Facsímil 06

Construir y operar

Capítulo 01

De prototipo a sistema operable

686f6c61 · 2026

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]

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

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

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	*Service Level Indicator*	Indicador de nivel de servicio.	¿Qué estamos midiendo realmente?
SLO	*Service Level Objective*	Objetivo de nivel de servicio.	¿Qué valor consideramos aceptable?
Presupuesto de error	*Error budget*	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?”	`runs_aceptadas / runs_totales`
SLO	“Queremos que casi todas salgan aceptables.”	`SLI_calidad >= 0.97`
Presupuesto de error	“Aceptamos un margen pequeño antes de parar.”	`1 - 0.97 = 0.03`

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	`runs_aceptadas / runs_totales`	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	`coste_total / runs_aceptadas`	Si el sistema es sostenible, incluyendo reintentos.
Abstención correcta	`abstenciones_correctas / casos_sin_evidencia`	Si el sistema sabe no responder cuando falta base.
Error de contrato	`salidas_invalidas / runs_totales`	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]

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

SÍMBOLO	SIGNIFICADO	EJEMPLO
A	Disponibilidad: fracción de tiempo en que el sistema sirve.	0,999 (tres nueves).
MTTF	*Mean Time To Failure*: tiempo medio en funcionamiento entre fallos.	720 horas.
MTTR	*Mean Time To Repair*: 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]

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

SÍMBOLO	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.

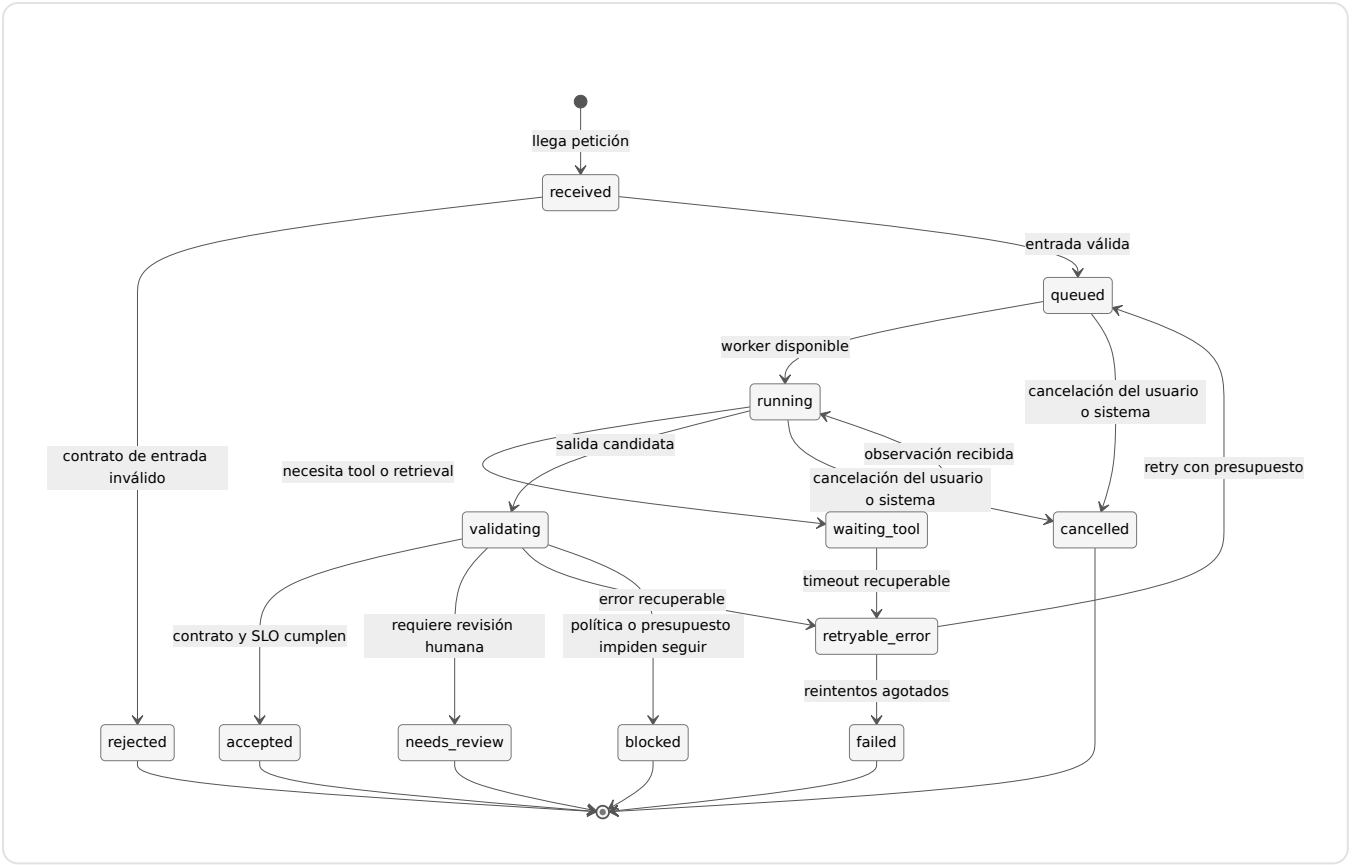
Para que una run sea operable, no basta con guardar el texto final. Debe dejar un contrato mínimo:

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

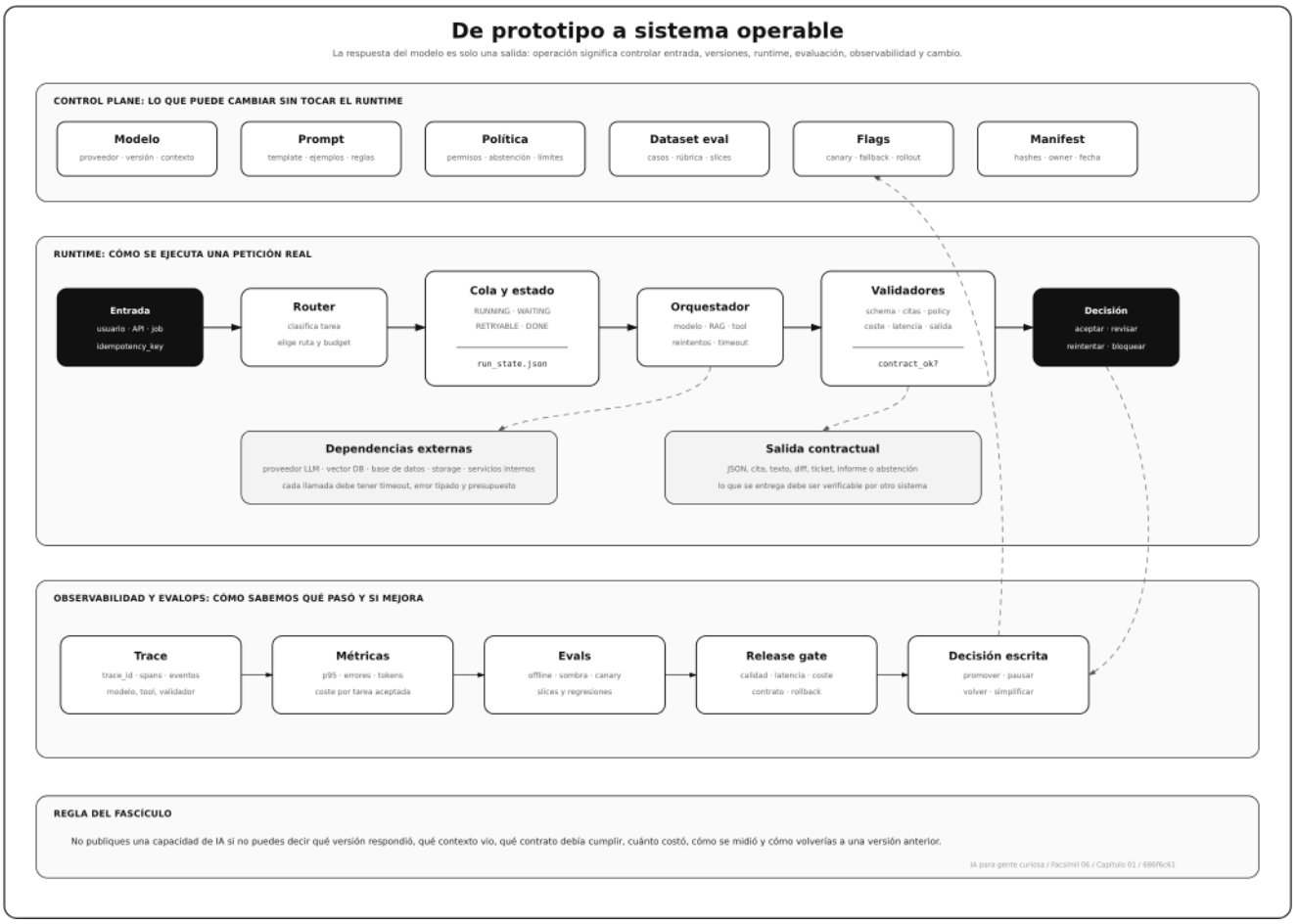
OpenAI documenta códigos de error como autenticación, permisos, límites, cuota y errores internos, y recomienda consultar la página de estado si aparece un error interno persistente.[15] Su página de estado separa componentes como APIs y muestra disponibilidad agregada, con la advertencia de que la disponibilidad 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:

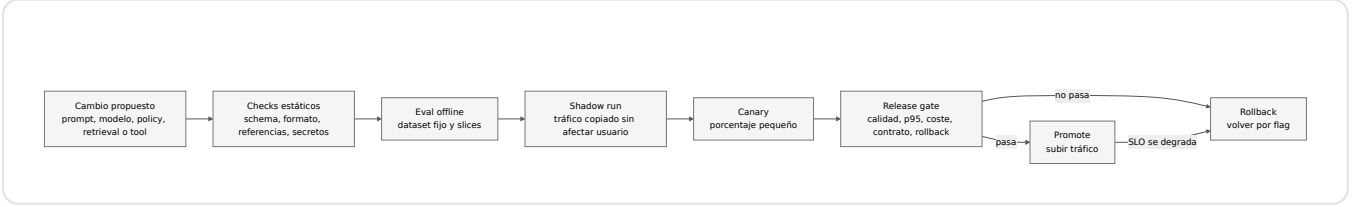
DECISIÓN	PREGUNTA CONCRETA	MALA SEÑAL	BUENA SEÑAL
Versión	¿Qué cambió hoy?	“Hemos tocado el prompt un poco”.	`prompt:v1.8.2`, `retriever:v3`, `policy:v4`.
Calidad	¿Qué casos cubre la eval?	Solo preguntas fáciles.	Casos frecuentes, ambiguos, sin evidencia y con documentos contradictorios.
Latencia	¿Qué 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
`manifest.yaml`	Versiones de modelo, prompt, política, dataset y runtime.	Diff revisable en PR.
`eval_dataset.jsonl`	Casos de entrada, expected, rúbrica y segmento.	Cobertura por tipo de caso.
`run_trace.jsonl`	Eventos de ejecución con `trace_id`, tiempos y atributos.	Permite reconstruir fallos.
`release_gate.py`	Código que decide si una versión pasa.	Tests unitarios y umbrales claros.
`rollback.md`	Cómo volver a la versión anterior.	Probado antes del lanzamiento.
`decision.md`	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

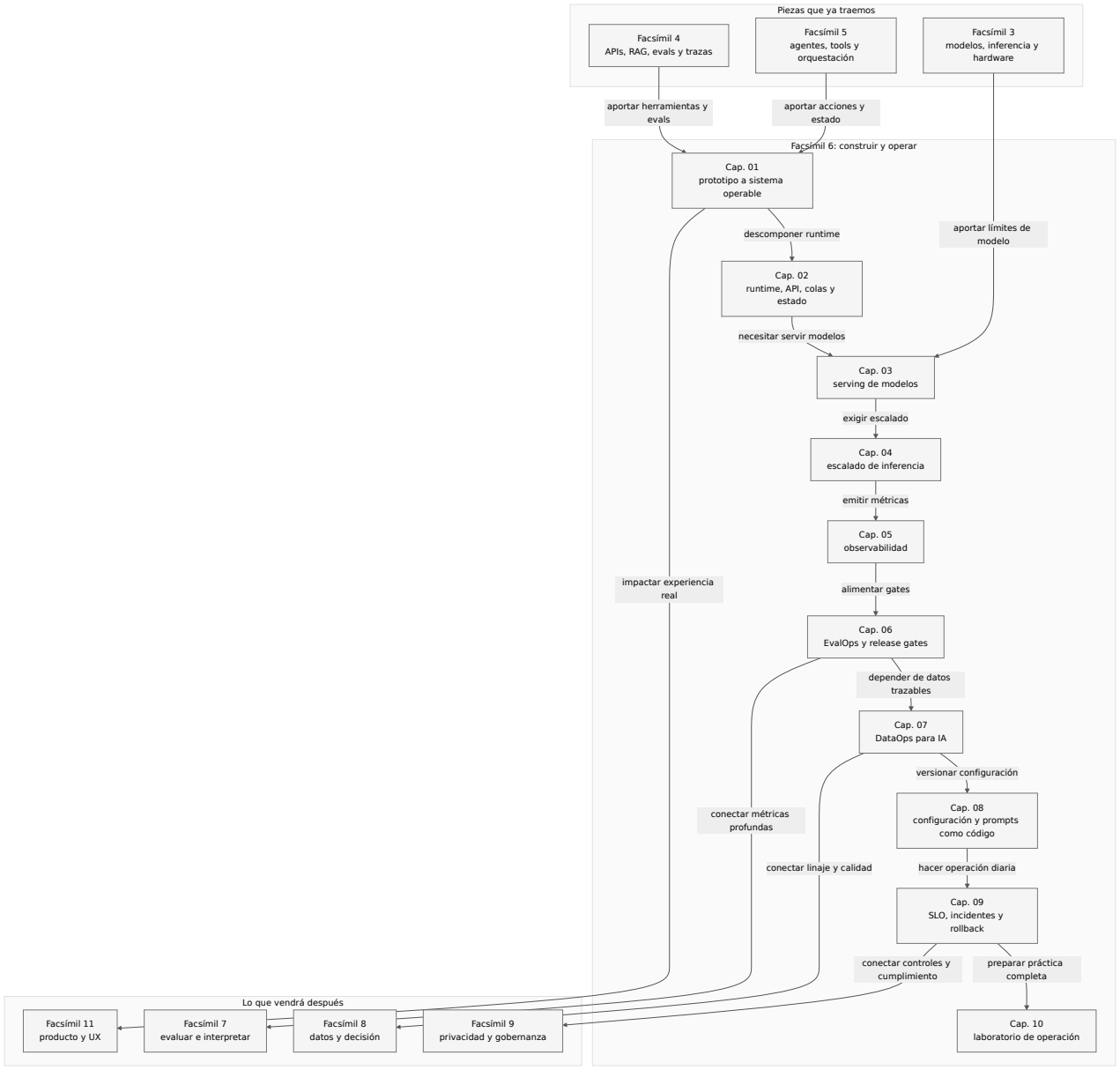
Porque la mayoría de los proyectos de IA no mueren por falta de un buen modelo, sino por no poder operarlos. Nadie sabe qué versión está en producción, por qué falló ayer, cuánto cuesta de verdad ni cómo volver atrás sin romper otra cosa. Las seis piezas de este capítulo (control plane, runtime, observabilidad, 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

TROPIEZO	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	Oculta 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
`AGENTS.md`	Archivo de instrucciones operativas del repositorio para agentes y personas: mapa, comandos, evidencias, límites y Definition of Done.
`SHOULD.md`	Especificación de comportamiento esperado: qué debe hacer el sistema, qué no debe hacer y cómo se evalúa.
`MUST / SHOULD / MAY`	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	*Service Level Indicator*: indicador medido, como calidad aceptada, p95, coste por tarea aceptada o error de contrato.
SLO	*Service Level Objective*: objetivo medible que fija qué nivel mínimo queremos sostener para un SLI.
Presupuesto de error	*Error budget*: 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/86df7dcfd896fcaf2674f757a2463eba-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/86df7dcfd896fcaf2674f757a2463eba-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. ↩

.0. World Wide Web Consortium. (2021). *Trace Context Level 2*. https://www.w3.org/TR/trace-context-2/. Consultado el 27 de mayo de 2026. ↩

.1. OpenAI. (2026). *API reference: debugging requests*. https://developers.openai.com/api/reference/overview#debugging-requests. Consultado el 27 de mayo de 2026. ↩

.2. Anthropic. (2026). *Errors*. https://platform.claude.com/docs/en/api/errors. Consultado el 27 de mayo de 2026. ↩

.3. Google. (2026). *Gemini API troubleshooting guide*. https://ai.google.dev/gemini-api/docs/troubleshooting. Consultado el 27 de mayo de 2026. ↩

.4. 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. ↩

.5. OpenAI. (2026). *Error codes*. https://developers.openai.com/api/docs/guides/error-codes. Consultado el 27 de mayo de 2026. ↩

.6. OpenAI. (2026). *OpenAI Status*. https://status.openai.com/. Consultado el 27 de mayo de 2026. ↩

.7. Anthropic. (2026). *Claude Status*. https://status.claude.com/. Consultado el 27 de mayo de 2026. ↩

18. 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. ↩

19. OpenAI. (2026). *AGENTS.md*. https://github.com/openai/agents.md. Consultado el 27 de mayo de 2026. ↩

20. Fowler, M. (2025). *Harness Engineering for Coding Agent Users*. https://martinfowler.com/articles/harness-engineering.html. Consultado el 27 de mayo de 2026. ↩

21. 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. ↩

22. 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. ↩