Facsímil 08

La ciencia de los datos

Capítulo 06

DataOps: pipelines, drift y monitorización

686f6c61 · 2026

Capítulo 06: DataOps: pipelines, drift y monitorización

Entrando en el tema

En el capítulo anterior vimos que una decisión no se publica solo porque tenga una métrica global aceptable. Hay que mirarla por slices, escribir gates y decidir `pass`, `review` o `block`. Ahora viene la pregunta operativa: **¿qué pasa después de publicar?**

Los datos cambian. Las colas cambian. Los idiomas cambian. Un producto nuevo concentra tráfico. Un campo deja de llegar. Una traza se pierde. Una latencia sube. La etiqueta real llega días después. Y el sistema que el lunes parecía razonable puede dejar de representar la producción del jueves.

Al terminar deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Diseñar un pipeline de datos para IA.	Distingues fuente, validación, transformación, evaluación, serving y monitorización.
Leer una ejecución como artefacto.	Sabes qué inputs, outputs, hashes, versiones y estado debe dejar una run.
Definir SLIs y SLOs de datos.	Escribes indicadores medibles, objetivos y acciones cuando fallan.
Medir drift sin convertirlo en superstición.	Calculas distancia entre referencia y producción y decides qué revisar.
Conectar slices con operación.	No miras solo drift global: mides slices críticos y consecuencias.
Crear un runbook y un postmortem.	Dejas una guía de respuesta y una explicación técnica de la incidencia.

La promesa de este capítulo es operativa: cuando algo cambie, no deberías empezar desde cero. Deberías poder abrir una run, ver qué datos entraron, qué contrato los validó, qué versión del pipeline los transformó, qué métricas se degradaron, qué slice concentra el daño y qué runbook toca ejecutar. Eso no elimina incidentes; elimina improvisación.

La frase central:

> DataOps no es tener dashboards. Es saber qué dato cambió, qué run lo produjo, qué decisión afecta y qué acción toca.

La escena: ayer pasaba, hoy bloquea

Imagina que el asistente académico ya está en producción. El viernes se aprobó la política de decisión: casos claros se priorizan, casos claramente normales pasan a flujo normal y casos inciertos van a revisión. El lunes todo parece estable.

El martes entra una campaña de prácticas internacionales. De repente suben los casos en inglés, casi todos con necesidad de accesibilidad, y además una parte del pipeline pierde `trace_id`. La métrica global del mes todavía no parece dramática, pero la ventana del día cuenta otra historia: más latencia, más revisión, más casos prioritarios que acaban como `normal`.

Ese es el problema que resuelve DataOps. No pregunta solo “¿funcionó el modelo?”. Pregunta:

PREGUNTA OPERATIVA	POR QUÉ IMPORTA
¿Qué ventana cambió?	Una media mensual puede esconder un día roto.
¿Qué versión de pipeline produjo los eventos?	Un cambio de código puede explicar el problema.
¿Qué datos entraron?	Sin hash ni versión, no se reproduce la decisión.
¿Qué slices fallan?	El daño operativo suele concentrarse.
¿Hay trazas completas?	Sin `trace_id`, investigar se vuelve conjetura.
¿Qué runbook se ejecuta?	Una alerta sin acción se convierte en ruido.

Estas preguntas enseñan una idea sencilla: un incidente de datos rara vez tiene una sola cara. Puede ser cambio de distribución, fallo de instrumentación, retraso de etiquetas, rotura de contrato, aumento de latencia o mezcla de todo lo anterior. Si el sistema solo tiene una alerta global, cada incidente se convierte en discusión. Si tiene trazas, versiones y SLIs, la conversación empieza mucho más cerca de la causa.

Qué no es DataOps

DataOps no es “tener un CSV limpio”. Tampoco es instalar un orquestador y dar por resuelta la operación. Un orquestador lanza jobs; no decide por sí mismo si un dataset representa producción, si un slice crítico se degradó o si una ventana debe bloquear un release.

Tampoco es guardar logs infinitos sin contrato. Si cada evento tiene mil campos pero no sabemos cuáles son obligatorios, qué SLI alimentan o quién responde cuando fallan, hemos cambiado desorden pequeño por desorden caro.

Y no es mirar drift como si fuera una alarma universal. Drift significa cambio de distribución. A veces es una incidencia; a veces es una campaña esperada; a veces es una mejora de cobertura; a veces indica que la evaluación ya no representa el uso real. La métrica abre una revisión. No sustituye el criterio.

Qué sí es DataOps para IA

DataOps es tratar los datos, runs y artefactos como piezas operables del sistema. Esto incluye contratos, validaciones, linaje, versionado, monitorización, gates y runbooks. En IA, además, debe conectar con modelos, features, embeddings, prompts, evals, slices y decisiones.

Sculley y otros explicaron que los sistemas de ML acumulan deuda técnica por dependencias de datos, cambios no locales y realimentaciones difíciles de ver.[1] TFX formalizó una arquitectura de producción con ingestión, validación, transformación, entrenamiento, evaluación y serving reproducibles.[2] El ML Test Score propuso que la madurez de un sistema ML se mida también por tests de datos, monitorización y gestión de cambios.[3]

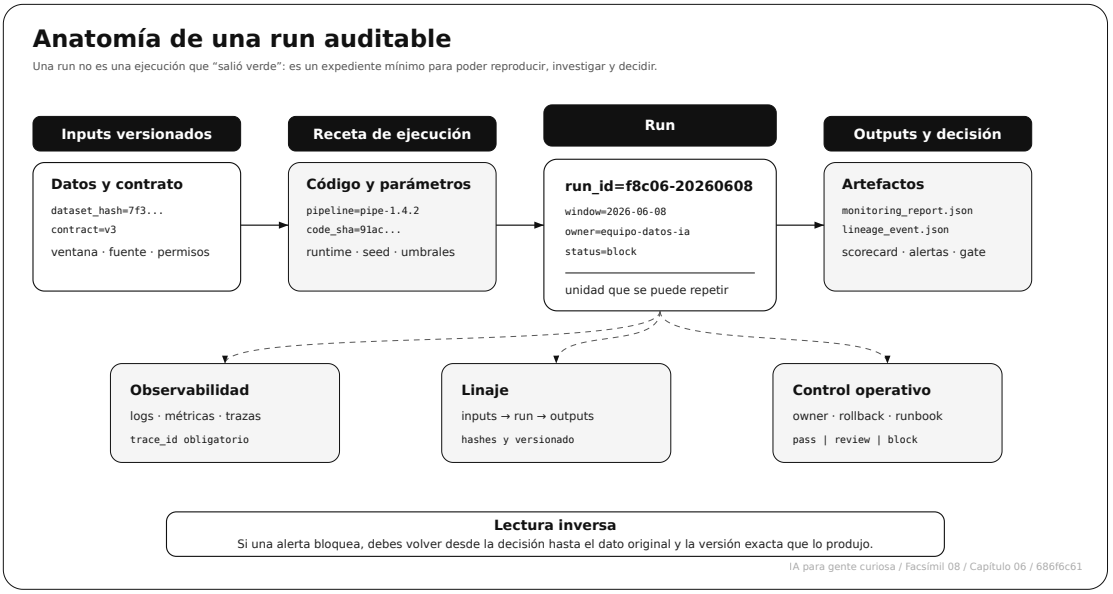
Fecha de corte: **7 de junio de 2026**. Fuentes consultadas ese día: OpenLineage, OpenTelemetry, Great Expectations, Evidently, Prometheus, Apache Airflow, Dagster, Prefect, dbt, TFX, ML Test Score y literatura de ingeniería de ML. Lo estable no es la herramienta concreta: es la obligación de versionar, medir, trazar y decidir.

El pipeline por dentro

Un pipeline de IA no es solo “entrenar y desplegar”. Es una cadena de contratos. Cada organización tendrá pasos distintos; lo importante no es adornarlo con símbolos, sino que cada paso transforme una entrada, produzca una salida y deje evidencia auditable.

PASO	ENTRADA	SALIDA	EVIDENCIA MÍNIMA
Ingesta	Fuente externa o interna.	Snapshot controlado.	`source_id`, timestamp, hash.
Validación	Snapshot.	Reporte de calidad.	Checks, columnas, valores inválidos.
Transformación	Datos válidos.	Features, chunks o embeddings.	Versión de código, parámetros, hashes.
Evaluación	Predicciones o outputs.	Métricas y slices.	Dataset, política, umbrales, intervalos.
Gate	Métricas.	`pass`, `review` o `block`.	Regla exacta y motivo.
Serving	Modelo o política.	Decisiones.	`trace_id`, versión, latencia.
Monitorización	Eventos de producción.	Alertas y scorecard.	SLI, SLO, runbook, owner.

La tabla se lee de izquierda a derecha, pero también al revés. Si una alerta dice que `2026-06-08` queda en `block`, deberías poder volver desde la decisión hasta la run, desde la run hasta los inputs, desde los inputs hasta los hashes y desde los hashes hasta el contrato que dice por qué ese dato era aceptable o no. Esa reversibilidad es la diferencia entre “tengo un dashboard” y “puedo defender técnicamente lo que ha pasado”.



> Una run auditable conecta entradas, contrato, versión de código, parámetros, outputs, trazas, linaje y decisión.

OpenLineage propone un estándar abierto para capturar metadatos de runs, jobs y datasets en componentes de pipeline.[4] OpenTelemetry define señales de observabilidad como trazas, métricas y logs; una traza permite seguir una operación por spans, una métrica mide valores agregados y un log registra eventos.[5][6][7]

En términos prácticos:

SEÑAL	QUÉ CONTESTA	EJEMPLO EN IA
Log	Qué ocurrió.	"Evento p013 llegó sin `trace_id`".
Métrica	Cuánto ocurre.	`missing_trace_rate = 0.1`.
Traza	Por dónde pasó.	Ingesta → validación → decisión → revisión.
Linaje	De dónde viene y qué produjo.	`production_events.csv` → `monitoring_report.json`.
Runbook	Qué hacer ahora.	Revisar pipeline `pipe-1.4.2`, idioma `en`, slice `access_need=si`.

Patrones de arquitectura de pipeline

Un ingeniero de software no debería quedarse en “tengo un script”. La pregunta es qué garantías necesita ese script cuando se ejecuta todos los días, con datos que cambian y con personas dependiendo de sus salidas.

PATRÓN	QUÉ RESUELVE	RIESGO SI NO LO DISEÑAS
Batch	Procesar ventanas cerradas.	Repetir una ventana y obtener otra decisión sin saber por qué.
Streaming	Procesar eventos conforme llegan.	Mezclar eventos tardíos, duplicados o incompletos.
Checkpoint	Recordar hasta dónde llegó una run.	Reprocesar a ciegas o perder eventos.
Backfill	Recalcular ventanas antiguas.	Cambiar histórico sin versionar pipeline, datos y contrato.
Replay	Reproducir eventos con una versión concreta.	No poder depurar una decisión pasada.
Retry	Reintentar un paso fallido.	Duplicar efectos si no hay idempotencia.
Canary	Probar una versión en una parte controlada.	Publicar un cambio a toda producción sin señal temprana.
Shadow mode	Ejecutar una versión sin afectar la decisión.	Confundir evaluación silenciosa con decisión real.

Apache Airflow organiza workflows como DAGs con tareas y dependencias.[8] Dagster enfatiza assets definidos por software, linaje, observabilidad y testabilidad.[9] Prefect documenta deployments con metadatos de ejecución remota, versiones y configuración, y también retries configurables por workflow o tarea.[10][11]

La herramienta importa menos que el contrato de ejecución:

DECISIÓN DE DISEÑO	PREGUNTA QUE DEBES RESPONDER
Orquestador	¿Quién lanza la run, con qué parámetros y dónde queda su estado?
Idempotencia	¿Qué clave evita duplicar una decisión si se reintenta?
Retries	¿Qué pasos se pueden repetir y cuáles requieren revisión manual?
Backfill	¿Qué versión de datos, código y contrato se usa para recalcular?
Checkpoint	¿Qué pasa si una run cae a mitad?
Output	¿La salida se escribe una vez, se sobreescribe o se versiona?
Rollback	¿Cómo volvemos a una política anterior sin perder trazabilidad?

Idempotencia merece una explicación propia. Una operación es idempotente si repetirla produce el mismo efecto observable que ejecutarla una vez. En pipelines de IA, la clave suele ser algo como:

```
idempotency_key = window + event_id
```

Si una tarea falla después de emitir una decisión y se reintenta sin esa clave, puedes duplicar un caso, mandar dos revisiones humanas o contar dos veces una alerta. Esto no es un detalle de plataforma: es ingeniería de software aplicada al dato.

Herramientas por capa, sin casarse con ninguna

En DataOps conviene hablar de herramientas porque el alumno las va a encontrar en empresas, prácticas, ofertas de trabajo y proyectos reales. Pero la herramienta no debe aparecer como tótem. Airflow, Dagster o Prefect no arreglan un contrato mal definido; Great Expectations no sabe qué coste tiene un falso negativo; OpenTelemetry no inventa el `trace_id` que tu aplicación no emitió; Prometheus no decide si una ventana debe bloquear un release. La herramienta hace visible y repetible una disciplina que ya deberías haber diseñado.

CAPA	HERRAMIENTAS HABITUALES	QUÉ APORTAN	PREGUNTA DE INGENIERÍA ANTES DE USARLAS
Orquestación	Airflow, Dagster, Prefect	Ejecutar dependencias, reintentos, scheduling y estado de runs.	¿Qué pasa si una tarea falla después de escribir salida parcial?
Contratos y calidad	dbt contracts, Great Expectations	Declarar columnas, tipos, checks y expectativas.	¿Qué cambio de schema rompe consumidores y cuál es compatible?
Linaje	OpenLineage e integraciones de orquestadores	Relacionar jobs, runs, datasets, inputs y outputs.	¿Puedo reconstruir qué produjo una decisión concreta?
Observabilidad	OpenTelemetry, Prometheus, Grafana	Emitir trazas, métricas, logs y alertas.	¿Qué SLI dispara acción y quién responde?
Drift y monitorización ML	Evidently, dashboards de ML observability	Comparar referencia y producción, slices y cambios de distribución.	¿El drift significa incidencia, campaña esperada o cambio de población?
Evidencia de release	CI, tests, artefactos versionados	Bloquear o aprobar ventanas con pruebas reproducibles.	¿El gate deja una decisión auditable o solo un número bonito?

Para un proyecto pequeño, un `Makefile`, scripts Python y JSON bien escritos pueden enseñar mejor que una plataforma enorme. Para producción, la pregunta cambia: cómo se integra con CI/CD, dónde viven los secretos, quién firma el cambio de contrato, cuánto cuesta retener trazas, qué datos personales viajan en logs y cómo se versiona cada artefacto. El cuaderno del facsímil usa un montaje mínimo para que se entienda el mecanismo antes de subirlo a una plataforma real.

Contratos entre servicios y evolución de schema

En capítulos anteriores hablamos de contrato de datos. Aquí ampliamos la idea: un pipeline de producción tiene contratos entre productores y consumidores. El productor promete columnas, tipos, semántica, frecuencia y trazabilidad. El consumidor promete cómo usará esos campos y qué cambios acepta.

dbt model contracts permiten declarar y hacer cumplir columnas, tipos y restricciones en modelos dbt.[1][2] Great Expectations organiza expectativas verificables sobre datos. OpenLineage conecta jobs y datasets. OpenTelemetry conecta ejecución y trazas. Son capas distintas de la misma idea: **un cambio no debería romper al consumidor en silencio**.

CAMBIO	COMPATIBLE	REQUIERE REVISIÓN	ROMPE
Añadir columna opcional	Sí	Si afecta coste o privacidad.	No.
Añadir valor nuevo a catálogo	A veces.	Sí, si cambia slices o política.	Si el consumidor no lo acepta.
Renombrar columna obligatoria	No.	Sí.	Sí, salvo alias versionado.
Cambiar significado de `decision`	No.	Sí.	Sí.
Quitar `trace_id`	No.	Sí.	Sí.
Subir un SLO	Sí.	Sí, si bloquea más ventanas.	No necesariamente.

Un contrato de evolución debería declarar:

PIEZA	EJEMPLO
Compatibilidad hacia atrás	La versión nueva acepta datos de la versión anterior.
Compatibilidad hacia delante	La versión anterior puede ignorar campos nuevos opcionales.
Deprecación	Campo permitido hasta una fecha o versión.
Aliases	`student_profile` puede aceptar temporalmente `profile`.
Campo obligatorio	`trace_id` no se puede quitar.
Política de rollback	Volver a `pipe-1.4.1` conserva outputs y hashes.

SLIs de datos que no son drift

Drift es importante, pero no es el único síntoma que rompe un sistema de IA. Wang y Strong ya separaban la calidad de datos en dimensiones que importan a los consumidores del dato, no solo al productor: exactitud, completitud, actualidad, relevancia, interpretabilidad y accesibilidad, entre otras.[13] En ingeniería de IA eso se traduce en una idea sencilla: una ventana puede no tener drift fuerte y aun así no ser usable porque llegó tarde, incompleta, sin etiqueta, sin trazas o con un contrato parcialmente roto.

Un SLI de datos debe tener tres piezas: una unidad medible, una ventana temporal y una consecuencia. Si dices “calidad de datos”, todavía no has dicho nada operativo. Si dices “porcentaje de eventos de decisión recibidos antes de 15 minutos desde su creación, por ventana y por canal”, ya puedes medir, alertar y decidir.

SLI DE DATOS	QUÉ MIDE	EJEMPLO DE CÁLCULO OPERATIVO	CUÁNDO BLOQUEA
Freshness	Edad del dato cuando llega al pipeline.	`event_available_at - event_created_at` .	Si el dato llega después de que la decisión ya se haya tomado.
Completitud	Campos obligatorios presentes.	`filas_con_campos_obligatorios / filas_totales` .	Si falta `trace_id` , `event_id` , `decision` , `window` o campo de slice crítico.
Validez	Valores dentro del contrato.	`valores_validos / valores_revisados` .	Si aparece un catálogo no permitido y el consumidor no sabe interpretarlo.
Unicidad	Duplicados por clave de idempotencia.	`duplicados(window,event_id)` .	Si un retry puede duplicar decisiones, alertas o revisiones humanas.
Puntualidad	Si la ventana cierra a tiempo.	`window_closed_at <= deadline` .	Si el batch llega tarde y contamina informes o entrenamiento.
Label delay	Retraso de la etiqueta real.	`label_available_at - decision_at` .	Si se evalúa miss rate antes de que haya verdad de terreno suficiente.
Trazabilidad	Eventos con traza investigable.	`eventos_con_trace_id / eventos_totales` .	Si una ventana no se puede depurar cuando falla.
Cobertura de slices	Slices críticos presentes con muestra mínima.	`n(slice) >= mínimo` .	Si una media global oculta que el segmento importante no llegó.

El valor de esta tabla está en obligarnos a separar síntomas. Freshness contesta si el dato llega a tiempo. Completitud contesta si llega entero. Validez contesta si respeta el contrato. Label delay contesta si estamos evaluando antes de que exista la verdad de terreno. Trazabilidad contesta si podremos investigar. Son dimensiones distintas; mezclarlas bajo la palabra “calidad” hace que el equipo reaccione tarde y mal.

Great Expectations y Deequ existen precisamente para expresar checks de calidad como reglas reproducibles en vez de como revisión manual difusa.[14][15] La parte que no puede hacer la herramienta por ti es decidir qué SLI tiene consecuencia. En un producto educativo, `label_delay` puede impedir evaluar una política durante varios días. En un RAG, `freshness` puede significar que una normativa nueva no entró al índice. En un sistema con agentes, `completitud` puede ser que falta el resultado de una herramienta y el agente está razonando con media verdad.

Para aterrizarlo: si una universidad recibe solicitudes de beca, una ventana con todos los campos completos pero con etiquetas reales retrasadas no sirve para estimar `miss_rate` de forma honesta. Puedes medir latencia, drift y trazabilidad, pero no puedes cerrar evaluación de acierto hasta que llegue la resolución real. En cambio, si el problema es freshness, quizá sí tienes etiquetas, pero llegaron demasiado tarde para que la alerta protegiera la decisión. Son fallos distintos y piden defensas distintas.

CASO REALISTA	SLI QUE MANDA	ACCIÓN RAZONABLE
Documentos de políticas actualizados por la noche.	Freshness del índice documental.	No usar el índice para respuestas normativas hasta terminar ingestión y validación.
Tickets con `access_need` vacío por cambio de formulario.	Completitud por slice crítico.	Bloquear automatización y corregir el productor del evento.
Etiquetas de resolución llegan 10 días después.	Label delay.	Separar monitorización temprana de evaluación final con etiqueta.
Reintento de pipeline repite `event_id`.	Unicidad por clave idempotente.	Detener escritura de salida y exigir replay controlado.
Aparece `language=pt` sin contrato.	Validez de catálogo.	Enviar a `review` hasta versionar contrato, slices y evaluación.

Observabilidad correlacionada

La observabilidad de IA no consiste en guardar un log grande. Consiste en poder saltar de una alerta a los eventos concretos y de ahí a la traza que explica dónde se degradó.

Una traza mínima de decisión debería poder responder:

CAMPO	POR QUÉ IMPORTA
`trace_id`	Une logs, métricas, spans y evento final.
`event_id`	Identifica la unidad de decisión.
`pipeline_version`	Distingue cambio de código de cambio de datos.
`model_version`	Permite comparar comportamiento por versión.
`data_version`	Evita investigar con otro snapshot.
`policy_version`	Explica por qué el mismo score acabó en otra decisión.
`span_name`	Localiza ingesta, validación, scoring, decisión o emisión.
`duration_ms`	Separa problema de calidad de problema de latencia.

En el cuaderno del facsímil, el contrato de ingeniería exige spans `ingest`, `validate`, `score`, `decide` y `emit`. El evento `p013` no tiene `trace_id`. El evento `p017` tiene traza, pero le falta `emit`. Los eventos `p011`, `p012` y `p017` muestran `score` lento. Esa correlación permite decir algo concreto: la ventana `2026-06-08` no solo cambió de distribución; también llegó peor instrumentada y más lenta.

EVENTO	VENTANA	SEÑAL
`p011`	`2026-06-08`	Traza total `710 ms`, `score` lento.
`p012`	`2026-06-08`	Traza total `755 ms`, `score` lento.
`p013`	`2026-06-08`	Sin `trace_id`.
`p017`	`2026-06-08`	Falta span `emit`, `score` lento.

Sin esta correlación, el equipo podría culpar al modelo, al dato o al usuario sin evidencia suficiente. Con trazas, la pregunta cambia: ¿qué parte del pipeline cambió entre `pipe-1.4.1` y `pipe-1.4.2`?

Esta es una diferencia profunda con una cultura de logs sin diseño. Un log suelto dice que algo ocurrió; una traza correlacionada permite reconstruir una decisión. Si `event_id`, `trace_id`, versión de pipeline, versión de datos y spans no viajan juntos, cada dashboard cuenta una parte de la historia. DataOps consiste en que esas partes se puedan juntar cuando más falta hace.

Telemetría sin datos sensibles

Hay una tensión real: para investigar necesitas contexto, pero si conviertes logs y trazas en un vertedero de datos personales, has creado otro problema. El Reglamento General de Protección de Datos exige principios como minimización, limitación de finalidad y protección desde el diseño; NIST Privacy Framework propone gestionar privacidad como riesgo de ingeniería, no como nota legal al final.[16][17]

En DataOps para IA esto significa que `trace_id` debe existir, pero no debe ser el DNI del usuario, el email del alumno ni el texto completo de una consulta sensible. Una traza útil necesita correlación, versiones, spans, duración y estado. No necesita copiar el contenido completo del documento privado que pasó por el pipeline.

ELEMENTO DE TELEMETRÍA	ÚTIL	RIESGO	FORMA MÁS SEGURA
`trace_id`	Correlacionar eventos.	Reidentificar si contiene datos personales.	ID aleatorio o hash no reversible con sal.
`user_email`	Depurar un caso concreto.	PII directa en logs.	Pseudónimo interno o referencia a sistema con permisos.
Texto completo del prompt	Diagnosticar comportamiento.	Puede contener datos personales, secretos o documentos privados.	Guardar hash, plantilla, longitud, categoría y muestra redaccionada si hay permiso.
Labels de Prometheus	Filtrar métricas por dimensión.	Alta cardinalidad y fuga de identificadores.	Labels de baja cardinalidad: modelo, versión, canal, región, slice permitido.
Span attributes	Entender paso lento o fallido.	Meter payloads enteros en trazas.	Estado, duración, versión, error class y tamaños, no contenido sensible.
Retención	Investigar incidentes.	Acumular más datos de los necesarios.	TTL por tipo de señal y política de borrado.

La observabilidad útil no debería convertirse en una copia paralela de producción llena de datos sensibles. Este equilibrio es difícil, pero muy práctico: guardar identificadores técnicos, hashes, tamaños, versiones y clases de error suele bastar para detectar patrones; revisar contenido completo debería ser un flujo excepcional, con permisos, redacción y auditoría. Si no diseñamos esto desde el principio, la depuración se vuelve cómoda a costa de privacidad y riesgo.

Prometheus recomienda nombres y etiquetas consistentes; en la práctica, además, conviene evitar labels de cardinalidad explosiva como `user_id`, `document_id`, `email` o `raw_prompt`, porque rompen coste, rendimiento y privacidad.[18] OpenTelemetry permite enriquecer trazas

con atributos, pero el criterio de ingeniería es el mismo: atributo sí, volcado indiscriminado no. Una traza sana te permite saber que `score` tardó 420 ms en `pipe-1.4.2` para `language=en`; no debería obligarte a guardar el texto personal de la solicitud.

Ejemplo concreto: para depurar un RAG académico no necesitas registrar la pregunta completa “mi expediente con DNI X tiene…” ni el PDF privado usado como contexto. Puedes registrar `retrieval_k=8`, `top_source_type=normativa`, `doc_version=2026-06-01`, `query_hash`, `prompt_template_version`, `input_token_count`, `output_token_count`, `policy_version` y `trace_id`. Si hace falta revisar contenido, se abre un flujo con permisos, redacción y auditoría. Eso es más lento que guardar todo, sí. También es la diferencia entre observabilidad profesional y una fuga esperando fecha.

Testing de pipelines de IA

Un pipeline de IA necesita tests, pero no todos los tests son iguales. Un test unitario puede comprobar que una función calcula una media. Eso es necesario, pero insuficiente. En producción también hay que probar contratos, ventanas, trazabilidad, linaje, regresiones por slices y comportamiento operativo.

TIPO DE TEST	QUÉ COMPRUEBA	EJEMPLO ÚTIL
Unitario	Una función aislada.	`total_variation_distance(P, Q)` devuelve `0.8` en un caso conocido.
Contract test	Productor y consumidor hablan el mismo idioma.	`trace_id`, `event_id`, `decision`, `latency_ms` y `window` existen y tienen tipo esperado.
Data quality test	La ventana tiene datos válidos.	No hay nulos en campos obligatorios, catálogos permitidos y rangos razonables.
Lineage test	La run conserva procedencia.	Los hashes de inputs y outputs quedan escritos en `lineage_event.json`.
Regression test	Una versión nueva no empeora una referencia.	`pipe-1.4.2` no reduce captura segura frente a `pipe-1.4.1`.
Slice test	No se esconde un fallo en la media.	`language=en` y `access_need=si` mantienen SLO propio.
Trace test	La operación se puede investigar.	Cada evento crítico tiene spans `ingest`, `validate`, `score`, `decide` y `emit`.
Replay test	Se puede reproducir una ventana.	Reejecutar `2026-06-08` con la misma versión produce la misma decisión.

El error típico es probar solo el código y no probar el sistema. En IA, el sistema incluye datos, contratos, scheduler, runtime, versión de modelo, política de decisión, evals, slices y observabilidad. Por eso el ML Test Score no se limita a accuracy: pregunta por tests de datos, monitorización, cambios y dependencia de features.[19]

Para un equipo de ingeniería, una regla práctica sería esta:

```
No hay release de pipeline si no pasan:
1. contrato de datos,
2. contrato de trazas,
3. replay de una ventana conocida,
4. scorecard de slices críticos,
5. evento de linaje con hashes.
```

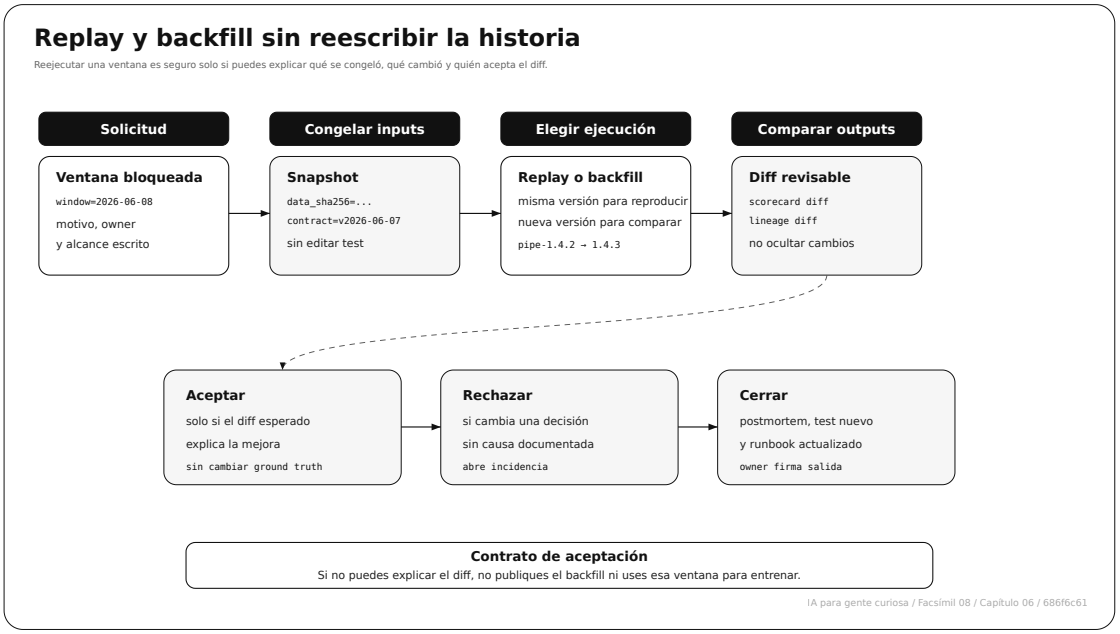
Esto parece mucho hasta que ocurre la primera incidencia real. Entonces descubres que esos cinco puntos no son burocracia: son la diferencia entre investigar en una hora o discutir durante días.

Backfill y replay sin romper histórico

Backfill y replay suelen parecer palabras de plataforma, pero para IA son decisiones delicadas. Un **replay** reejecuta una ventana para reproducir o comparar una decisión. Un **backfill** recalcula ventanas antiguas para completar histórico o aplicar una versión nueva. Los dos pueden ser sanos; los dos pueden destruir trazabilidad si se hacen sin contrato.

La regla profesional es esta: no se recalcula histórico “porque sí”. Se abre un expediente de backfill. Ese expediente dice qué ventana se toca, qué snapshot entra, qué código se usa, qué contrato gobierna, qué política de decisión se aplica, qué outputs se esperan, qué diff se acepta y qué queda prohibido. Google SRE trata los cambios operativos e incidentes como sistemas que necesitan preparación, respuesta y aprendizaje; en IA aplicada, backfill y replay son parte de esa disciplina.[20][21]

PASO	QUÉ SE CONGELA O REVISA	EJEMPLO
1. Motivo	Por qué se reejecuta.	`trace_id` faltante en ventana `2026-06-08`.
2. Snapshot	Datos exactos de entrada.	`production_events.csv` con hash SHA-256.
3. Código	Versión de pipeline.	`pipe-1.4.2` o parche `pipe-1.4.3`.
4. Contrato	Reglas vigentes.	`monitoring_contract.json` versión fechada.
5. Política	Umbrales y gates.	No cambiar SLOs para que el backfill “pase”.
6. Salida esperada	Artefactos a comparar.	`monitoring_report.json`, `slo_scorecard.csv`, `lineage_event.json`.
7. Diff permitido	Cambios aceptables.	Se corrige traza; no cambia etiqueta ni decisión sin justificación.
8. Cierre	Quién firma y qué test queda.	Postmortem + test de spans obligatorios.



> Replay y backfill necesitan un contrato propio: snapshot, versión, política, diff aceptable, postmortem y test nuevo.

Ejemplo: si el evento `p017` no emitió el span `emit`, no debes “rellenar” el histórico a mano. Primero congelas la ventana, reejecutas con la misma versión para comprobar reproducibilidad y luego pruebas el parche que fuerza spans obligatorios. Si el único cambio

esperado es que `p017` ahora tiene traza completa y el resto del scorecard no cambia, el backfill es defendible. Si cambian decisiones, etiquetas o slices sin explicación, no es un backfill: es una alteración del experimento.

Drift, SLI, SLO y presupuesto de error

El drift se mide comparando una distribución de referencia P con una distribución actual Q. Una medida clásica es la distancia de variación total, habitual en probabilidad y teoría de la información.[22]

$$TV(P, Q) = \frac{1}{2} \sum_{c \in C} |P(c) - Q(c)|$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
P	Distribución de referencia.	Idiomas durante validación.
Q	Distribución actual.	Idiomas del 8 de junio de 2026.
C	Categorías posibles.	`es` , `ca` , `en` .
$P(c)$	Proporción de la categoría c en referencia.	`en = 0.2` .
$Q(c)$	Proporción de la categoría c en producción.	`en = 1.0` .
$TV(P, Q)$	Distancia entre 0 y 1.	`0.8` , cambio fuerte.

En palabras: $TV(P, Q)$ suma cuánto se mueve cada categoría entre referencia y producción y divide por dos para no contar el desplazamiento dos veces. Si en referencia el idioma `en` era el 20 % y en producción pasa al 100 %, mientras `es` y `ca` desaparecen de esa ventana, el cambio es enorme. La fórmula no dice por sí sola “incidencia”: dice que la ventana ya no se parece a la referencia y que debes abrir una investigación.

También se usa PSI (*Population Stability Index*) en scoring, riesgo y monitorización de distribuciones. Es una señal industrial útil, no una prueba universal.[23]

$$PSI(P, Q) = \sum_{c \in C} (Q(c) - P(c)) \log \frac{Q(c)}{P(c)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
PSI	Índice de estabilidad poblacional.	`11.711575` para `language`.
$\log$	Logaritmo natural.	Penaliza cambios grandes de proporción.
$Q(c) - P(c)$	Diferencia de proporciones.	`en` sube mucho.

En palabras: PSI pesa la diferencia de proporciones por el logaritmo del cociente entre producción y referencia. Por eso castiga mucho los casos donde una categoría casi desaparece o aparece de golpe. En ingeniería se usa como señal de estabilidad, no como verdad absoluta. Si `language` tiene PSI altísimo, no se reentrena automáticamente: primero se pregunta si hubo campaña, cambio de canal, bug de ingestión, cambio de producto o sesgo de muestra.

Pero drift no es lo único. Para operar necesitamos SLIs y SLOs.

CONCEPTO	DEFINICIÓN	EJEMPLO
SLI	Indicador medible.	`latency_p95_ms`, `miss_rate`, `missing_trace_rate`.
SLO	Objetivo sobre un SLI.	`latency_p95_ms <= 650`.
Presupuesto de error	Margen tolerado antes de frenar cambios.	Si `miss_rate > 0.12`, no se aumenta automatización.
Gate	Regla que convierte SLOs en estado.	`block` si falta trazabilidad o cae captura segura.

Prometheus insiste en que los nombres de métricas deben ser claros, consistentes y expresar unidades cuando aplica.[24] Eso parece menor hasta que tienes que investigar una alerta a las 9 de la mañana. `latency` es ambiguo; `decision_latency_p95_ms` dice mucho más.

Presupuesto consumido y burn rate

En ingeniería de fiabilidad, el presupuesto de error traduce un SLO a margen consumible: si prometes que una proporción mínima de eventos cumple, la parte que puede fallar antes de frenar cambios es $1 - s$, donde s es el objetivo del SLO. Google SRE popularizó esta forma de conectar señales operativas con riesgo para usuarios.[25] El *SRE Workbook* lo lleva a alerting basado en consumo de presupuesto, no solo en umbrales aislados.[26]

En un capítulo de datos para IA, esta idea no se limita a disponibilidad web. Puede aplicarse a una ventana de decisiones: eventos sin traza, casos críticos no capturados, spans que superan latencia, registros fuera de contrato o slices que incumplen el mínimo pactado. Lo importante es no llamarlo “intuición”. Debe quedar escrito qué cuenta como evento malo, cuál era el SLO y qué ocurre si se consume demasiado presupuesto.

Si una ventana contiene N eventos evaluables y el SLO exige una proporción mínima s de eventos correctos, el presupuesto de error permitido para esa ventana es:

$$B = (1 - s)N$$

SÍMBOLO	SIGNIFICADO	EJEMPLO DEL CUADERNO
B	Número esperado de errores permitidos antes de gastar el presupuesto de la ventana.	`1.2` fallos permitidos.
s	Objetivo del SLO expresado como proporción.	`0.88` de captura segura mínima.
N	Eventos evaluables en la ventana.	`10` eventos.

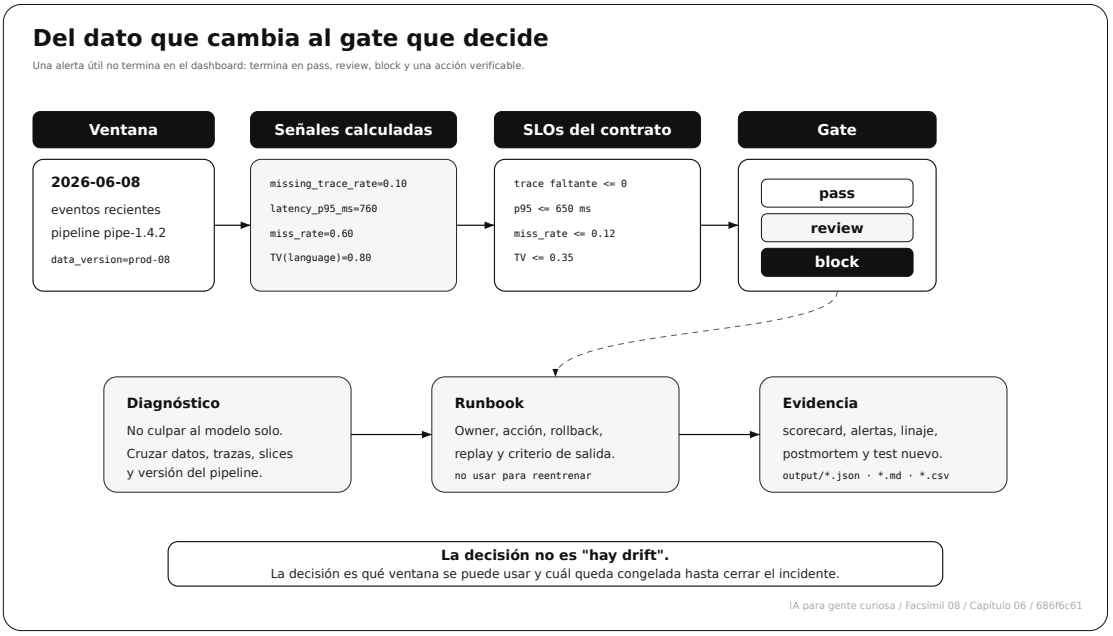
En palabras: si aceptas como SLO que al menos el 88 % de los casos críticos queden capturados, estás aceptando como margen máximo un 12 % de fallos. En una ventana de 10 eventos, eso equivale a 1.2 fallos permitidos. No significa que puedas tener “un fallo con decimales”; significa que, agregando ventanas, ese es el margen estadístico que estás consumiendo. Si observas 6 fallos, la ventana no está cerca del límite: lo ha atravesado con claridad.

Para comparar ventanas de distinto tamaño se usa la idea de *burn rate*: cuántas veces más rápido estás consumiendo el presupuesto de error respecto a lo permitido por el SLO. En una ventana w, con e_w eventos malos entre n_w eventos evaluables:

$$BR_w = \frac{e_w / n_w}{1 - s}$$

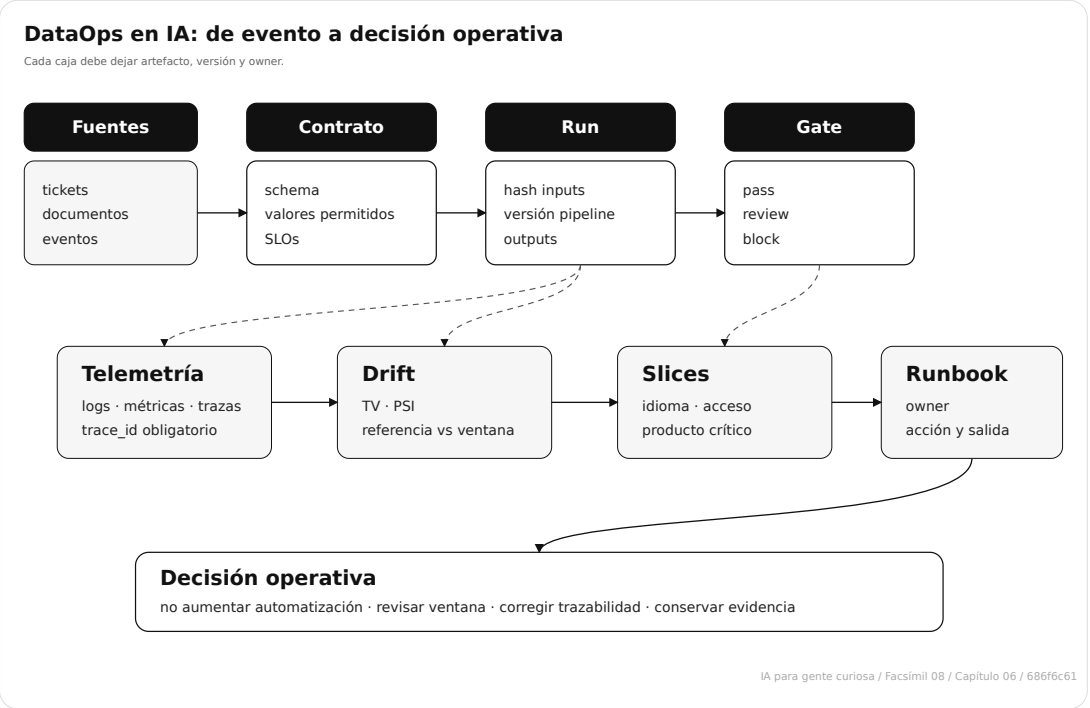
SÍMBOLO	SIGNIFICADO	EJEMPLO DEL CUADERNO
BR_w	Burn rate de la ventana w.	`5.0` , consumo cinco veces superior al permitido.
e_w	Eventos malos observados en la ventana.	`6` fallos.
n_w	Eventos evaluables en la ventana.	`10` eventos.
$1 - s$	Fracción de error permitida por el SLO.	`0.12` .

En palabras: si el SLO permite fallar un 12 % y la ventana falla un 60 %, el burn rate es $0.60/0.12 = 5$. Esa cifra ya no es una opinión sobre si “parece mal”: es una señal operativa para bloquear, abrir incidente y exigir replay antes de reusar la ventana. En sistemas de IA esto importa muchísimo porque una ventana rota puede acabar convertida en dataset de reentrenamiento, benchmark interno o decisión de producto.



El gate convierte señales sueltas en una decisión operativa: qué se bloquea, quién actúa y qué evidencia queda.

Arquitectura operativa de un sistema de datos para IA



Un pipeline DataOps convierte eventos en evidencia: contrato, run, telemetría, drift, slices, gate y runbook.

En el día a día

En producción trabajamos con ventanas. Una ventana puede ser un día, una hora, un batch, una versión de modelo, una campaña o un segmento de tráfico. La ventana es importante porque las medias largas esconden problemas cortos.

En el cuaderno del facsímil, la ventana `2026-06-07` pasa. La ventana `2026-06-08` bloquea.

VENTANA	ESTADO	N	TRACE FALTANTE	LATENCIA P95	REVISIÓN	PÉRDIDA	CAPTURA SEGURA	FLAGS
2026-06-07	`pass`	10	0.0	590.0	0.4	0.0	1.0	0
2026-06-08	`block`	10	0.1	760.0	0.7	0.6	0.4	15

La segunda ventana no bloquea por una sola razón. Bloquea porque varias señales coinciden:

SEÑAL	LECTURA
`missing_trace_rate = 0.1`	Falta trazabilidad; una investigación quedaría incompleta.
`latency_p95_ms = 760`	La ventana supera el SLO de latencia.
`miss_rate = 0.6`	Demasiados casos prioritarios pasan a flujo normal.
`safety_capture = 0.4`	La política ya no captura de forma segura los casos importantes.
Drift en `language` y `access_need`	Producción se aleja mucho de la referencia.
Slices críticos fallan	`language=en` , `access_need=si` y `product=practicas` concentran señales.

Aquí se ve la diferencia entre dashboard y operación. Un dashboard muestra números. Un gate operativo dice: **no aumentes automatización ni uses esta ventana para reentrenar sin investigar**.

Severidad: no todas las alertas pesan igual

Un capítulo de DataOps necesita una matriz de severidad porque no todo `review` ni todo `block` significan lo mismo. Una alerta de latencia aislada puede pedir seguimiento. Una ventana sin trazas en eventos críticos puede impedir investigar una decisión. Un slice crítico con `miss_rate` alto puede afectar directamente a usuarios. Google SRE separa respuesta de emergencia, gestión de incidentes y cultura de postmortems para evitar que todo se trate igual y para que cada incidente deje aprendizaje verificable.[272829]

Para IA, la severidad no se decide solo por cuántas filas fallan. Se decide por consecuencia: si afecta una decisión automática, si contamina entrenamiento, si rompe trazabilidad, si toca un slice crítico, si expone datos sensibles o si impide medir el resultado real.

SEVERIDAD	SEÑAL TÍPICA	CONSECUENCIA	ACCIÓN MÍNIMA
S0	Exposición de datos sensibles en logs, decisión automática dañina o pérdida masiva de trazabilidad.	Riesgo para personas, cumplimiento o continuidad.	Congelar ventana, parar automatización afectada, abrir incidente y activar responsable.
S1	Slice crítico con `block`, `trace_id` faltante en eventos críticos o miss rate muy por encima del SLO.	No se puede defender la ventana.	No reentrenar, no aumentar automatización, replay y postmortem.
S2	Drift fuerte, latencia alta o revisión humana por encima de capacidad.	La ventana puede operar con restricciones, pero no promocionarse.	`review`, owner, seguimiento y acción correctiva fechada.
S3	Alerta leve sin impacto directo y con trazabilidad completa.	Señal de observación.	Registrar, mirar tendencia y no abrir incidente mayor.

Ejemplo: `latency_p95_ms = 760` puede ser S2 si la decisión sigue siendo correcta y trazable. `missing_trace_rate = 0.1` puede subir a S1 si afecta eventos críticos, porque sin traza no puedes reconstruir qué pasó. Una fuga de prompt con datos personales en logs no es “solo observabilidad”: puede ser S0 aunque el modelo responda bien.

La matriz de severidad evita dos errores opuestos. El primero es dramatizar todo y saturar al equipo. El segundo es normalizar señales graves porque el dashboard tiene muchas luces. Un sistema maduro no reacciona por ansiedad: reacciona porque una regla escrita conecta señal, consecuencia y acción.

Incidentes, postmortems y acciones correctivas

Cuando una ventana queda en `block`, el trabajo no termina en “hay alerta”. Empieza la parte profesional: reconstruir impacto, señales, causa probable, acción correctiva y criterio de cierre. El postmortem no busca repartir culpa. Busca impedir que el mismo patrón vuelva a repetirse sin que el sistema lo detecte.

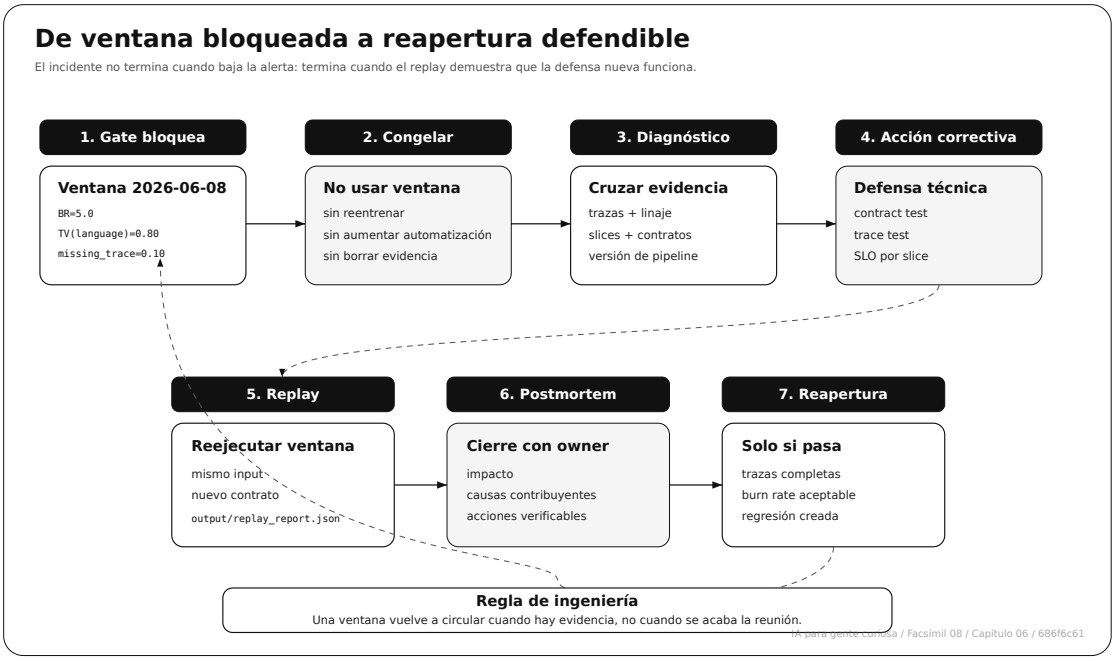
Un postmortem técnico mínimo debería incluir:

BLOQUE	PREGUNTA QUE RESPONDE	EJEMPLO DEL CUADERNO
Resumen	¿Qué estado queda y qué se bloquea?	`Estado de ingeniería: block`.
Impacto	¿Qué decisión queda afectada?	No aumentar automatización en `2026-06-08`.
Timeline	¿Qué scripts y señales detectaron el problema?	`monitor_dataops.py` y `inspect_pipeline_engineering.py`.
Señales	¿Qué eventos concretos fallaron?	`p013` sin `trace_id`, `p017` sin `emit`.
Causa probable	¿Qué explicación encaja con la evidencia?	Cambio de distribución más trazabilidad incompleta y `score` lento.
Acción correctiva	¿Qué se cambia en código, contrato o operación?	Hacer `trace_id` bloqueante y añadir test de spans.
Criterio de cierre	¿Cómo sabemos que está resuelto?	Replay con trazas completas y scorecard sin `block`.

El detalle importante es que cada acción correctiva debe convertirse en una defensa técnica. Si el problema fue una columna obligatoria ausente, añade un contract test. Si fue una traza incompleta, añade un trace test. Si fue un retry que duplicó efectos, añade clave de idempotencia. Si fue una ventana de datos rota, añade un gate que impida usarla para reentrenamiento.

PROBLEMA DETECTADO	ACCIÓN POBRE	ACCIÓN DE INGENIERÍA
Falta `trace_id`	“Mirarlo manualmente”.	Bloquear emisión si falta `trace_id` en eventos críticos.
Span `score` lento	“Optimizar algo”.	Medir p95 por versión y crear SLO por span.
Slice crítico degradado	“Avisar al equipo”.	Añadir SLO por slice y replay de la ventana.
Drift fuerte	“Reentrenar rápido”.	Separar drift esperado, drift operativo y datos no aptos.
Backfill de histórico	“Recalcular y ya”.	Congelar versión de código, contrato, datos y política.

Un postmortem útil también deja una decisión negativa explícita: **qué no se va a hacer**. En el cuaderno del facsímil, no se usa `2026-06-08` para aumentar automatización ni para reentrenar hasta que el gate cierre. Esa frase protege al equipo de convertir datos dudosos en verdad operativa.



El ciclo profesional de una ventana bloqueada: congelar, diagnosticar, corregir, reproducir y reabrir solo con evidencia.

Por qué debería importarte

Si no entiendes DataOps, puedes tomar malas decisiones con muy buena intención. Puedes reentrenar con una ventana rota. Puedes comparar modelos con datos que ya no representan producción. Puedes ignorar un slice crítico porque la media global parece estable. Puedes perder trazas justo cuando más las necesitas.

La operación de IA necesita una idea incómoda: el modelo no vive solo. Vive dentro de contratos, datos, versiones, colas, revisiones humanas, latencia, permisos, trazas y decisiones de producto.

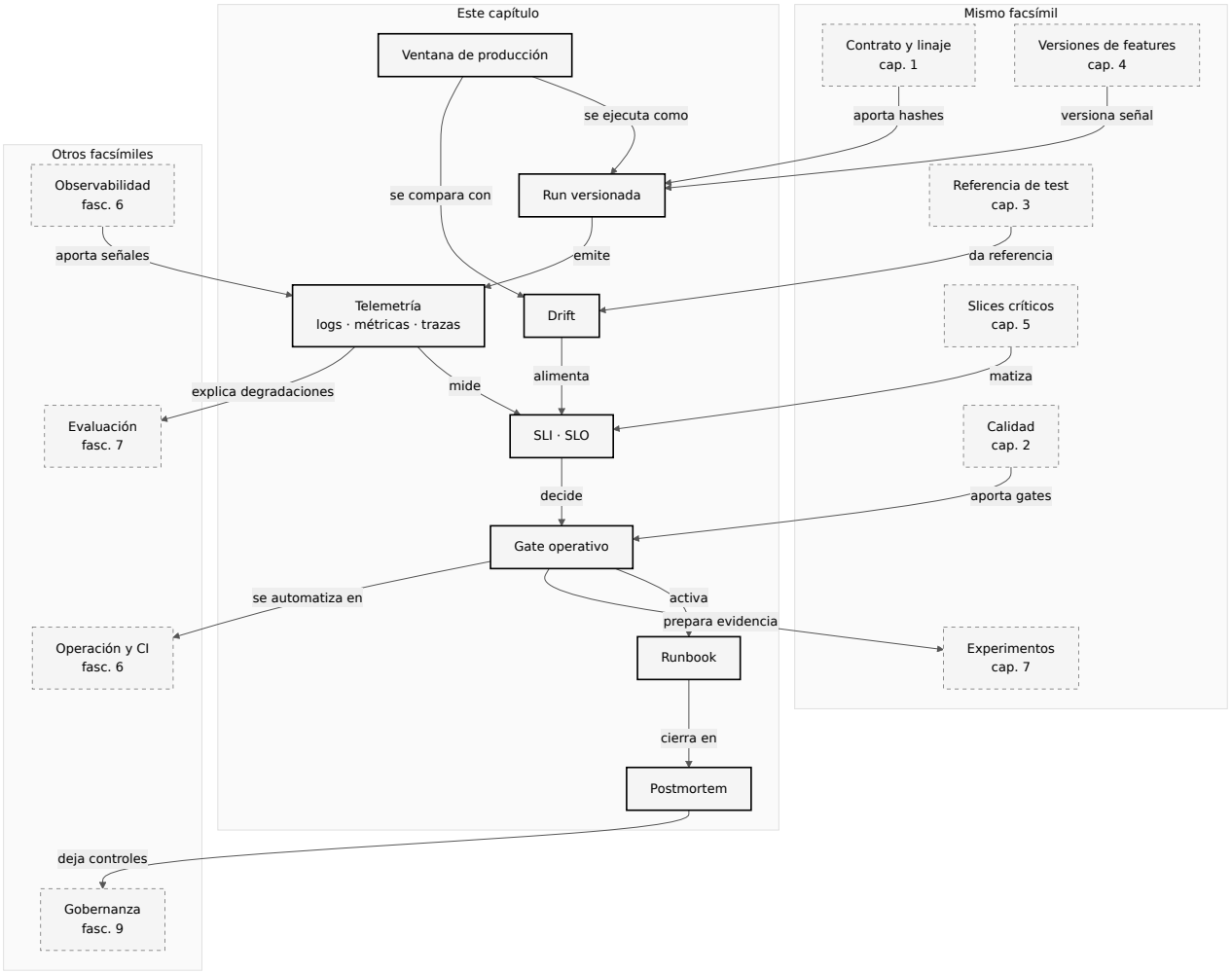
Dónde solía tropezar yo

TROPIEZO	POR QUÉ OCURRE	ANTÍDOTO
Mirar solo métricas globales	La media da sensación de control.	Medir ventanas y slices críticos.
Confundir drift con fallo automático	Un cambio puede ser esperado.	Revisar causa, contexto y consecuencia.
No exigir `trace_id`	Parece detalle de logging.	Tratar trazabilidad como SLO bloqueante.
Tener alerta sin runbook	El dashboard avisa, pero nadie sabe qué hacer.	Escribir owner, acción y criterio de salida.
Reentrenar con producción rota	Se intenta arreglar rápido.	Bloquear ventanas con gate `block`.
Reintentar sin idempotencia	Se piensa que retry siempre es inocente.	Definir `idempotency_key` antes de automatizar retries.
Hacer backfill sin congelar versión	Parece solo recalcular.	Guardar datos, código, contrato, política y hashes.
Escribir postmortem sin test nuevo	Se documenta el problema, pero no se evita repetirlo.	Cada postmortem debe cerrar con una defensa verificable.

Cómo encaja todo

Este mapa se lee como continuidad del facsímil. Los capítulos 01 a 05 construyeron contrato, calidad, split, representación y slices. Este capítulo convierte esas piezas en operación: ventanas, SLIs, SLOs, drift, trazas, linaje y runbooks.

La decisión que enseña no es “qué modelo elegir”. Es más básica y más profesional: **qué ventana de producción es apta para seguir tomando decisiones y cuál exige revisión**. Después, el capítulo 07 usará esta evidencia para análisis aplicado, experimentos y causalidad.


Este capítulo
Ventana de producción
se ejecuta como
se compara con
Run versionada
emite
Telemetría
logs · métricas · trazas
Drift
alimenta
mide
explica degradaciones
SLI · SLO
decide
Gate operativo
se automatiza en
activa
prepara evidencia
Runbook
cierra en
Postmortem
Mismo facsímil
Contrato y linaje
cap. 1
Versiones de features
cap. 4
aporta hashes
versiona señal
Referencia de test
cap. 3
da referencia
Slices críticos
cap. 5
matiza
Calidad
cap. 2
aporta gates
Experimentos
cap. 7
Otros facsímiles
Observabilidad
fasc. 6
aporta señales
Evaluación
fasc. 7
Operación y CI
fasc. 6
deja controles
Gobernanza
fasc. 9

Vocabulario aprendido

TÉRMINO	DEFINICIÓN BREVE
DataOps	Operar datos como producto versionado, verificable y monitorizable.
Pipeline	Secuencia reproducible de pasos que transforma entradas en artefactos.
Run	Ejecución concreta con inputs, outputs, versiones, hashes y estado.
Drift	Cambio medible entre referencia y producción.
SLI	Indicador medible.
SLO	Objetivo sobre un SLI.
Presupuesto de error	Margen permitido antes de frenar cambios.
Burn rate	Velocidad relativa a la que una ventana consume el presupuesto de error.
Trace ID	Identificador que permite seguir una operación por el sistema.
Lineage event	Evento que registra job, run, entradas, salidas, hashes y relación entre datasets.
Freshness	Edad operativa del dato respecto al momento en que se necesita para decidir.
Completitud	Proporción de campos, eventos o relaciones obligatorias que llegan con valor usable.
Puntualidad	Capacidad de que el dato llegue antes del cierre operativo de su ventana.
Label delay	Retraso entre una predicción y la etiqueta real que permite evaluarla.
Cardinalidad	Número de valores distintos de una etiqueta, campo o dimensión en métricas, logs o trazas.
Minimización de datos	Recoger y conservar solo los datos necesarios para una finalidad concreta.
Severidad	Clasificación que conecta señal, consecuencia, urgencia y acción mínima.
Runbook	Guía de acción cuando una alerta o gate falla.
Gate operativo	Regla que convierte señales en `pass`, `review` o `block`.
Idempotencia	Propiedad que permite repetir una operación sin duplicar efectos.

TÉRMINO	DEFINICIÓN BREVE
Backfill	Reprocesado de ventanas antiguas con versión controlada de datos, código y contrato.
Replay	Reejecución de una ventana para reproducir o comparar una decisión.
Span	Tramo de una traza que representa un paso medible de una operación.
Contract test	Test que verifica que productor y consumidor cumplen el contrato pactado.
Postmortem	Documento técnico que convierte una incidencia en acciones verificables.

Antes de pasar página

Antes de avanzar, deberías poder responder:

1. ¿Qué diferencia hay entre pipeline y run?
2. ¿Por qué una ventana corta puede contar más que una media larga?
3. ¿Qué mide `missing_trace_rate` y por qué puede bloquear?
4. ¿Qué diferencia hay entre log, métrica y traza?
5. ¿Qué mide la distancia de variación total?
6. ¿Qué mide PSI y por qué no se lee sin contexto?
7. ¿Qué es un SLI?
8. ¿Qué es un SLO?
9. ¿Qué significa presupuesto de error?
10. ¿Cómo se calcula el presupuesto consumido de una ventana?
11. ¿Qué significa un burn rate de 5.0 en una ventana de producción?
12. ¿Por qué una ventana como `2026-06-08` queda en `block`?
13. ¿Qué hace `lineage_event.json`?
14. ¿Qué debe contener un runbook útil?
15. ¿Qué slices críticos monitorizarías en tu proyecto?
16. ¿Por qué no deberías reentrenar con una ventana rota?
17. ¿Por qué un retry sin idempotencia puede duplicar efectos?
18. ¿Qué diferencia hay entre backfill y replay?
19. ¿Qué comprobaría un contract test de trazas?
20. ¿Qué acción correctiva añadirías a un postmortem de datos?

21. ¿Qué evidencia pedirías antes de reabrir una ventana bloqueada?
22. ¿Cómo conecta este capítulo con análisis aplicado?
23. ¿Qué herramienta usarías para orquestar, cuál para contratos, cuál para trazas y cuál para drift?
24. ¿Qué evidencia pedirías antes de usar una ventana nueva para reentrenar?
25. ¿Qué SLI usarías para medir freshness en un RAG documental?
26. ¿Por qué `label_delay` puede hacer que una métrica parezca buena demasiado pronto?
27. ¿Qué campos jamás guardarías como labels de Prometheus?
28. ¿Cómo investigarías una incidencia sin guardar prompts crudos?
29. ¿Qué diferencia operativa hay entre replay y backfill?
30. ¿Qué debe congelarse antes de un backfill seguro?
31. ¿Cuándo subirías una alerta de S2 a S1?
32. ¿Qué postmortem te dejaría un test nuevo y no solo una explicación bonita?

En resumen

IDEA	QUÉ TE LLEVAS
DataOps es operación, no decoración.	Contratos, runs, hashes, gates y runbooks sostienen el sistema.
El drift abre investigación.	Un cambio de distribución no se interpreta sin contexto ni consecuencia.
Los SLOs deben ser medibles.	`latency_p95_ms <= 650` es operativo; “que vaya bien” no lo es.
El presupuesto de error se consume.	Si el burn rate se dispara, la ventana se congela hasta tener replay y defensa nueva.
La trazabilidad puede bloquear.	Sin `trace_id`, no hay investigación fiable.
Calidad de datos no es solo drift.	Freshness, completitud, puntualidad y retraso de etiqueta pueden romper una decisión aunque la distribución parezca estable.
La telemetría también tiene privacidad.	Una traza útil debe explicar la operación sin convertirse en almacén de prompts crudos o datos personales.
Los slices siguen vivos en producción.	Lo que se auditó en test debe monitorizarse después.
Un gate protege decisiones.	Evita aumentar automatización o reentrenar con ventanas rotas.
Los retries necesitan idempotencia.	Repetir una tarea no debe duplicar decisiones ni alertas.
Backfill y replay necesitan contrato.	Recalcular histórico sin snapshot, versión y diff aceptado puede destruir evidencia.
La severidad traduce señal en acción.	No todas las alertas pesan igual: impacto, trazabilidad, privacidad y automatización cambian la respuesta.
Un postmortem debe crear una defensa.	La acción correctiva se traduce en test, contrato o gate.

Para saber más

Apache Airflow. (2026). Core Concepts. Documentación

AWS Labs. (2026). Deequ: Unit Tests for Data. GitHub

Baye, C. A. (2016). Emergency Response. En *Site Reliability Engineering*. Google SRE

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. *KDD*, 1387-1395. DOI

Breck, E., Cai, S., Nielsen, E., Salib, M. y Sculley, D. (2017). The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. *IEEE Big Data*, 1123-1132. Google Research

Cover, T. M. y Thomas, J. A. (2006). *Elements of Information Theory* (2.ª ed.). Wiley.

Dagster. (2026). Dagster documentation. Documentación

dbt Labs. (2026). Model contracts. Documentación

Evidently AI. (2026). Data Drift Documentation. Documentación

European Parliament and Council of the European Union. (2016). Regulation (EU) 2016/679. Eur-Lex

Ewaschuk, R. (2016). *Monitoring Distributed Systems*. En *Site Reliability Engineering*. Google SRE

Great Expectations. (2026). Expectations Overview. Documentación

Jones, C., Wilkes, J., Murphy, N. R. y Beyer, B. (2016). Service Level Objectives. En *Site Reliability Engineering*. Google SRE

Lunney, J. y Lueder, S. (2016). *Postmortem Culture: Learning from Failure*. En *Site Reliability Engineering*. Google SRE

National Institute of Standards and Technology. (2020). *NIST Privacy Framework: A Tool for Improving Privacy Through Enterprise Risk Management, Version 1.0*. DOI

OpenLineage. (2026). OpenLineage Documentation. Documentación

OpenTelemetry. (2026). Logs. Documentación

OpenTelemetry. (2026). Metrics. Documentación

OpenTelemetry. (2026). Traces. Documentación

Prefect. (2026). Deployments. Documentación

Prefect. (2026). How to automatically rerun your workflow when it fails. Documentación

Prometheus. (2026). Metric and label naming. Documentación

Sculley, D. y otros (2015). Hidden Technical Debt in Machine Learning Systems. *NeurIPS*. Paper

Siddiqi, N. (2006). *Credit Risk Scorecards: Developing and Implementing Intelligent Credit Scoring*. Wiley.

Stribblehill, A. (2016). Managing Incidents. En *Site Reliability Engineering*. Google SRE

Wang, R. Y. y Strong, D. M. (1996). Beyond Accuracy: What Data Quality Means to Data Consumers. *Journal of Management Information Systems*, 12(4), 5-33. DOI

Wilkinson, J. (2018). *Alerting on SLOs*. En *The Site Reliability Workbook*. Google SRE

Notas

1. Sculley, D. y otros (2015). Hidden Technical Debt in Machine Learning Systems. *NeurIPS*. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html ↩

2. Baylor, D. y otros (2017). TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. *KDD*, 1387-1395. https://doi.org/10.1145/3097983.3098021 ↩

3. Breck, E., Cai, S., Nielsen, E., Salib, M. y Sculley, D. (2017). The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. *IEEE Big Data*, 1123-1132. https://research.google/pubs/pub46555/ ↩

4. OpenLineage. (2026). *OpenLineage Documentation*. https://openlineage.io/docs/. Consultado el 7 de junio de 2026. ↩

5. OpenTelemetry. (2026). *Traces*. https://opentelemetry.io/docs/concepts/signals/traces/. Consultado el 7 de junio de 2026. ↩

6. OpenTelemetry. (2026). *Metrics*. https://opentelemetry.io/docs/concepts/signals/metrics/. Consultado el 7 de junio de 2026. ↩

7. OpenTelemetry. (2026). *Logs*. https://opentelemetry.io/docs/concepts/signals/logs/. Consultado el 7 de junio de 2026. ↩

8. Apache Airflow. (2026). *Core Concepts*. https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/. Consultado el 7 de junio de 2026. ↩

9. Dagster. (2026). *Dagster documentation*. https://docs.dagster.io/getting-started. Consultado el 7 de junio de 2026. ↩

.0. Prefect. (2026). *Deployments*. https://docs.prefect.io/concepts/deployments. Consultado el 7 de junio de 2026. ↩

.1. Prefect. (2026). *How to automatically rerun your workflow when it fails*. https://docs.prefect.io/v3/how-to-guides/workflows/retries. Consultado el 7 de junio de 2026. ↩

.2. dbt Labs. (2026). *Model contracts*. https://docs.getdbt.com/docs/mesh/govern/model-contracts. Consultado el 7 de junio de 2026. ↩

.3. Wang, R. Y. y Strong, D. M. (1996). Beyond Accuracy: What Data Quality Means to Data Consumers. *Journal of Management Information Systems*, 12(4), 5-33. https://doi.org/10.1080/07421222.1996.11518099 ↩

.4. Great Expectations. (2026). *Expectations Overview*. https://docs.greatexpectations.io/docs/cloud/expectations/expectations_overview/. Consultado el 7 de junio de 2026. ↩

.5. AWS Labs. (2026). *Deequ: Unit Tests for Data*. https://github.com/awslabs/deequ. Consultado el 7 de junio de 2026. ↩

.6. European Parliament and Council of the European Union. (2016). *Regulation (EU) 2016/679*. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679. Consultado el 7 de junio de 2026. ↩

.7. National Institute of Standards and Technology. (2020). *NIST Privacy Framework: A Tool for Improving Privacy Through Enterprise Risk Management, Version 1.0*. https://doi.org/10.6028/NIST.CSWP.01162020 ↩

.8. Prometheus. (2026). *Metric and label naming*. https://prometheus.io/docs/practices/naming/. Consultado el 7 de junio de 2026. ↩

.9. Breck, E., Cai, S., Nielsen, E., Salib, M. y Sculley, D. (2017). The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. *IEEE Big Data*, 1123-1132. https://research.google/pubs/pub46555/ ↩

:0. Baye, C. A. (2016). *Emergency Response*. En *Site Reliability Engineering*. https://sre.google/sre-book/emergency-response/. Consultado el 7 de junio de 2026. ↩

:1. Stribblehill, A. (2016). *Managing Incidents*. En *Site Reliability Engineering*. https://sre.google/sre-book/managing-incidents/. Consultado el 7 de junio de 2026. ↩

:2. Cover, T. M. y Thomas, J. A. (2006). *Elements of Information Theory* (2.ª ed.). Wiley. https://dl.acm.org/doi/book/10.5555/1146355 ↩

:3. Siddiqi, N. (2006). *Credit Risk Scorecards: Developing and Implementing Intelligent Credit Scoring*. Wiley. ↩

24. Prometheus. (2026). *Metric and label naming*. https://prometheus.io/docs/practices/naming/. Consultado el 7 de junio de 2026. ↩

25. 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 7 de junio de 2026. ↩

26. 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 7 de junio de 2026. ↩

27. Baye, C. A. (2016). *Emergency Response*. En *Site Reliability Engineering*. https://sre.google/sre-book/emergency-response/. Consultado el 7 de junio de 2026. ↩

28. Stribblehill, A. (2016). *Managing Incidents*. En *Site Reliability Engineering*. https://sre.google/sre-book/managing-incidents/. Consultado el 7 de junio de 2026. ↩

29. Lunney, J. y Lueder, S. (2016). *Postmortem Culture: Learning from Failure*. En *Site Reliability Engineering*. https://sre.google/sre-book/postmortem-culture/. Consultado el 7 de junio de 2026. ↩