Facsímil 06

Construir y operar

Capítulo 04

Observabilidad: logs, métricas, trazas y costes

686f6c61 · 2026

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

Qué deberías poder hacer al terminar

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

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

Al terminar, deberías poder hacer esto:

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

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

El problema que venimos a resolver

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

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

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

Qué no es observabilidad

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

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

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

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

Qué sí es observabilidad en IA

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

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

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

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

Las tres señales principales

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

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

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

Una forma de pensarlo:

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

La traza de una run de IA

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

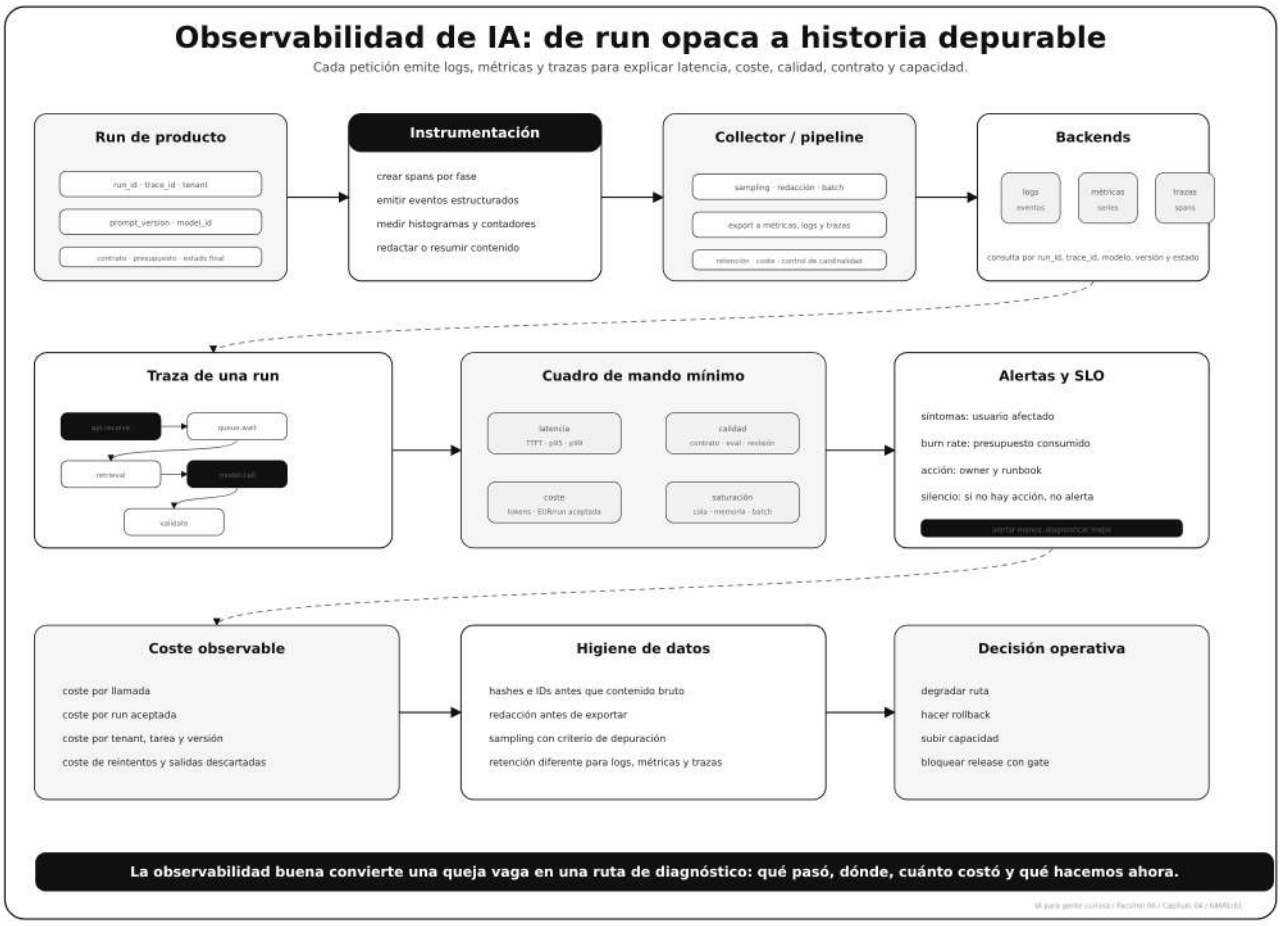
```
trace_id = trace_8f21...
run_id   = run_20260527_0042

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

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

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

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

Anatomía visual de observabilidad para IA



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

SLIs y SLOs para sistemas de IA

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

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

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

La fórmula del SLI de éxito operativo puede ser:

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

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

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

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

Para coste:

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

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

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

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

Burn rate: gastar el presupuesto de error

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

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

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

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

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

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

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

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

Métricas sin volverse loco con la cardinalidad

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

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

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

Nombres razonables:

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

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

Logs: detalle sin ruido

Un log útil en IA debería ser estructurado. No:

```
algo falló llamando al modelo
```

Mejor:

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

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

Una regla práctica:

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

Costes observables

El coste en IA tiene varias capas:

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

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

Comparación sencilla:

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

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

Sampling y retención

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

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

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

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

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

De métricas a decisiones

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

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

La ley de Little vuelve a aparecer:

$$L = \lambda W$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Correlacionar por versión

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

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

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

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

Dashboard mínimo que sí usaría

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

Yo lo diseñaría con seis paneles:

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

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

Alertas concretas, no ruido

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

Algunas reglas razonables para empezar:

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

Ejemplo expresado como pseudorregla:

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

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

Runbook operativo

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

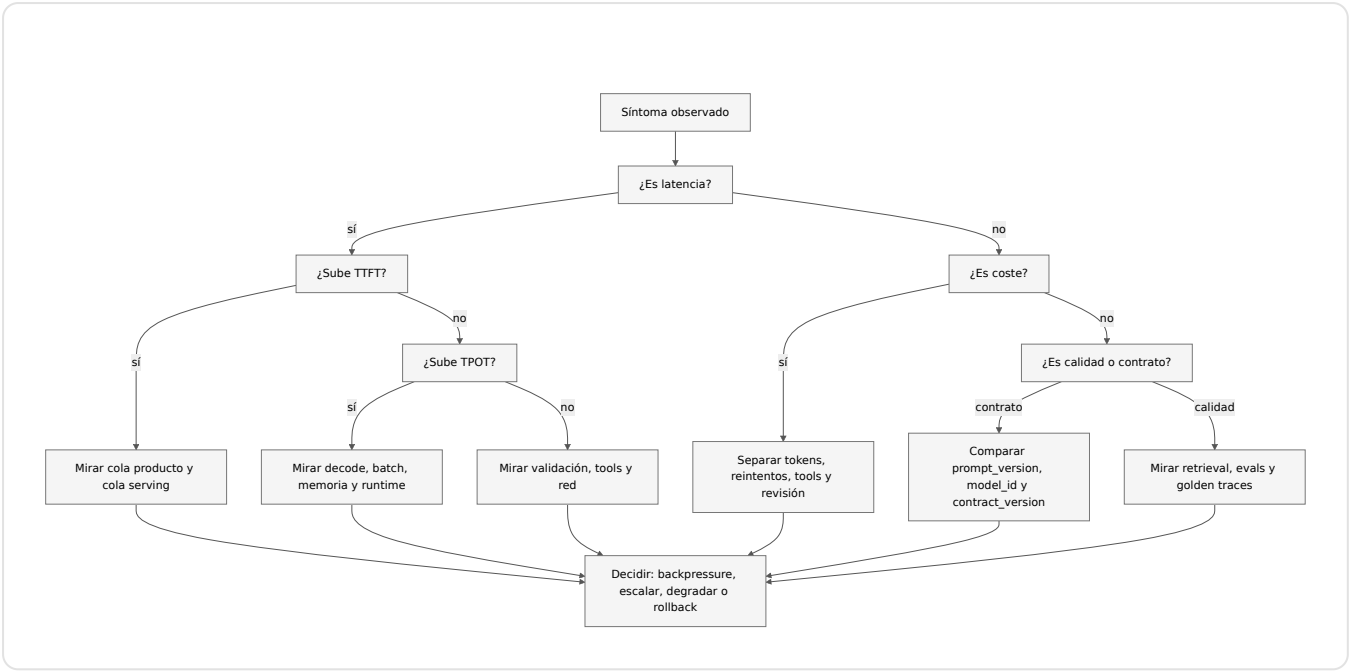
Para este capítulo, un runbook mínimo podría ser:

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

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

Árbol de diagnóstico

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



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

Golden traces

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

Ejemplos:

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

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

Observabilidad específica de RAG y tools

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

Para RAG, yo exigiría:

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

Para tools:

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

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

Coste de observar

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

Costes típicos:

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

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

Herramientas que se usan para observar sistemas de IA

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

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

Estándar e instrumentación

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

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

Backends generales de métricas, logs y trazas

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

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

Herramientas específicas de LLM, RAG y agentes

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

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

Cómo elegir sin perderse

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

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

Lo que ninguna herramienta arregla por ti

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

En el día a día

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

Una checklist mínima para producción:

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

Por qué debería importarte

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

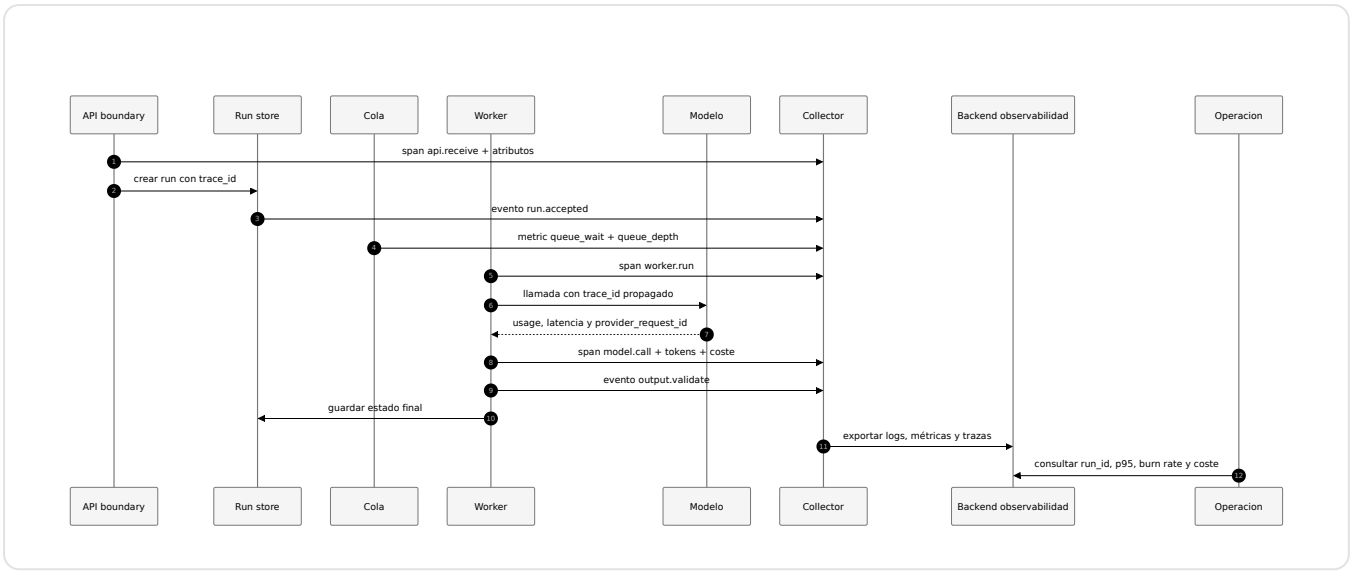
Con observabilidad, las conversaciones se vuelven verificables: “el 70% de las runs caras vienen de `task=legal_summary` con contexto p95 de 42.000 tokens”, “la versión `prompt@1.8` duplicó `contract_failed`”, “el p99 subió por cola de serving, no por retrieval”, “el coste por aceptada bajó aunque subió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

Dónde solía tropezar yo

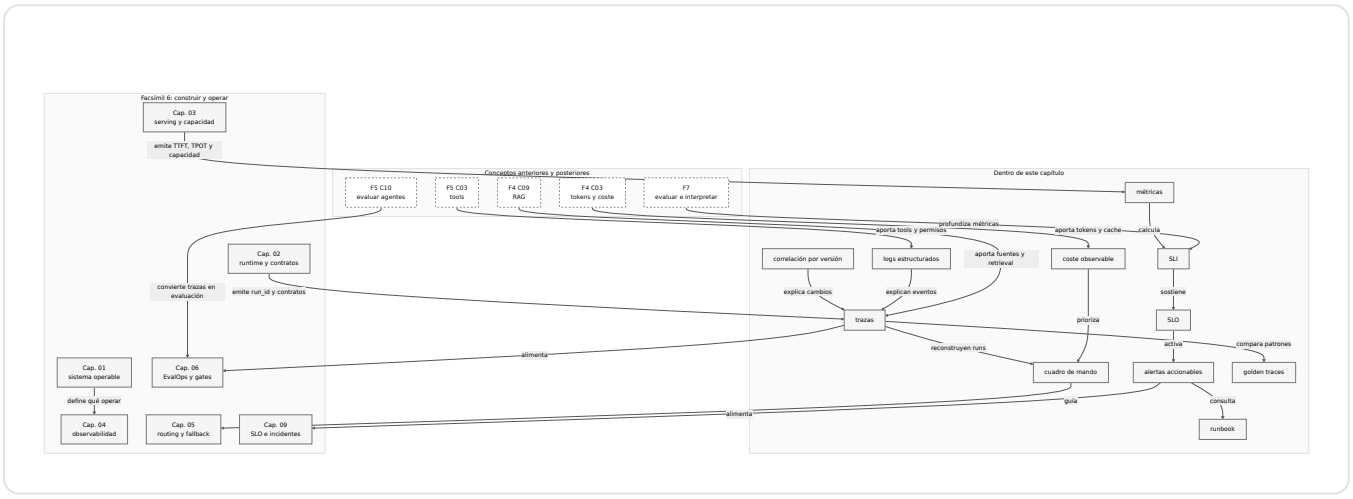
TROPIEZO	POR QUÉ ES UN PROBLEMA	ANTÍDOTO
Guardar demasiado contenido	Complica privacidad, coste y lectura.	Guardar IDs, hashes, tamaños, versiones y muestras controladas.
Medir solo la media	La media oculta la cola de usuarios que peor lo pasan.	Usar histogramas, p95, p99 y trazas lentas.
Meter `run_id`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

Cómo encaja todo

Primero, el flujo de telemetría de una run:



Y el mapa conceptual:



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

Vocabulario aprendido

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

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

Antes de pasar página

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

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

En resumen

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

Para saber más

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

Notas

1. Ewaschuk, R. (2016). *Monitoring Distributed Systems*. En B. Beyer, C. Jones, J. Petoff y N. R. Murphy (eds.), *Site Reliability Engineering*. https://sre.google/sre-book/monitoring-distributed-systems/. Consultado el 27 de mayo de 2026. ↩

2. OpenTelemetry. (2026). *Traces*. https://opentelemetry.io/docs/concepts/signals/traces/. Consultado el 27 de mayo de 2026. ↩

3. OpenTelemetry. (2026). *Tracing API*. https://opentelemetry.io/docs/specs/otel/trace/api/. Consultado el 27 de mayo de 2026. ↩

4. OpenTelemetry. (2026). *Semantic Conventions for Generative AI Systems*. https://opentelemetry.io/docs/specs/semconv/gen-ai/. Consultado el 27 de mayo de 2026. ↩

5. OpenTelemetry. (2026). *Logs*. https://opentelemetry.io/docs/concepts/signals/logs/. Consultado el 27 de mayo de 2026. ↩

6. OpenTelemetry. (2026). *Metrics*. https://opentelemetry.io/docs/concepts/signals/metrics/. Consultado el 27 de mayo de 2026. ↩

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

8. Ewaschuk, 2016. ↩

9. Wilkinson, J. (2018). *Alerting on SLOs*. En B. Beyer, N. R. Murphy, D. Rensin, K. Kawahara y S. Thorne (eds.), *The Site Reliability Workbook*. https://sre.google/workbook/alerting-on-slos/. Consultado el 27 de mayo de 2026. ↩

10. Prometheus. (2026). *Metric and label naming*. https://prometheus.io/docs/practices/naming/. Consultado el 27 de mayo de 2026. ↩

11. Dean, J. y Barroso, L. A. (2013). *The Tail at Scale*. Communications of the ACM, 56(2), 74-80. https://doi.org/10.1145/2408776.2408794. ↩

12. Little, J. D. C. (1961). *A Proof for the Queuing Formula: L = λW*. Operations Research, 9(3), 383-387. https://doi.org/10.1287/opre.9.3.383. ↩

13. Apdex Alliance; Sevcik, P. (2005). *Defining the Application Performance Index*. El índice Apdex es un estándar abierto para resumir la satisfacción de rendimiento en un valor entre 0 y 1. ↩

14. Cochran, W. G. (1977). *Sampling Techniques* (3.ª ed.). Wiley. Texto de referencia sobre el error estándar de una proporción estimada a partir de una muestra. ↩

.5. Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8), 861-874. https://doi.org/10.1016/j.patrec.2005.10.010. Relaciona sensibilidad, tasa de falsos positivos y prevalencia mediante el teorema de Bayes. ↩

.6. Dunning, T. (2021). The t-digest: Efficient estimates of distributions. *Software Impacts*, 7, 100049. https://doi.org/10.1016/j.simpa.2020.100049. Describe el t-digest para estimar cuantiles en flujos de datos de forma agregable y precisa en las colas. ↩

Cuadernos para practicar

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

Observabilidad: la caja negra que sí puedes mirar

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