> .
Calcula tiempos.	<code>mtta_minutes = 6</code> y <code>mttr_minutes = 38</code> .
Propone severidad.	<code>severity = SEV2</code> .
Conecta con EvalOps.	Genera acciones recomendadas y pide caso de regresión.
Es defendible en clase/equipo.	El alumno puede explicar síntoma, mitigación, replay y acción correctiva.

En el día a día

Las incidencias no son un «si pasa», son un «cuándo pasa», y la diferencia entre un equipo que aprende y uno que tropieza dos veces con la misma piedra está en lo que hace después. En el momento caliente, lo operable es tener una clasificación de severidad (no es lo mismo un SEV1 que tira el servicio que un SEV3 cosmético), un runbook que diga los primeros pasos sin tener que pensarlos a las tres de la mañana, y una decisión clara sobre si se mitiga, se hace rollback o se escala. Improvisar bajo presión es como casi todo el mundo gestiona su primera incidencia grave, y casi nadie lo recuerda con orgullo.

Después viene la parte que de verdad mejora el sistema: el postmortem. La escena sana no busca un culpable, busca la cadena de causas (qué falló, qué lo permitió, qué lo detectó tarde) y sale con acciones concretas y con dueño: un test de regresión nuevo, una alerta que faltaba, un límite que no estaba. La escena tóxica busca a quién señalar, y garantiza que el mismo fallo vuelva, porque nadie se atreve a contar lo que de verdad pasó.

Por qué debería importarte

Porque en operación no se trata de no fallar nunca (eso no existe en sistemas no deterministas conectados a proveedores que también fallan), sino de fallar cada vez de una forma nueva, porque las viejas ya se cerraron con una acción concreta. La gestión de incidencias y el postmortem sin culpa son la maquinaria que convierte cada fallo en una mejora permanente del sistema, en vez de en una herida que se reabre.

Y hay una razón muy personal para tomárselo en serio. La forma en que un equipo trata sus incidencias determina si la gente cuenta la verdad cuando algo sale mal o la esconde. Un postmortem sin culpa, con severidades claras y acciones con dueño, construye lo único que de verdad hace fiable a un sistema operado por personas: la confianza para decir «esto lo rompí yo, y así evitamos que vuelva a pasar».

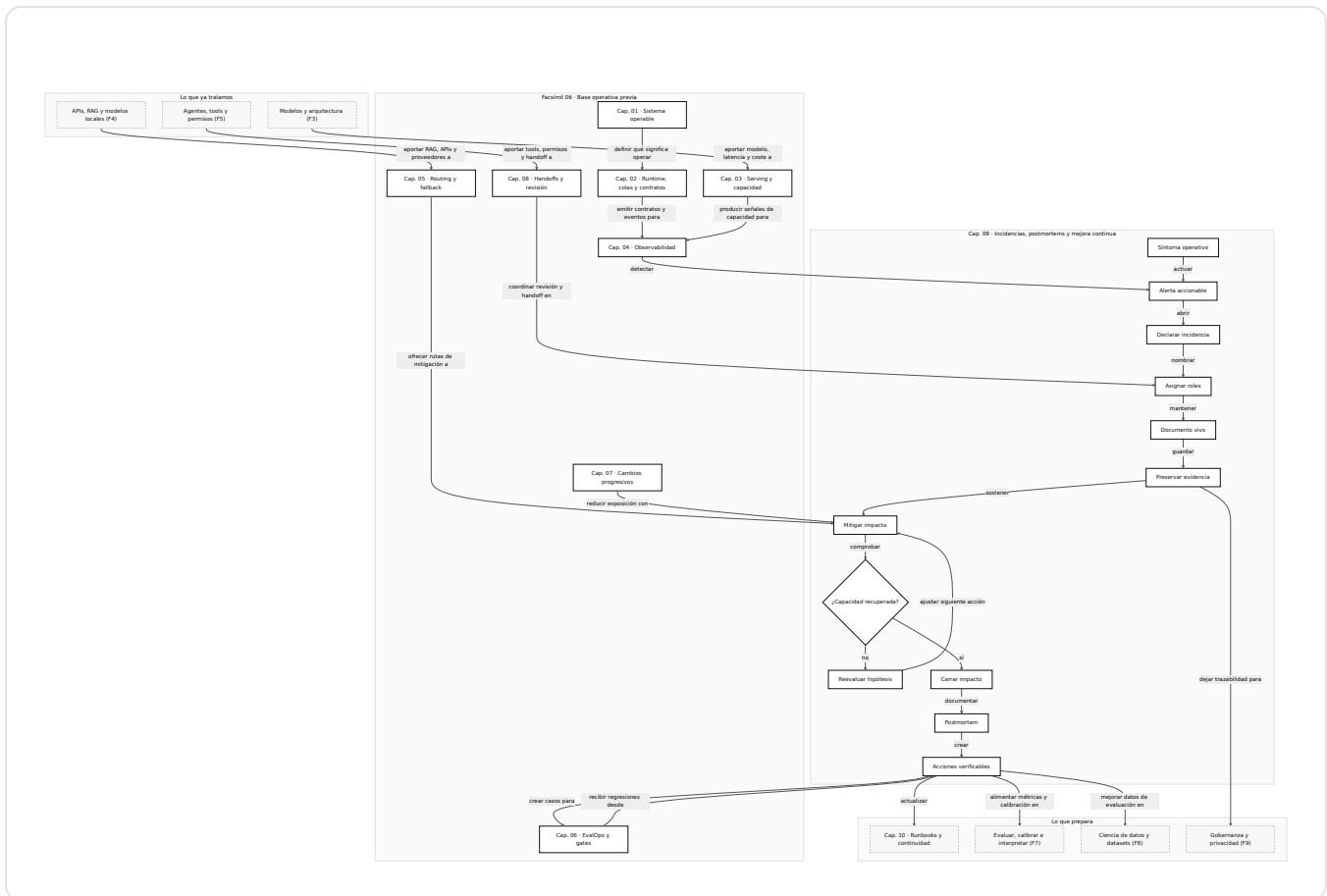
Dónde solía tropezar yo

Me costó aceptar que una incidencia bien operada puede parecer lenta al principio: parar, nombrar roles y escribir estado parece burocracia. Luego descubres que evita cambios cruzados, duplicidad de mensajes y pérdida de evidencia.

TROPIEZO	ANTÍDOTO
Investigar sin declarar.	Si afecta SLO o requiere otro equipo, declara pronto.
Cambiar varias cosas a la vez.	Una mitigación, un owner, un registro.
Cerrar al recuperar.	Cerrar solo cuando hay postmortem o decisión de no hacerlo.
Acciones vagas.	Cada acción con owner, fecha y verificación.
No alimentar EvalOps.	Cada incidencia debe crear o revisar casos de regresión.

La frase útil: **la recuperación devuelve el servicio; el postmortem mejora el sistema.**

Cómo encaja todo



Este mapa no intenta repetir toda la incidencia paso a paso. Su función es enseñar dónde vive el capítulo 09 dentro del facsímil: nace de observabilidad, routing, EvalOps, cambios progresivos y handoffs; convierte una degradación en evidencia y acciones; y deja preparado el capítulo 10, donde esas acciones se vuelven runbooks, continuidad y laboratorio de operación.

Relación con otros capítulos

CAPÍTULO	QUÉ APORTA AQUÍ
F6 · Capítulo 04	Señales, SLI, SLO, alertas y runbooks.
F6 · Capítulo 05	Fallback, presupuestos, rate limits y degradación controlada.
F6 · Capítulo 06	Casos de incidencia que vuelven a datasets y gates.
F6 · Capítulo 07	Rollback, kill switch y post-release review.
F6 · Capítulo 08	Handoffs, colas, SLO de revisión y decisiones humanas.

Amershi y colaboradores mostraron que los sistemas de ML en producción requieren procesos de ingeniería alrededor de datos, evaluación, monitorización y operación.¹⁵ El NIST AI RMF organiza el trabajo alrededor de gobernar, mapear, medir y gestionar sistemas de IA.¹⁶ Una incidencia bien cerrada toca los cuatro verbos: alguien gobierna, el equipo mapea impacto, mide señales y gestiona cambios.

Para entenderlo

Tres escenas:

SITUACIÓN	RESPUESTA DÉBIL	RESPUESTA OPERATIVA
Sube latencia p95 tras canary.	“Estamos mirando logs”.	SEV2, bajar canary, guardar trazas y abrir postmortem.
RAG cita documentos obsoletos.	Cambiar el índice sin dejar rastro.	Rollback de índice, conservar queries y crear regresión.
Cola de revisión caduca.	Pedir paciencia a soporte.	Activar backpressure y revisar política de handoff.

La madurez no está en no tener incidencias. Está en que cada una reduzca la próxima.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
Incidencia	Evento que degrada una propiedad operativa relevante.
Severidad	Clasificación por impacto, alcance, duración y urgencia.
MTTA	Tiempo hasta reconocer una incidencia.
MTTR	Tiempo hasta recuperar el servicio o capacidad.
Comando de incidencia	Rol que mantiene estado global y coordina respuesta.
Mitigación	Acción para reducir impacto antes de corregir causa profunda.
Postmortem	Documento de aprendizaje y acciones verificables.
Acción correctiva	Trabajo con owner y verificación que reduce recurrencia o impacto.

Antes de pasar página

Comprueba que puedes responder:

1. ¿Qué diferencia hay entre alerta, incidencia y postmortem?
2. ¿Qué dimensiones usarías para severidad en un sistema de IA?
3. ¿Qué roles separarías durante una incidencia?
4. ¿Por qué mitigar no siempre corrige la causa profunda?
5. ¿Qué datos necesitas para calcular MTTA y MTTR?
6. ¿Qué acción correctiva considerarías verificable?
7. ¿Cómo convertirías una incidencia en un caso de EvalOps?

En resumen

IDEA	PARA LLEVARTE
Una incidencia de IA puede ser calidad, contrato, coste, latencia o trazabilidad.	No mires solo caídas de servicio.
La respuesta necesita roles.	Comando, operaciones, comunicación y planificación reducen caos.
Mitigar protege, postmortem aprende.	Las dos cosas son necesarias y distintas.
Una acción sin verificación no cambia el sistema.	Owner, fecha, prueba y enlace al caso de regresión.

Para saber más

- Amershi, S. y otros *Software Engineering for Machine Learning: A Case Study*. <https://doi.org/10.1109/ICSE-SEIP.2019.00042>
- Baye, C. A. *Emergency Response*. <https://sre.google/sre-book/emergency-response/>
- Beyer, B. y otros *Handling Overload*. <https://sre.google/sre-book/handling-overload/>
- Dean, J. y Barroso, L. A. *The Tail at Scale*. <https://doi.org/10.1145/2408776.2408794>
- Ewaschuk, R. *Monitoring Distributed Systems*. <https://sre.google/sre-book/monitoring-distributed-systems/>
- Little, J. D. C. *A Proof for the Queuing Formula: $L = \lambda W$* . <https://doi.org/10.1287/opre.9.3.383>
- Lunney, J. y Lueder, S. *Postmortem Culture: Learning from Failure*. <https://sre.google/sre-book/postmortem-culture/>
- OpenTelemetry. *Logs*. <https://opentelemetry.io/docs/concepts/signals/logs/>
- OpenTelemetry. *Metrics*. <https://opentelemetry.io/docs/concepts/signals/metrics/>
- OpenTelemetry. *Traces*. <https://opentelemetry.io/docs/concepts/signals/traces/>
- Stribblehill, A. *Managing Incidents*. <https://sre.google/sre-book/managing-incidents/>
- Wilkinson, J. *Alerting on SLOs*. <https://sre.google/workbook/alerting-on-slos/>

Notas

1. Ewaschuk, R. (2016). *Monitoring Distributed Systems*. En B. Beyer, C. Jones, J. Petoff y N. R. Murphy (eds.), *Site Reliability Engineering*. <https://sre.google/sre-book/monitoring-distributed-systems/>. Consultado el 27 de mayo de 2026. ↵
2. Wilkinson, J. (2018). *Alerting on SLOs*. En B. Beyer, N. R. Murphy, D. Rensin, K. Kawahara y S. Thorne (eds.), *The Site Reliability Workbook*. <https://sre.google/workbook/alerting-on-slos/>. Consultado el 27 de mayo de 2026. ↵
3. Baye, C. A. (2016). *Emergency Response*. En *Site Reliability Engineering*. <https://sre.google/sre-book/emergency-response/>. Consultado el 28 de mayo de 2026. ↵
4. Stribblehill, A. (2016). *Managing Incidents*. En *Site Reliability Engineering*. <https://sre.google/sre-book/managing-incidents/>. Consultado el 28 de mayo de 2026. ↵
5. Lunney, J. y Lueder, S. (2016). *Postmortem Culture: Learning from Failure*. En *Site Reliability Engineering*. <https://sre.google/sre-book/postmortem-culture/>. Consultado el 28 de mayo de 2026. ↵
6. Keeney, R. L. y Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley. Formaliza la función de valor aditiva ponderada para decidir entre alternativas. ↵
7. Little, J. D. C. (1961). A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383> ↵
8. Trivedi, K. S. (2016). *Probability and Statistics with Reliability, Queuing, and Computer Science Applications* (2.ª ed.). Wiley. Para una tasa de fallo constante, el tiempo hasta el fallo es exponencial y la fiabilidad decae como $e^{-\lambda t}$. ↵
9. Pareto, V. (1896). *Cours d'économie politique*. La distribución de Pareto, $P(X > x) = (x_m/x)^\alpha$, modela fenómenos donde una minoría concentra la mayor parte del efecto; su aplicación a la calidad la popularizó J. M. Juran. ↵
10. Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps*. IT Revolution. Define y valida empíricamente las cuatro métricas clave de rendimiento de entrega, entre ellas la tasa de fallo de cambios y el tiempo de recuperación. ↵
11. Crow, L. H. (1975). *Reliability Analysis for Complex, Repairable Systems* (AMSAA Technical Report 138). U.S. Army Materiel Systems Analysis Activity. Modelo de crecimiento de fiabilidad de Crow-AMSAA basado en un proceso de potencia. ↵
12. Dean, J. y Barroso, L. A. (2013). The Tail at Scale. *Communications of the ACM*, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794> ↵
13. Stribblehill, 2016. ↵

- .4. Beyer, B., Jones, C., Petoff, J. y Murphy, N. R. (2016). *Handling Overload*. En *Site Reliability Engineering*. <https://sre.google/sre-book/handling-overload/>. Consultado el 27 de mayo de 2026. ↵
- .5. Amershi, S. y otros (2019). *Software Engineering for Machine Learning: A Case Study*. Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, 291-300. <https://doi.org/10.1109/ICSE-SEIP.2019.00042> ↵
- .6. Tabassi, E. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. NIST AI 100-1. <https://doi.org/10.6028/NIST.AI.100-1> ↵

Capítulo 10: Runbooks y continuidad de operación

Qué deberías poder hacer al terminar

Este capítulo cierra el facsímil 6. Ya hemos hablado de prototipos que pasan a sistemas, runtime, colas, serving, observabilidad, routing, EvalOps, cambios progresivos, revisión humana e incidencias. Ahora falta la pieza que separa un equipo que “sabe mucho” de un equipo que puede operar bajo presión: **runbooks, continuidad y práctica reproducible**.

Al terminar, deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Escribir un runbook operable.	Incluyes señales, consultas, decisión, comando, rollback, verificación y criterio de salida.
Definir continuidad.	Separas RTO, RPO, modo degradado, recuperación y pérdida aceptable de estado.
Revisar readiness.	Compruebas si SLO, observabilidad, EvalOps, rollback, handoff e incidencia están listos.
Diseñar un ensayo operativo.	Practicas una degradación controlada sin improvisar datos ni decisiones.
Reunir una práctica reproducible.	Dejas archivos, comandos, salidas esperadas y criterios de aceptación.
Cerrar el facsímil con criterio operativo.	Sabes juntar todo lo visto en construcción y operación en una decisión de readiness defendible.

La idea central: **un sistema de IA no está listo cuando responde; está listo cuando alguien puede recuperarlo, explicarlo, medirlo y mejorarlo**.

El cierre que convierte capítulos en operación

Imagina que mañana publicas un asistente RAG para soporte. Tiene tests, trazas, canary y un panel de coste. Parece suficiente. Pero llega una mañana mala: el proveedor principal degrada latencia, el índice nuevo trae documentos equivocados, la cola de revisión crece y

una release candidata empezó a circular por una parte del tráfico.

La pregunta no es “¿quién sabe de esto?”. La pregunta operativa es:

¿Dónde está escrito qué mirar, qué cambiar, quién decide, cómo se vuelve a estado conocido y cómo sabemos que hemos recuperado?

Esa respuesta vive en runbooks. No como documento muerto, sino como contrato práctico entre ingeniería, producto, soporte y operación.

Qué no es un runbook

Un runbook no es una colección de notas sueltas. Tampoco es un tutorial para leer tranquilamente. Y desde luego no es una página que diga “mirar dashboard” o “reiniciar servicio” sin explicar cuándo, por qué y cómo comprobar el resultado.

Un runbook pobre suele tener tres síntomas:

SÍNTOMA	QUÉ OCURRE EN PRODUCCIÓN
Dice qué hacer, pero no cuándo.	Dos personas ejecutan acciones distintas ante el mismo síntoma.
Tiene comandos sin verificación.	Nadie sabe si el cambio arregló algo o solo cambió la gráfica.
No versiona contexto.	No se sabe qué modelo, prompt, índice, router o contrato estaba activo.
No define dueño.	La decisión se queda flotando en un canal.
No tiene criterio de salida.	La operación parece cerrada antes de recuperar SLO, contrato y evidencia.

Un runbook útil no quita pensamiento. Quita ruido. Deja espacio mental para lo difícil: interpretar señales, elegir mitigación y aprender.

Qué sí es un runbook operativo

Para este facsímil, un runbook operativo es:

Un procedimiento versionado que convierte una situación reconocible en señales, decisiones, acciones, verificación y aprendizaje.

Un runbook se compone de estas piezas. No es una ecuación de la literatura, es una lista de ingeniería:

SÍMBOL O	SIGNIFICADO	EJEMPLO
<i>S</i>	Señales de entrada.	Latencia p95, fallos de contrato, coste p95, cola, feedback.
<i>Q</i>	Consultas preparadas.	PromQL, SQL, enlace a trazas, panel de canary.
<i>D</i>	Decisión que debe tomarse.	Rollback, fallback, modo degradado, pausa de canary.
<i>A</i>	Acciones concretas.	Comando, cambio de flag, ruta alternativa, handoff.
<i>V</i>	Verificación.	SLO recuperado, contrato válido, coste normal, traza completa.
<i>E</i>	Evidencia que se preserva.	<code>trace_id</code> , <code>release_id</code> , <code>model_id</code> , <code>prompt_version</code> , <code>index_version</code> .
<i>C</i>	Criterio de cierre.	Condiciones para dar por recuperada la capacidad.

Si falta *Q*, la persona buscará a mano. Si falta *V*, ejecutará sin saber si ayudó. Si falta *C*, cerrará por cansancio, no por evidencia.

Fecha de corte del estado del arte

Fecha de corte: 28 de mayo de 2026.

Fuentes consultadas: capítulos de Google SRE sobre monitorización, respuesta de emergencia, gestión de incidencias, postmortems y sobrecarga; SRE Workbook sobre alerting basado en SLOs; AWS Builders Library sobre timeouts, retries y backoff; OpenTelemetry sobre trazas, logs y métricas; NIST AI RMF; y trabajos de ingeniería de ML.

Google SRE insiste en que las alertas que interrumpen a personas deben ser accionables, relevantes para el usuario y fáciles de interpretar.¹ El SRE Workbook recomienda alertar usando consumo de presupuesto de error para conectar síntomas con SLOs, en vez de reaccionar a umbrales aislados.²

La preparación también forma parte del sistema. Google SRE trata la respuesta de emergencia como una habilidad que se entrena antes de necesitarla.³ En gestión de incidencias separa roles para coordinar decisiones, trabajo operativo, comunicación y planificación.⁴ Después, los postmortems deben convertir lo ocurrido en acciones verificables y aprendizaje del sistema.⁵

Lo estable no es la herramienta concreta de guardia, CI, dashboard o proveedor. Lo estable es el circuito: observar, decidir, actuar, verificar, documentar, practicar y mejorar.

Continuidad: RTO, RPO y modo degradado

Continuidad no significa que nada falle. Significa que el sistema tiene caminos pensados para seguir prestando lo esencial cuando una parte se degrada.

Dos siglas son importantes:

SIG LA	QUÉ MIDE	EJEMPLO EN IA
RTO	Tiempo objetivo para recuperar una capacidad.	“El asistente de soporte debe volver a responder en menos de 30 minutos”.
RPO	Pérdida máxima aceptable de estado o datos medida como tiempo.	“Podemos perder como máximo 10 minutos de cola o memoria operativa”.

Si $t_{recuperacion}$ es el tiempo real hasta recuperar y RTO el objetivo:

$$t_{recuperacion} \leq RTO$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$t_{recuperacion}$	Minutos desde inicio hasta capacidad recuperada.	24 minutos.
RTO	Máximo aceptable acordado.	30 minutos.

Si el sistema recibe λ trabajos por minuto y queda degradado W minutos, el backlog esperado se aproxima como:

$$L = \lambda W$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
L	Trabajos acumulados.	180 solicitudes.
λ	Llegadas por minuto.	6 solicitudes/minuto.
W	Minutos degradado.	30 minutos.

Esta es una forma directa de usar la ley de Little para operación.⁶ Si la cola crece, no basta con desear que baje: o reducimos entrada, o subimos capacidad, o aceptamos más espera.

Cuánta redundancia de verdad necesitas: el sistema k-de-n. Tener tres réplicas no significa «tolero que se caigan dos» a menos que con una sola sigas dando servicio. Un sistema k-de-n funciona si al menos k de sus n componentes están vivos, y su fiabilidad se calcula sumando las combinaciones binomiales.⁷

$$R_{k/n} = \sum_{i=k}^n \binom{n}{i} r^i (1-r)^{n-i}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
n	Número total de réplicas.	3
k	Réplicas que deben estar vivas para dar servicio.	2
r	Fiabilidad de cada réplica.	0,95
$R_{k/n}$	Fiabilidad del conjunto.	$\approx 0,993$ para 2-de-3.

En palabras: «2-de-3» (necesitas dos de tres) no es lo mismo que «1-de-3» (te basta una): con réplicas al 95%, exigir dos vivas da 99,3%, pero bastar con una da 99,99%. Diseñar continuidad es elegir ese k a conciencia, porque define cuántas caídas simultáneas sobrevives, no cuántas máquinas tienes.

Cuánta carga admites cuando vas degradado: la fórmula de Erlang B. En modo degradado tienes menos capacidad y debes decidir a quién admites y a quién rechazas. La fórmula de pérdida de Erlang B da la probabilidad de que una petición se rechace por no haber servidor libre, sin cola de espera.⁸

$$B(c, a) = \frac{\frac{a^c}{c!}}{\sum_{k=0}^c \frac{a^k}{k!}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
a	Carga ofrecida en erlangs.	8
c	Servidores disponibles en modo degradado.	6
$B(c, a)$	Probabilidad de que una petición se rechace.	$\approx 0,12$.

En palabras: con 8 erlangs de carga y solo 6 servidores, alrededor del 12% de las peticiones se rechazan. Saber este número por adelantado convierte la degradación en una decisión («admito lo prioritario, rechazo el resto con un mensaje claro») en lugar de un colapso silencioso donde todo se ralentiza hasta caer.

¿Cumplirás tu objetivo de recuperación? La probabilidad de cumplir el RTO. El RTO es una promesa: «me recupero en menos de X». Si el tiempo de recuperación es exponencial con media MTTR, la probabilidad de cumplir esa promesa tiene una forma cerrada que conviene mirar antes de comprometerla.⁹

$$P(T_{\text{rec}} \leq \text{RTO}) = 1 - e^{-\text{RTO}/\text{MTTR}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
RTO	<i>Recovery Time Objective</i> : plazo prometido.	1 hora.
MTTR	Tiempo medio real de recuperación.	45 minutos.
$P(T_{\text{rec}} \leq \text{RTO})$	Probabilidad de cumplir el RTO.	$1 - e^{-60/45} \approx 0,74$.

En palabras: si tu MTTR real es de 45 minutos, cumplir un RTO de una hora solo ocurre el 74% de las veces, no siempre. Para cumplirlo el 95% de las veces necesitarías un RTO de unas tres veces el MTTR. Esta cuenta evita firmar objetivos de recuperación que tu media de reparación no puede sostener.

¿Cuántos datos puedes perder? La pérdida esperada frente al RPO. El RPO marca cuántos datos toleras perder, y se cumple con la frecuencia de las copias. Si el fallo puede caer en cualquier momento entre dos copias, la pérdida media es la mitad del intervalo y la peor, el intervalo entero.¹⁰

$$\mathbb{E}[L] = \frac{T_{\text{copia}}}{2}, \quad L_{\text{max}} = T_{\text{copia}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
T_{copia}	Intervalo entre copias de seguridad.	60 minutos.
$\mathbb{E}[L]$	Pérdida media de datos ante un fallo.	30 minutos.
L_{max}	Pérdida máxima (peor caso).	60 minutos.

En palabras: copias cada hora significan perder de media media hora de datos y, en el peor caso, una hora entera. Si tu RPO es de 15 minutos, necesitas copias cada 15 minutos como mucho, no cada hora. La continuidad no se promete, se dimensiona: la frecuencia de copia se deduce del RPO, no al revés.

Runbooks por capas

Un sistema de IA se opera por capas. El runbook debe ayudar a distinguirlas:

CAPA	PREGUNTA	SEÑALES	ACCIÓN TÍPICA
Producto	¿El usuario completa la tarea?	Conversión, aceptación, tickets, abandono.	Modo degradado, estado claro, revisión.
Contrato	¿La salida sigue siendo compatible?	JSON inválido, campos ausentes, enum fuera de catálogo.	Schema anterior, validador estricto, rollback de prompt.
Modelo	¿Cambió calidad, latencia o coste?	<code>model_id</code> , p95, coste, flake rate, errores.	Cambiar ruta, reducir esfuerzo, fallback.
RAG	¿El contexto recuperado sirve?	<code>index_version</code> , chunks, citas, recall manual.	Volver índice, bajar <code>top_k</code> , pausar reranker.
Runtime	¿El sistema procesa bien?	Queue depth, timeouts, goodput, workers.	Rate limit, escalar, reducir contexto.
Handoff	¿La revisión avanza?	Backlog, aging, SLA, owner.	Priorizar, reasignar, limitar entrada.
EvalOps	¿La release es defendible?	Scorecard, regresiones, canary, gates.	Bloquear, revertir, añadir caso.
Observabilidad	¿Podemos reconstruir?	Trazas, logs, métricas, atributos.	Subir sampling, añadir atributos obligatorios.

OpenTelemetry separa señales como trazas, logs y métricas para reconstruir comportamiento desde ángulos distintos.¹¹¹²¹³ En IA, esas señales deben llevar atributos como `task`, `model_id`, `prompt_version`, `route_id`, `release_id`, `trace_id` e `index_version`.

Readiness: la pregunta antes de publicar

Una revisión de readiness responde una pregunta incómoda:

Si esto falla hoy a las 03:17, ¿tenemos lo necesario para entenderlo, limitarlo y recuperarlo?

Esa pregunta se convierte en una puntuación de preparación, una media ponderada de los criterios:

$$R = \frac{\sum_{i=1}^n w_i c_i}{\sum_{i=1}^n w_i}$$

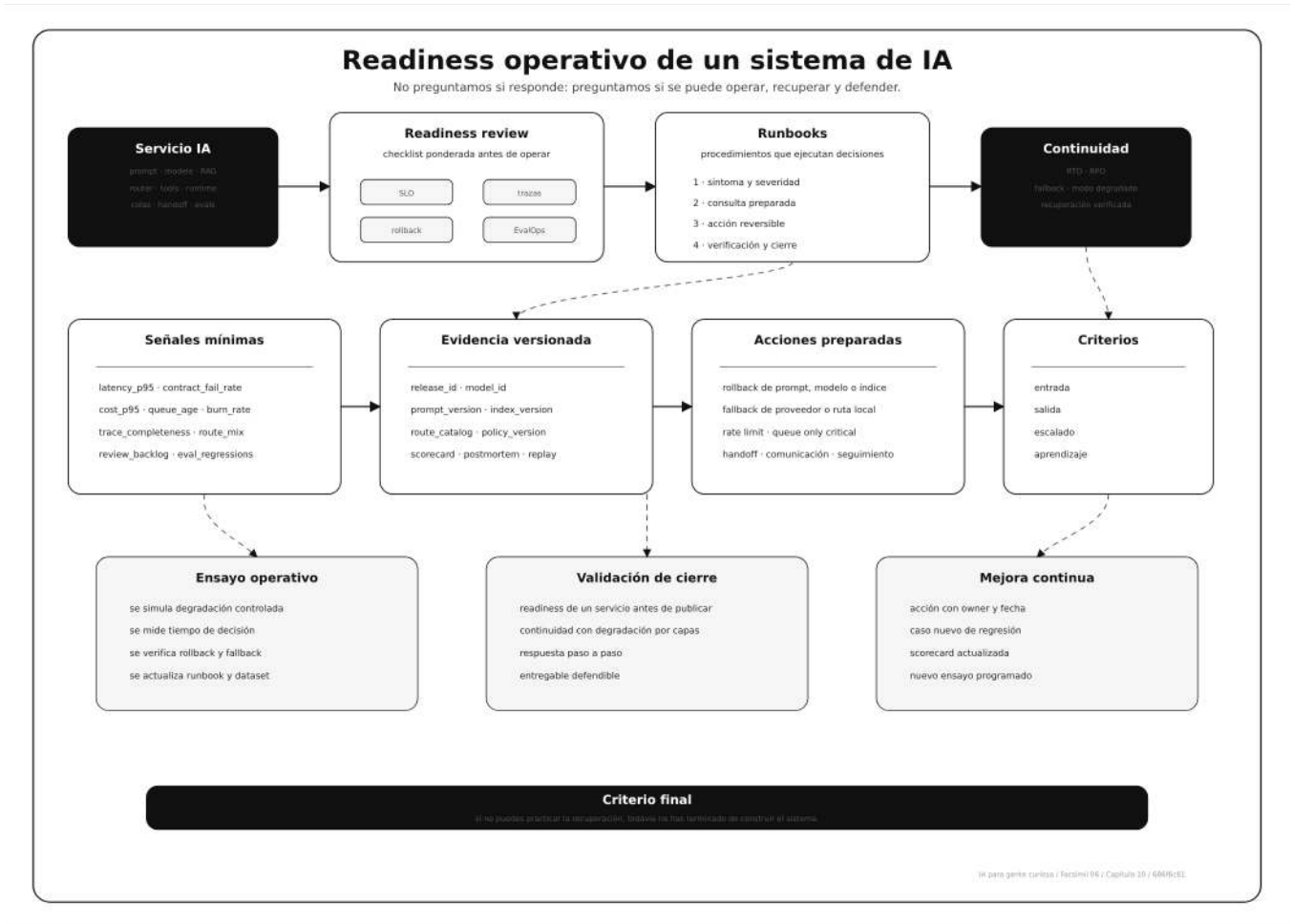
SÍMBOL O	SIGNIFICADO	EJEMPLO
R	Puntuación de readiness entre 0 y 1.	0,84.
w_i	Peso de la comprobación i .	2 para rollback, 1 para documentación auxiliar.
c_i	Resultado de la comprobación: 1 si pasa, 0 si falta.	<code>trace_id</code> presente: 1.
n	Número de comprobaciones.	26.

Una puntuación alta no garantiza que nada vaya mal. Indica algo más modesto y más útil: que el sistema tiene piezas para operar con método.

BANDA	DECISIÓN
$R \geq 0,90$	Listo para operar con seguimiento normal.
$0,75 \leq R < 0,90$	Puede avanzar con condiciones escritas.
$R < 0,75$	No publicaría sin corregir huecos.

El NIST AI RMF propone gestionar riesgos de IA mediante funciones como gobernar, mapear, medir y gestionar.¹⁴ En este capítulo lo llevamos a ingeniería: no basta reconocer riesgos; hay que convertirlos en checks, runbooks, gates y evidencias.

Anatomía visual del cierre operativo



Ensayos operativos: practicar antes de necesitarlo

Un ensayo operativo es una prueba controlada. No buscamos “romper cosas”; buscamos comprobar si el equipo sabe operar una situación concreta.

Ejemplos de ensayos útiles:

ENSAYO	QUÉ SE PRACTICA	CRITERIO DE ÉXITO
Proveedor lento	Fallback, routing, timeouts y comunicación.	p95 vuelve a SLO y coste no se dispara.
Índice RAG equivocado	Rollback de índice y replay.	Recuperas fuentes correctas y generas regresión.
Fallo de contrato JSON	Validador, schema anterior y gate.	La salida vuelve a ser parseable.
Cola de revisión llena	Priorización, modo degradado y SLA.	Casos críticos no caducan.
Canary con peor calidad	Gate online y rollback.	Se corta candidate sin afectar todo el tráfico.
Trazas incompletas	Sampling temporal y atributos obligatorios.	Puedes reconstruir runs relevantes.

La AWS Builders Library recomienda tratar timeouts, retries y backoff como diseño explícito porque los reintentos pueden amplificar carga si se usan sin cuidado.¹⁵ En IA esto se nota enseguida: un retry sobre una ruta cara, una tool lenta o un contexto largo puede convertir una degradación pequeña en coste y cola.

En el día a día

Los runbooks aparecen el día en que la persona que sabía cómo arreglar algo no está, y todo el conocimiento de cómo operar el sistema vivía en su cabeza. Un runbook es lo contrario de esa fragilidad: un procedimiento escrito, probado y a mano para las situaciones que se repiten (un proveedor caído, una cola saturada, una regresión que dispara el gate). La prueba de un buen runbook es brutal y honesta: ¿podría seguirlo alguien del equipo que no diseñó el sistema, a las tres de la mañana, sin llamar a nadie? Si la respuesta es no, todavía no es un runbook, es una nota para iniciados.

La continuidad es la versión grande de la misma idea. No basta con saber arreglar fallos pequeños; hay que haber pensado qué pasa si cae el proveedor principal, si se corrompe el índice, si se va la persona clave. Tener planeado el degradado (a qué se baja, qué se sacrifica

primero, cómo se vuelve) es lo que convierte una catástrofe potencial en una molestia gestionable. Es el equivalente operativo de saber dónde está la salida de emergencia antes de que haya humo.

Por qué debería importarte

Porque un sistema que solo sabe operar su autor no es un sistema operable, es una dependencia personal disfrazada. Los runbooks y los planes de continuidad son lo que reparte ese conocimiento en el equipo y lo que permite que el sistema sobreviva a vacaciones, bajas, relevos y al peor día imaginable. Convierten «esto solo lo sabe arreglar Marta» en «esto lo puede arreglar cualquiera siguiendo el runbook», que es la única forma de operar algo durante años sin quemar a las personas.

Y hay un cierre que vale la pena nombrar, porque este capítulo cierra el facsímil. Todo lo que hemos construido (contratos, trazas, gates, routing, handoffs, postmortems) culmina aquí: en la capacidad de operar un sistema de IA de forma que no dependa de héroes, que se pueda transferir, auditar y sostener. Eso es, al final, lo que separa un proyecto de IA que dura de uno que brilla un trimestre y se apaga cuando se va quien lo entendía.

Dónde solía tropezar yo

TROIEZO	POR QUÉ PASA	ANTÍDOTO
Escribir runbooks como notas	Parece suficiente cuando todo va bien.	Convertir cada nota en señal, decisión, acción y verificación.
No practicar recuperación	El equipo confía en que sabrá hacerlo.	Programar ensayos operativos con datos y roles.
Mezclar RTO con deseo	Se promete recuperar rápido sin capacidad real.	Medir cola, tiempo de decisión y comandos disponibles.
Olvidar RPO	Se piensa solo en servicio, no en estado perdido.	Definir qué memoria, cola o evento se puede reconstruir.
No conectar postmortem con EvalOps	El aprendizaje queda en un documento.	Cada incidencia relevante crea un caso de regresión o un gate.
Hacer checklist sin pesos	Todo parece igual de importante.	Ponderar rollback, trazas, SLO y continuidad más que detalles secundarios.

Cuadernos para practicar

Has operado sistemas de IA a lo largo del facsímil; estos cuadernos te dejan instrumentarlos. Son *notebooks* que se abren en Google Colab —gratis, en el navegador— o te puedes descargar. Cada uno, explicado paso a paso, con salidas reales, y anclado al capítulo del que sale.

Observabilidad: la caja negra que sí puedes mirar

Qué prácticas: registrar cada petición y construir el panel que dice si un servicio va bien. **Dónde encaja:** capítulo 4 (observabilidad: logs, métricas, trazas y costes). **Qué necesitas:** un navegador. Corre en CPU; sin claves.

Pones un servicio «en producción» (simulado), registras mil peticiones con su latencia, sus tokens y su coste, y montas el panel que mirarías cada mañana. Ahí descubres por qué la media engaña: la latencia media sale en **589 ms**, pero el percentil 95 —el 5% que peor lo pasa— está en **2.285 ms**. Si solo vigilas la media, no ves el problema hasta que esos usuarios se van. Terminas definiendo una alerta con umbrales: cuándo merece la pena despertar a alguien.

Un sistema sin observabilidad es una caja negra que solo te avisa de sus fallos por boca de los usuarios enfadados.

► [Abrir en Google Colab](#)

↓ [Descargar el cuaderno \(.ipynb\)](#)

Routing, fallback y presupuesto: el modelo adecuado para cada tarea

Qué prácticas: enrutar cada tarea al modelo justo y poner topes de coste, sin perder calidad. **Dónde encaja:** capítulo 5 (routing, fallback y presupuestos por tarea). **Qué necesitas:** un navegador. Corre en CPU; sin claves (se simulan los modelos).

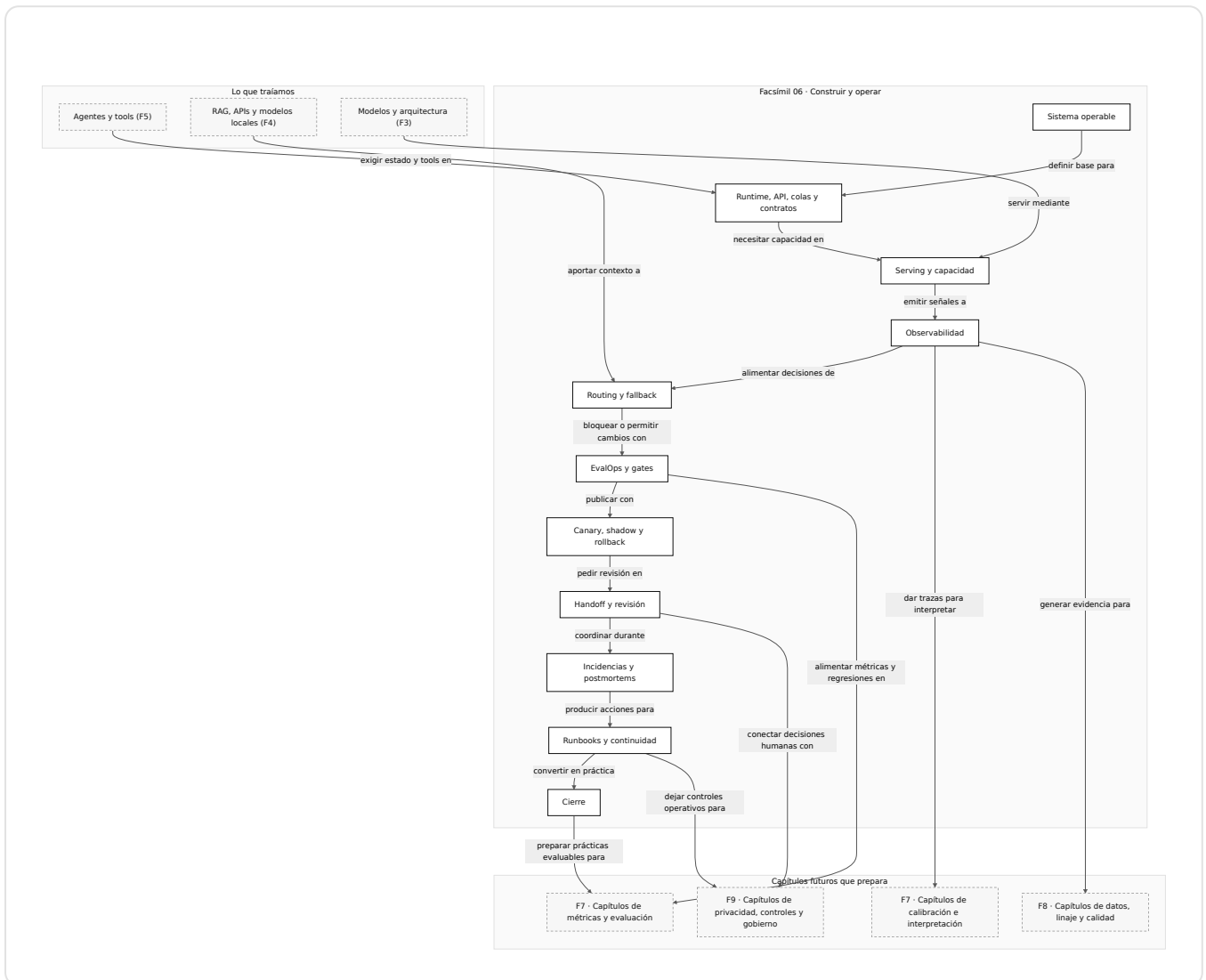
Dejas de mandar todo al modelo caro. Montas un router que envía las tareas fáciles a un modelo barato y solo escala al caro cuando el barato responde con **poca confianza** —no cuando «sabes» que falló, que en la realidad no lo sabes—. El resultado frente a «caro para todo»: recortas un **54% de la factura** perdiendo solo **algo más de un punto** de acierto, porque la mayoría de las tareas no necesitaban el modelo potente. Y añades un tope de presupuesto por tarea para que un caso testarudo no se lleve el gasto de cien.

Usar el modelo más potente para todo es cómodo y ruinoso. La diferencia entre un sistema viable y uno que quema dinero suele estar en enrutar bien.

► **Abrir en Google Colab**

↓ **Descargar el cuaderno (.ipynb)**

Cómo encaja todo



El mapa resume el motivo de este capítulo: runbooks y continuidad no son una sección administrativa. Son el punto donde todos los capítulos anteriores se convierten en práctica operable. Como los facsímiles 7, 8 y 9 todavía están planificados, no inventamos títulos cerrados de capítulos; dejamos nodos de capítulo futuro por tema, para que la relación quede clara sin fingir una estructura que aún puede cambiar.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN ÚTIL
Runbook	Procedimiento operativo versionado para diagnosticar, decidir, actuar y verificar.
Readiness review	Comprobación previa que dice si un servicio está listo para ser operado.
RTO	Tiempo objetivo máximo para recuperar una capacidad.
RPO	Pérdida máxima aceptable de datos o estado medida como tiempo.
Modo degradado	Servicio reducido que conserva lo esencial mientras se recupera lo completo.
Ensayo operativo	Prueba controlada para practicar recuperación, roles, señales y decisiones.
Criterio de entrada	Señal que indica cuándo activar un runbook.
Criterio de salida	Condición verificable para cerrar una operación.
Paquete de evidencias	Trazas, métricas, logs, versiones y decisiones que reconstruyen lo ocurrido.
Last known good	Última versión conocida que funcionaba de forma aceptable.
Gate	Regla verificable que deja avanzar o bloquea un cambio.
Acción correctiva	Cambio con owner y verificación que reduce repetición de un problema.

Antes de pasar página

Responde estas preguntas antes de cerrar el facsímil:

PREGUNTA	VUELVE A
¿Puedes explicar qué diferencia hay entre runbook, postmortem y checklist?	Qué sí es un runbook operativo .
¿Tu servicio tiene RTO y RPO escritos?	Continuidad: RT0, RP0 y modo degradado .
¿Sabes qué ruta usar si el proveedor principal degrada latencia?	<u>Capítulo 05.</u>
¿Tienes consultas preparadas para latencia, contrato, coste y cola?	<u>Capítulo 04.</u>
¿Puedes volver a una versión conocida sin tocar tres cosas a la vez?	<u>Capítulo 07.</u>
¿Una incidencia crea casos nuevos de regresión?	<u>Capítulo 06.</u>
¿Tu cola de revisión tiene SLO y modo degradado?	<u>Capítulo 08.</u>
¿Tu postmortem deja acciones con owner y verificación?	<u>Capítulo 09.</u>
¿Puedes ejecutar el verificador de readiness y explicar el resultado?	Practícalo en el cuaderno del facsímil.

En resumen

IDEA FUERZA	QUÉ TE LLEVAS
Operar es diseñar recuperación.	No basta con publicar; hay que saber volver, limitar y explicar.
El runbook debe ser ejecutable mentalmente.	Señal, consulta, decisión, acción, verificación y cierre.
Continuidad tiene números.	RTO, RPO, cola, capacidad y modo degradado.
Readiness es evidencia, no optimismo.	SLO, trazas, rollback, EvalOps, handoff e incidencia deben existir.
La práctica importa.	Ensayar antes reduce improvisación cuando aparece una incidencia real.
Operar bien cierra el facsímil.	Construir, justificar y adaptar lo aprendido es lo que demuestra el criterio operativo.

Recursos para seguir: leer, construir y experimentar

Construir y operar va de lo que pasa después de la demo: runtime, serving, observabilidad, gates de release, incidencias y continuidad. No se aprende en un playground, se aprende montando y vigilando sistemas.

Para experimentar. Aquí no hay un botón mágico en el navegador, pero sí piezas reales que puedes levantar en tu máquina para tocar cada concepto: Grafana y Prometheus para métricas y paneles, una cola con Redis para entender backpressure, y herramientas de observabilidad de LLM como Langfuse o LangSmith para ver trazas, coste y latencia de cada run. Montar estas piezas en tu máquina te deja simular la operación completa y practicar las decisiones sin necesidad de producción.

Para construir. El stack operativo típico: OpenTelemetry para instrumentar trazas y métricas; Prometheus y Grafana para monitorizar; vLLM o Triton para servir modelos con batching; y contenedores con Docker para reproducibilidad. Para cambios seguros, patrones de shadow, canary y rollback, que el facsículo explica y conviene practicar en tu propio entorno.

Para leer. La referencia clásica de operación es el *Site Reliability Engineering* de Google, cuyas ideas (SLI, SLO, presupuesto de error, postmortems sin culpa) recorren todo el facsículo. Cada capítulo enlaza sus fuentes en «Para saber más». Cerrar la distancia entre «funciona en mi portátil» y «aguanta en producción con quien lo opere» es justo lo que persigue este facsículo.

Para saber más

Amazon Web Services. (2026). *Timeouts, Retries, and Backoff with Jitter*. AWS Builders Library. <https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/>

Amershi, S. y otros (2019). *Software Engineering for Machine Learning: A Case Study*. *International Conference on Software Engineering: Software Engineering in Practice*, 291-300. <https://doi.org/10.1109/ICSE-SEIP.2019.00042>

Baye, C. A. (2016). *Emergency Response*. En *Site Reliability Engineering*. <https://sre.google/sre-book/emergency-response/>

Dean, J. y Barroso, L. A. (2013). *The Tail at Scale*. *Communications of the ACM*, 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>

Ewaschuk, R. (2016). *Monitoring Distributed Systems*. En *Site Reliability Engineering*. <https://sre.google/sre-book/monitoring-distributed-systems/>

GitHub. (2026). *Workflow Syntax for GitHub Actions*. <https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax>

- Little, J. D. C. (1961). A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>
- Lunney, J. y Lueder, S. (2016). *Postmortem Culture: Learning from Failure*. En *Site Reliability Engineering*. <https://sre.google/sre-book/postmortem-culture/>
- OpenTelemetry. (2026). *Logs*. <https://opentelemetry.io/docs/concepts/signals/logs/>
- OpenTelemetry. (2026). *Metrics*. <https://opentelemetry.io/docs/concepts/signals/metrics/>
- OpenTelemetry. (2026). *Traces*. <https://opentelemetry.io/docs/concepts/signals/traces/>
- Stribblehill, A. (2016). *Managing Incidents*. En *Site Reliability Engineering*. <https://sre.google/sre-book/managing-incidents/>
- Tabassi, E. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. National Institute of Standards and Technology, NIST AI 100-1. <https://doi.org/10.6028/NIST.AI.100-1>
- Wilkinson, J. (2018). *Alerting on SLOs*. En *The Site Reliability Workbook*. <https://sre.google/workbook/alerting-on-slos/>
-

Notas

1. Ewaschuk, R. (2016). *Monitoring Distributed Systems*. En B. Beyer, C. Jones, J. Petoff y N. R. Murphy (eds.), *Site Reliability Engineering*. <https://sre.google/sre-book/monitoring-distributed-systems/>. Consultado el 27 de mayo de 2026. ↵
2. Wilkinson, J. (2018). *Alerting on SLOs*. En B. Beyer, N. R. Murphy, D. Rensin, K. Kawahara y S. Thorne (eds.), *The Site Reliability Workbook*. <https://sre.google/workbook/alerting-on-slos/>. Consultado el 27 de mayo de 2026. ↵
3. Baye, C. A. (2016). *Emergency Response*. En *Site Reliability Engineering*. <https://sre.google/sre-book/emergency-response/>. Consultado el 28 de mayo de 2026. ↵
4. Stribblehill, A. (2016). *Managing Incidents*. En *Site Reliability Engineering*. <https://sre.google/sre-book/managing-incidents/>. Consultado el 28 de mayo de 2026. ↵
5. Lunney, J. y Lueder, S. (2016). *Postmortem Culture: Learning from Failure*. En *Site Reliability Engineering*. <https://sre.google/sre-book/postmortem-culture/>. Consultado el 28 de mayo de 2026. ↵
6. Little, J. D. C. (1961). A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383> ↵

7. Trivedi, K. S. (2016). *Probability and Statistics with Reliability, Queuing, and Computer Science Applications* (2.ª ed.). Wiley. Fiabilidad de un sistema k-de-n con componentes independientes mediante la distribución binomial. ↵
8. Erlang, A. K. (1917). Solution of some Problems in the Theory of Probabilities of Significance in Automatic Telephone Exchanges. *The Post Office Electrical Engineers' Journal*, 10, 189-197. La fórmula de pérdida de Erlang B da la probabilidad de bloqueo en un sistema sin espera. ↵
9. Trivedi, K. S. (2016). *Probability and Statistics with Reliability, Queuing, and Computer Science Applications* (2.ª ed.). Wiley. Con tiempo de reparación exponencial de media MTTR, la probabilidad de recuperarse antes de un plazo es $1 - e^{-t/\text{MTTR}}$. ↵
10. La esperanza de una variable uniforme en $[0, T]$ es $T/2$; aplicado a continuidad, con copias cada T y fallo en instante uniforme, la pérdida media de datos es $T/2$. Véase Ross, S. M. (2014). *Introduction to Probability Models* (11.ª ed.). Academic Press. ↵
1. OpenTelemetry. (2026). *Traces*. <https://opentelemetry.io/docs/concepts/signals/traces/>. Consultado el 27 de mayo de 2026. ↵
2. OpenTelemetry. (2026). *Logs*. <https://opentelemetry.io/docs/concepts/signals/logs/>. Consultado el 27 de mayo de 2026. ↵
3. OpenTelemetry. (2026). *Metrics*. <https://opentelemetry.io/docs/concepts/signals/metrics/>. Consultado el 27 de mayo de 2026. ↵
4. Tabassi, E. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. National Institute of Standards and Technology, NIST AI 100-1. <https://doi.org/10.6028/NIST.AI.100-1>. ↵
5. Amazon Web Services. (2026). *Timeouts, Retries, and Backoff with Jitter*. AWS Builders Library. <https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/>. Consultado el 27 de mayo de 2026. ↵