Facsímil 06

Construir y operar

Capítulo 06

EvalOps y gates de release

686f6c61 · 2026

Capítulo 06: EvalOps y gates de release

Qué deberías poder hacer al terminar

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

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

Al terminar, deberías poder hacer esto:

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

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

El cambio pequeño que no era pequeño

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

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

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

EvalOps pone una pregunta delante de cada cambio:

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

Qué no es EvalOps

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

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

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

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

Qué sí es EvalOps

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

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

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

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

Fecha de corte del estado del arte

Fecha de corte: 28 de mayo de 2026.
Fuentes consultadas: documentación oficial de OpenAI sobre evals, graders, agent evals y trace grading; documentación de Google ADK sobre evaluación de agentes; LangSmith Evaluation; Arize Phoenix Evaluation; Ragas metrics; Braintrust Evaluation; Promptfoo; OpenTelemetry; Google SRE Workbook; y trabajos clásicos de ingeniería de ML como TFX y *Software Engineering for Machine Learning*.

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

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

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

Qué evaluar según lo que cambias

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

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

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

CI/CD real: dónde vive EvalOps

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

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

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

Un ejemplo mínimo en GitHub Actions:

```yaml
name: evalops

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

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

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

Qué debe importarte del YAML:

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

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

```
stages:
  - test
  - eval
  - release

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

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

Contrato JSONL de un caso de evaluación

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

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

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

Campos mínimos:

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

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

Los datasets no son una carpeta de ejemplos

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

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

No todos los datasets cumplen el mismo papel:

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

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

Evaluadores como software

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

Tipos de evaluadores:

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

Qué debe tener un evaluador serio:

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

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

```
CALIBRATION = [
    {
        "case_id": "judge-cal-001",
        "input": "Respuesta con cita correcta y fuente recuperada.",
        "expected_label": "pass",
    },
    {
        "case_id": "judge-cal-002",
        "input": "Respuesta segura pero sin fuente recuperada.",
        "expected_label": "fail",
    },
]

def test_evaluator_calibration(evaluator):
    labels = [evaluator(item["input"]).label for item in CALIBRATION]
    expected = [item["expected_label"] for item in CALIBRATION]
    assert labels == expected
```

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

La comparación baseline contra candidate

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

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

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

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

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

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

Estadística mínima para no engañarnos

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

El primer indicador de inestabilidad es el flake rate:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Comparación pareada: misma entrada, dos versiones

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

La comparación pareada ejecuta cada caso contra baseline y candidate:

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

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

Después clasificamos cada caso:

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

Scorecard pareada:

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

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

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

Matriz de criticidad por tarea

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

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

Variables para decidir criticidad:

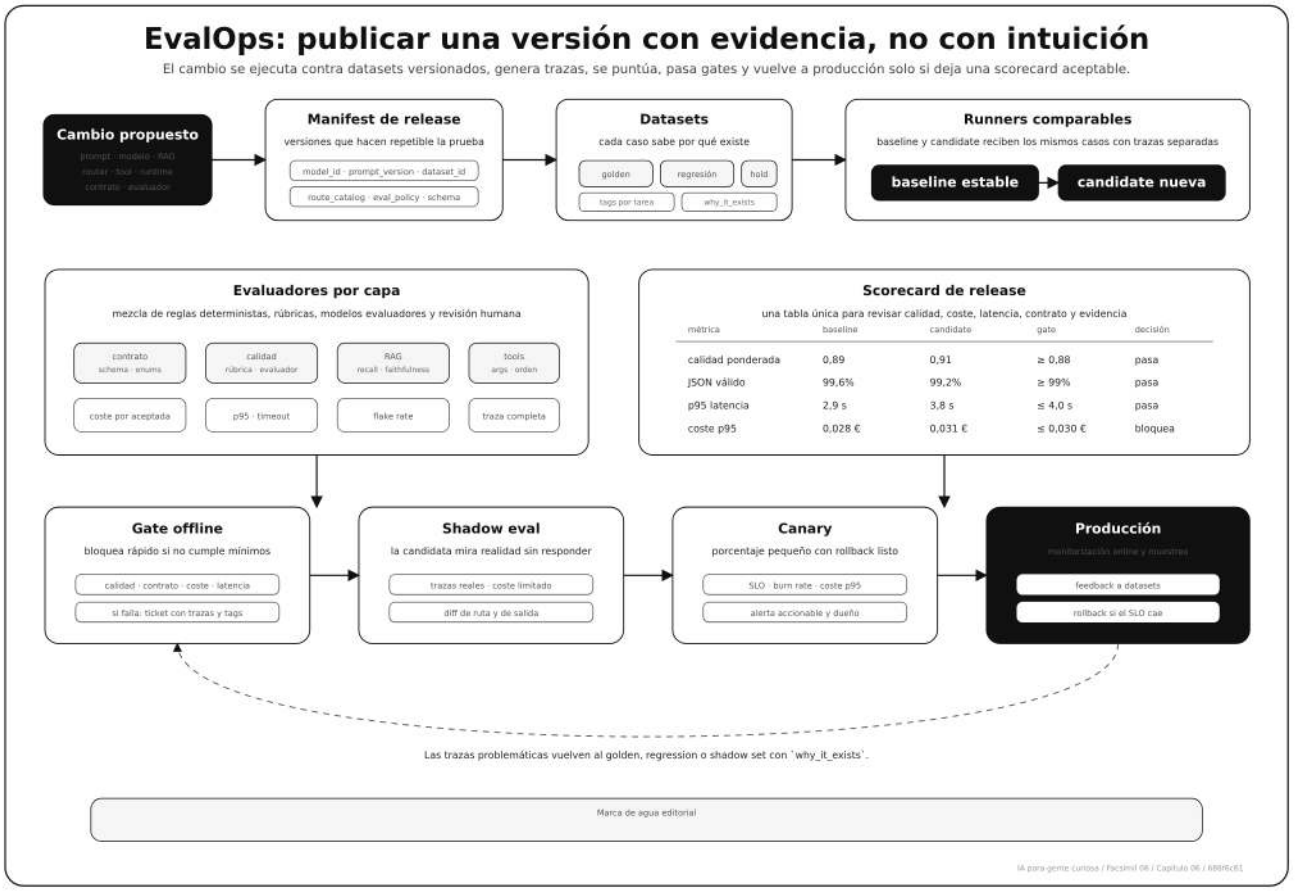
VARIABLE	PREGUNTA
Reversibilidad	¿Se puede deshacer sin daño operativo?
Dependencia downstream	¿Otra parte del sistema confía en esta salida?
Contrato	¿Hay schema, tipo, enum o cita obligatoria?
Exposición de datos	¿La run toca datos internos, personales o documentos sensibles?
Coste de error	¿Qué pasa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

Una política simple:

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

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

Anatomía visual de un pipeline EvalOps



Gates por entorno

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

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

La clave es que cada entorno responda una pregunta distinta:

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

Herramientas: qué aporta cada una

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

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

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

Coste de evaluar

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

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

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

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

Palancas de optimización:

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

Una política útil:

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

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

Privacidad y datasets de evaluación

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

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

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

Campos recomendables:

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

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

Scorecard: el documento que autoriza avanzar

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

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

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

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

Runbook cuando falla un gate

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

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

Plantilla de incidencia de gate:

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

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

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

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

En el día a día

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

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

Por qué debería importarte

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

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

Dónde solía tropezar yo

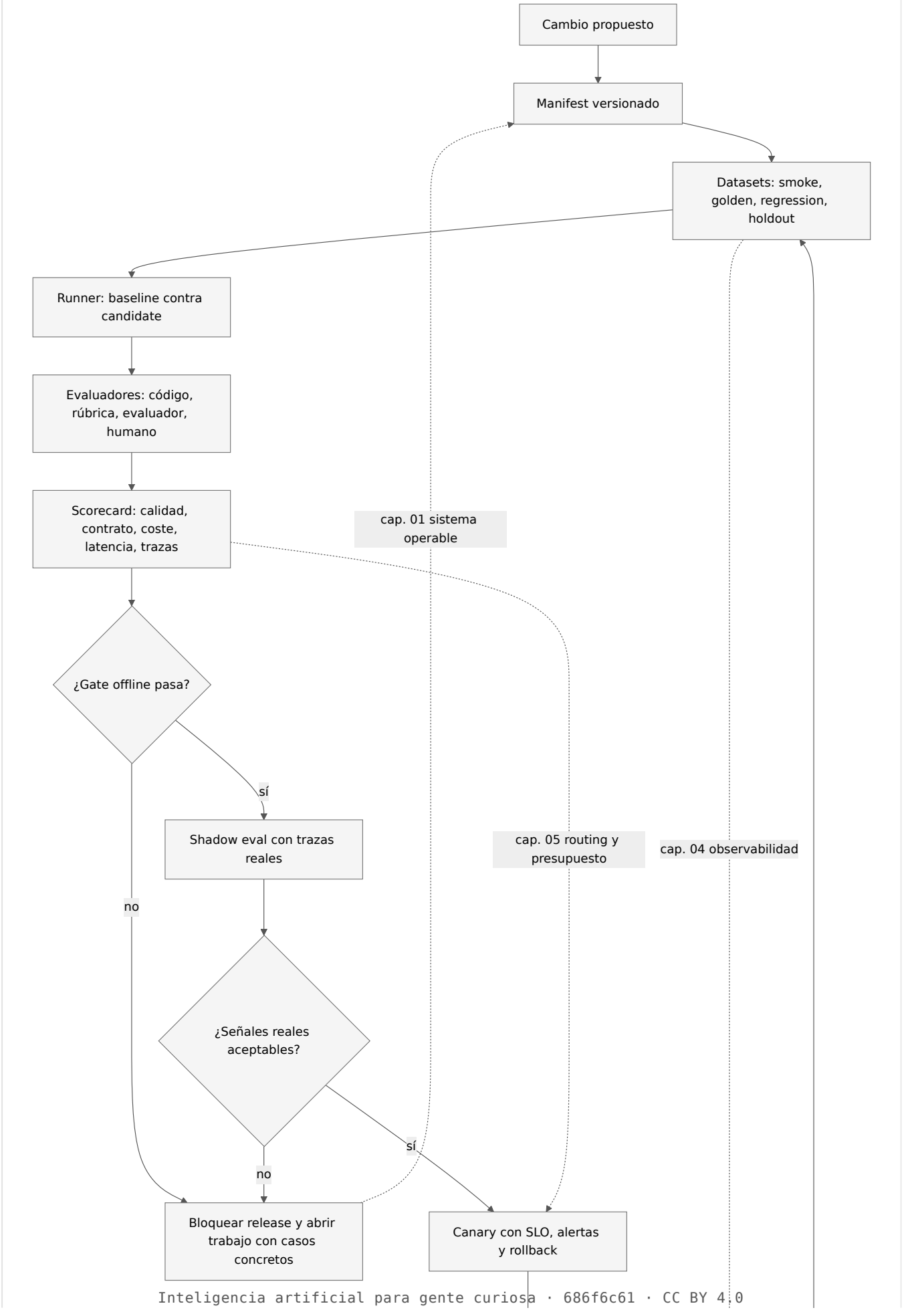
Durante mucho tiempo confundía “tener evals” con “estar evaluando bien”. Tenía un dataset, corría un script y miraba un score. Sonaba serio, pero faltaban cosas.

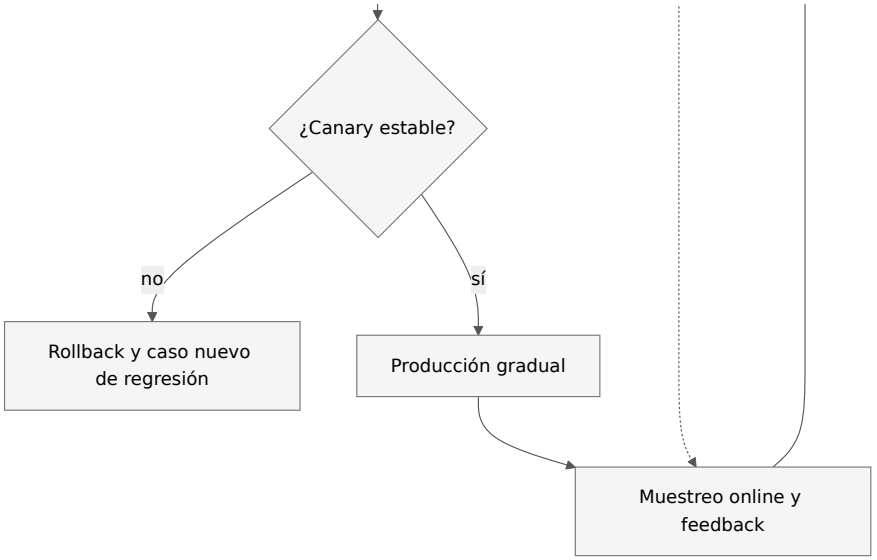
Me tropezaba en cinco sitios:

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

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

Cómo encaja todo


Cambio propuesto
Manifest versionado
Datasets: smoke, golden, regression, holdout
Runner: baseline contra candidate
Evaluadores: código, rúbrica, evaluador, humano
Scorecard: calidad, contrato, coste, latencia, trazas
cap. 01 sistema operable
¿Gate offline pasa?
sí
Shadow eval con trazas reales
cap. 05 routing y presupuesto
cap. 04 observabilidad
no
¿Señales reales aceptables?
sí
no
Bloquear release y abrir trabajo con casos concretos
Canary con SLO, alertas y rollback



Relación con otros capítulos

EvalOps se apoya en piezas que ya hemos construido:

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

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

Para entenderlo

Piensa en tres situaciones:

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

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

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

Vocabulario aprendido

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

Antes de pasar página

Comprueba que puedes responder sin mirar:

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

En resumen

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

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

Para saber más

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

Notas

1. OpenAI. (2026). *Working with evals*. https://developers.openai.com/api/docs/guides/evals. Consultado el 28 de mayo de 2026. ↩

2. Google. (2026). *Why Evaluate Agents*. https://adk.dev/evaluate/. Consultado el 28 de mayo de 2026. ↩

3. LangChain. (2026). *LangSmith Evaluation*. https://docs.langchain.com/langsmith/evaluation. Consultado el 28 de mayo de 2026. ↩

4. Arize Phoenix. (2026). *Evaluation concepts*. https://arize.com/docs/phoenix/evaluation/concepts-evals/evaluation. Consultado el 28 de mayo de 2026. ↩

5. Braintrust. (2026). *Evaluate systematically*. https://www.braintrust.dev/docs/evaluate. Consultado el 28 de mayo de 2026. ↩

6. Promptfoo. (2026). *Intro*. https://www.promptfoo.dev/docs/intro/. Consultado el 28 de mayo de 2026. ↩

7. GitHub. (2026). *Workflow syntax for GitHub Actions*. https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax. Consultado el 28 de mayo de 2026. ↩

8. GitLab. (2026). *CI/CD YAML syntax reference*. https://docs.gitlab.com/ci/yaml/. Consultado el 28 de mayo de 2026. ↩

9. Ragas. (2026). *List of available metrics*. https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/. Consultado el 28 de mayo de 2026. ↩

10. McNemar, Q. (1947). Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2), 153-157. https://doi.org/10.1007/BF02295996. Prueba para proporciones correlacionadas en diseños emparejados. ↩

11. Cohen, J. (1988). *Statistical Power Analysis for the Behavioral Sciences* (2.ª ed.). Lawrence Erlbaum. Relaciona tamaño del efecto, nivel de significación, potencia y tamaño de muestra. ↩

12. Clopper, C. J., & Pearson, E. S. (1934). The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4), 404-413. https://doi.org/10.1093/biomet/26.4.404. Intervalo de confianza exacto para una proporción binomial. ↩

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