Facsímil 06

Construir y operar

Capítulo 10

Runbooks y continuidad de operación

686f6c61 · 2026

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:

SÍMBOLO	SIGNIFICADO	EJEMPLO
S	Señales de entrada.	Latencia p95, fallos de contrato, coste p95, cola, feedback.
Q	Consultas preparadas.	PromQL, SQL, enlace a trazas, panel de canary.
D	Decisión que debe tomarse.	Rollback, fallback, modo degradado, pausa de canary.
A	Acciones concretas.	Comando, cambio de flag, ruta alternativa, handoff.
V	Verificación.	SLO recuperado, contrato válido, coste normal, traza completa.
E	Evidencia que se preserva.	`trace_id`, `release_id`, `model_id`, `prompt_version`, `index_version`.
C	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:

SIGLA	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]

$$B(c,a) = \frac{\dfrac{a^{c}}{c!}}{\displaystyle\sum_{k=0}^{c} \frac{a^{k}}{k!}}$$

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	*Recovery Time Objective*: 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]

$$\mathbb{E}[L] = \frac{T_{\text{copia}}}{2}, \qquad L_{\max} = T_{\text{copia}}$$

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?	`model_id`, p95, coste, flake rate, errores.	Cambiar ruta, reducir esfuerzo, fallback.
RAG	¿El contexto recuperado sirve?	`index_version`, chunks, citas, recall manual.	Volver índice, bajar `top_k`, 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.[1][11][12][13] 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:

$$R = \frac{\sum_{i=1}^{n} w_i c_i}{\sum_{i=1}^{n} w_i}$$

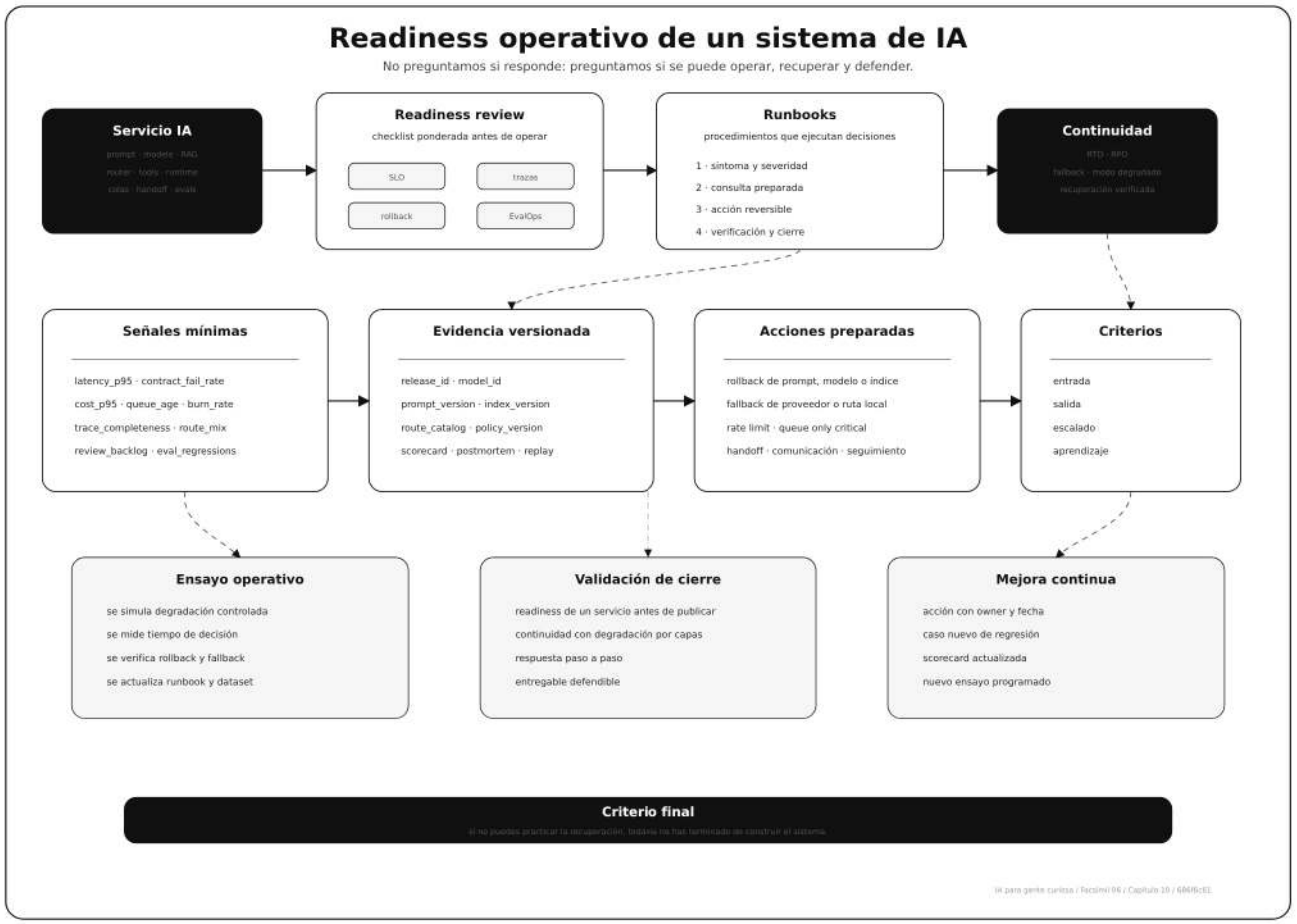
SÍMBOLO	SIGNIFICADO	EJEMPLO
R	Puntuación de readiness entre 0 y 1.	0,84.
w_i	Peso de la comprobación i.	2 para rollback, 1 para documentación auxiliar.
c_i	Resultado de la comprobación: 1 si pasa, 0 si falta.	`trace_id` presente: 1.
n	Número de comprobaciones.	26.

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

TROPIEZO	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 practicas: 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

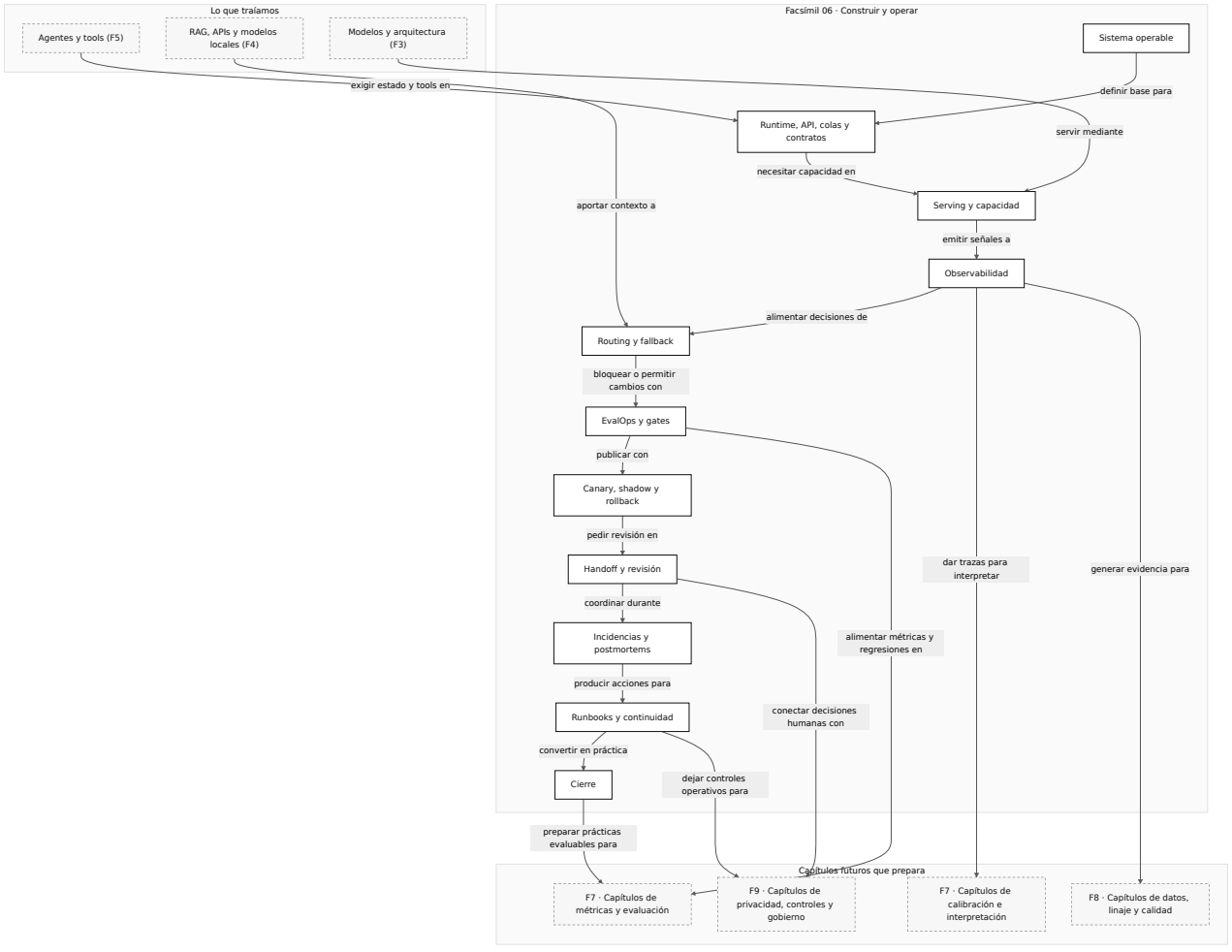
Qué practicas: enrutar cada tarea al modelo justo y poner topes de coste, sin perder calidad. **Dónde encaja:** capítulo 5 (routing, fallback y presupuestos por tarea). **Qué necesitas:** un navegador. Corre en CPU; sin claves (se simulan los modelos).

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: RTO, RPO y modo degradado`.
¿Sabes qué ruta usar si el proveedor principal degrada latencia?	Capítulo 05.
¿Tienes consultas preparadas para latencia, contrato, coste y cola?	Capítulo 04.
¿Puedes volver a una versión conocida sin tocar tres cosas a la vez?	Capítulo 07.
¿Una incidencia crea casos nuevos de regresión?	Capítulo 06.
¿Tu cola de revisión tiene SLO y modo degradado?	Capítulo 08.
¿Tu postmortem deja acciones con owner y verificación?	Capítulo 09.
¿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 = λW. *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 = λW. *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 $1 - e^{-t/\mathrm{MTTR}}$. ↩

.0. 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. ↩