Facsímil 06

Construir y operar

Capítulo 07

Cambios progresivos: shadow, canary y rollback

686f6c61 · 2026

Capítulo 07: Cambios progresivos: shadow, canary y rollback

Qué deberías poder hacer al terminar

En el capítulo 06 dejamos una idea clara: una versión candidata no debería llegar a producción solo porque parece mejor. Debe pasar por evidencia: dataset, evaluadores, scorecard y gates. Ahora toca la siguiente pregunta: **si el gate pasa, cómo la exponemos sin apostar todo el sistema a la primera tirada**.

Los sistemas de IA tienen una superficie de cambio más amplia que muchas aplicaciones clásicas. Puedes cambiar código, prompt, modelo, cuantización, runtime, catálogo de rutas, índice RAG, política de abstención, contrato JSON, tool schema, coste máximo o umbral de fallback. Muchos de esos cambios no se ven como una nueva pantalla, pero cambian lo que el sistema responde.

Al terminar, deberías poder hacer esto:

RESULTADO DE APRENDIZAJE	EVIDENCIA DE QUE LO SABES HACER
Separar deployment de release.	Puedes desplegar una versión sin exponerla todavía.
Diseñar shadow run.	Copias entradas reales sin producir efectos persistentes ni duplicar acciones.
Diseñar canary.	Repartes tráfico de forma estable, medible y reversible.
Definir criterios de avance.	Cada porcentaje tiene métricas, mínimos, duración y dueño.
Preparar rollback.	Sabes volver por flag, configuración, ruta, prompt o versión.
Elegir estrategia.	Distingues rolling, blue/green, dark launch, percentage rollout, guarded rollout y canary.
Construir un controlador mínimo.	Simulas asignación por cohorte y decisión de avanzar, pausar o volver.

La idea central: **un cambio serio no se “lanza”; se expone progresivamente, se mide por segmentos y conserva una salida de vuelta**.

La release que no debería ser un salto al vacío

Imagina que el asistente de soporte usa `prompt_v12`, `model_a`, `rag_index_2026_05` y `router_policy_31`. El capítulo 06 nos ayudó a decidir que una candidata, `prompt_v13` con `router_policy_32`, pasa el gate offline. Mejora respuestas largas y reduce llamadas caras. Bien.

Pero producción no es el dataset. En producción hay usuarios con preguntas raras, tenants con documentos distintos, momentos de carga, límites de proveedor, colas, documentos recién subidos y prompts que combinan mal con ciertos casos. Si pasamos de 0% a 100%, cualquier sorpresa cae sobre todo el mundo.

Un rollout progresivo cambia la pregunta:

> ¿Qué porcentaje pequeño, durante cuánto tiempo y con qué métricas nos permite aprender sin comprometer todo el sistema?

Deployment no es release

En conversaciones de equipo se mezclan mucho estas dos palabras. Conviene separarlas:

CONCEPTO	QUÉ CAMBIA	EJEMPLO
Deployment	El código o configuración queda disponible.	Desplegamos `prompt_v13` y `router_policy_32` en producción, pero desactivados.
Release	La capacidad se expone a tráfico real.	El 5% de `support_summary` usa `prompt_v13`.

OpenFeature describe los feature flags como decisiones evaluadas en runtime que permiten alterar el comportamiento sin desplegar código nuevo.[1] Esa separación es la base de muchos cambios progresivos: el código puede estar desplegado y la release puede seguir controlada por una bandera, una regla de targeting o una política del router.

Kubernetes, por su parte, ofrece Deployments para gestionar Pods y ReplicaSets y actualizar el estado deseado de una carga de trabajo.[2] Eso resuelve una parte: cómo llegan contenedores nuevos. Pero una release de IA también puede depender de configuración viva: qué modelo se usa, qué prompt está activo, qué índice RAG se consulta y qué porcentaje de tráfico entra en la variante.

Qué no es un canary

Canary no es “subir al 10% y esperar”. Tampoco es una excusa para saltarse EvalOps. Si una candidata no pasó eval offline, canary no la convierte en segura; solo desplaza el experimento a producción.

Canary tampoco debería ser aleatorio en cada petición. Si el mismo usuario o tenant cae a veces en baseline y a veces en candidate, será difícil comparar, depurar y explicar diferencias. Para muchas tareas conviene asignación estable por cohorte: usuario, tenant, tarea, conversación o documento.

Y rollback no es “ya veremos cómo volvemos”. Rollback debe estar probado antes de exponer tráfico. En IA, volver puede significar restaurar un prompt, bajar un porcentaje, cambiar ruta, recuperar un índice, volver al contrato anterior o desactivar una tool.

CONFUSIÓN	QUÉ FALTA
“Canary es probar en producción”	Falta gate offline, porcentaje, métricas, duración y rollback.
“Deployment y release son lo mismo”	Falta flag, targeting o política que se pueda cambiar sin redesplegar.
“Si falla, hacemos rollback”	Falta saber qué revertir, cuánto tarda y qué datos deja la versión candidata.
“Shadow es ejecutar dos veces y comparar”	Falta evitar efectos persistentes, duplicar coste sin límite o llamar tools de escritura.
“Subimos por usuarios al azar”	Falta cohortes estables y segmentación por tarea, tenant o criticidad.

Qué sí es progressive delivery en IA

Progressive delivery es una forma de publicar cambios reduciendo el radio de impacto. En IA generativa, el cambio puede estar en muchas capas:

CAPA	CAMBIO PROGRESIVO POSIBLE
Prompt	Activar `prompt_v13` por porcentaje, tarea o tenant.
Modelo	Comparar `model_a` y `model_b` con rutas separadas.
RAG	Probar un índice nuevo en shadow antes de usarlo como fuente.
Router	Ejecutar `router_policy_32` en modo decisión sombra.
Tool	Activar una tool primero en modo dry-run o solo lectura.
Contrato	Aceptar nuevo schema solo en consumidores preparados.
Runtime	Mover porcentaje a un serving nuevo con límites de capacidad.

Una release progresiva se describe con estas piezas. No es una ecuación de la literatura, es una lista de ingeniería:

SÍMBOLO	SIGNIFICADO	EJEMPLO
v_b	Versión baseline.	`support-rag@1.8.0`.
v_c	Versión candidata.	`support-rag@1.9.0-rc1`.
W	Secuencia de pesos de tráfico.	0%, shadow, 1%, 5%, 25%, 50%, 100%.
M	Métricas observadas por paso.	p95, coste, contrato, aceptación, fallback, calidad online.
G	Gates de avance.	Condiciones para pasar de un peso al siguiente.
B	Plan de rollback.	Flag, configuración previa, ruta anterior e índice anterior.

Argo Rollouts define canary como una estrategia donde una versión nueva recibe un porcentaje pequeño del tráfico de producción, con pasos como `setWeight` y `pause` para controlar el avance.[3] LaunchDarkly separa opciones como percentage rollouts, progressive rollouts, guarded rollouts y experiments según si se quiere asignar porcentaje fijo, aumentar tráfico con el tiempo, vigilar métricas o comparar variantes.[4] Google Cloud Deploy describe canary como un despliegue progresivo que divide tráfico entre una versión ya desplegada y una nueva antes de llegar al despliegue completo.[5]

Estrategias de cambio progresivo

No existe una única estrategia buena. Hay familias, y cada una protege algo distinto.

ESTRATEGIA	QUÉ HACE	SIRVE CUANDO	CUIDADO
Rolling update	Sustituye instancias gradualmente.	Cambio de código con compatibilidad clara.	No controla comportamiento por usuario o tarea.
Blue/green	Mantiene dos entornos y conmuta tráfico.	Necesitas vuelta rápida a entorno anterior.	Duplica infraestructura y no mide bien por segmentos.
Dark launch	Despliega sin exponer al usuario.	Quieres probar wiring, dependencias y observabilidad.	No prueba aceptación real.
Shadow run	Ejecuta candidate en paralelo sin afectar salida.	Quieres comparar con entradas reales.	Debe evitar efectos persistentes y coste ilimitado.
Percentage rollout	Envía un porcentaje fijo a candidate.	Quieres muestra controlada.	Sin métricas puede ser solo azar administrado.
Progressive rollout	Sube porcentaje por pasos.	Quieres aprender antes de ampliar.	Cada paso necesita criterio de avance.
Guarded rollout	Sube tráfico vigilando métricas.	Quieres detener si empeoran señales.	Depende de métricas fiables y umbrales claros.
Canary por cohorte	Expone tenant, tarea o segmento específico.	Quieres controlar impacto por dominio.	Puede sesgar resultados si la cohorte no representa bien.

En IA, muchas veces combinamos varias. Por ejemplo:

1. Dark launch de `model_b` en el runtime.
2. Shadow run con entradas reales de `support_summary`.
3. Canary al 1% por cohorte estable.
4. Progressive rollout 5%, 25%, 50%.
5. Guarded rollout con rollback automático si contrato, p95 o coste empeoran.

Matriz de elección de estrategia

La pregunta práctica no es “qué patrón está de moda”. La pregunta buena es: **qué necesito aprender sin exponer más superficie de la necesaria**. Esta matriz sirve para elegir.

CAMBIO	RIESGO PRINCIPAL	ESTRATEGIA PREFERENTE	POR QUÉ
Prompt de respuesta libre	Calidad, tono, longitud y coste.	Shadow y canary por tarea.	Puedes comparar salidas y después exponer solo una tarea.
Prompt con JSON estricto	Fallo de contrato.	Shadow con validador y canary pequeño.	El contrato debe romperse antes de llegar al usuario.
Modelo nuevo	Latencia, coste, calidad y límites de proveedor.	Shadow, canary por cohorte y fallback.	La misma entrada puede comportarse distinto en tokens, rutas y coste.
Índice RAG nuevo	Recuperación peor o fuentes incorrectas.	Shadow de retrieval y canary por tenant.	La salida puede parecer correcta aunque la fuente haya cambiado.
Tool de lectura	Latencia, permisos y formato de argumentos.	Dark launch, dry-run y canary por tarea.	Conviene probar wiring y trazas antes de exponer.
Tool de escritura	Efectos persistentes.	No usar shadow real con escritura; empezar con modo aprobación o dry-run.	La versión candidata no debe duplicar acciones.
Runtime de inferencia	Saturación, colas y p95.	Blue/green o canary de tráfico.	Aquí importa tanto la arquitectura como el modelo.
Schema de salida	Compatibilidad con consumidores.	Expand-contract.	Los consumidores antiguos y nuevos deben convivir durante la migración.

Un alumno debería poder justificar su elección así: “uso shadow porque necesito comparar entradas reales sin cambiar salida; uso canary por tenant porque el índice RAG depende de documentos; uso rollback por flag porque el cambio está en prompt y ruta, no en código”.

Compatibilidad: dos versiones conviven

Una release progresiva no solo reparte tráfico. Durante un rato conviven dos mundos: baseline y candidate. Esa convivencia exige compatibilidad.

SUPERFICIE	QUÉ PUEDE ROMPER	REGLA DE COMPATIBILIDAD
JSON de respuesta	Consumidores esperan campos antiguos.	Añadir campos antes de quitar campos.
Tool schema	El modelo manda argumentos nuevos.	Versionar schema y aceptar ambos durante la transición.
Índice RAG	Citas apuntan a documentos con IDs distintos.	Mantener alias o mapa de documentos.
Prompt template	Cambia el formato esperado por evaluadores.	Versionar plantilla y scorecard.
Modelo	Cambia tokenizer, límites o tool calling.	Registrar `model_id`, `tokenizer_id` y límites por release.
Base de datos	Candidate escribe datos que baseline no entiende.	Usar migración expand-contract.

La regla expand-contract es sencilla:

1. Expandir: añadir lo nuevo sin retirar lo antiguo.
2. Migrar: mover tráfico y consumidores progresivamente.
3. Contraer: quitar lo antiguo cuando ya no se usa y hay evidencia.

Checklist mínimo antes de canary:

PREGUNTA	RESPUESTA ESPERADA
¿Baseline puede leer lo que candidate escribe?	Sí, o candidate no escribe todavía.
¿Candidate puede leer datos antiguos?	Sí.
¿El contrato JSON tiene versión?	Sí, `schema_version` o equivalente.
¿Los dashboards separan baseline y candidate?	Sí, por `variant` y `release_id`.
¿El rollback sabe qué hacer con datos nuevos?	Sí, ignorar, transformar o bloquear.

Asignación estable por cohorte

Para repartir tráfico no basta con llamar a `random()`. Necesitamos estabilidad. Si una conversación está en candidate, debe seguir en candidate mientras dure la prueba. Si un tenant entra en baseline, no debería saltar de variante por cada request.

Una forma común es usar hash determinista:

$$b(c, f) = \frac{H(c \| f) \bmod 10000}{100}$$

$$variant(c) = \begin{cases} candidate & \text{si } b(c, f) < w \\ baseline & \text{si } b(c, f) \geq w \end{cases}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
c	Clave de cohorte.	`tenant_42`, `user_91`, `conversation_abc`.
f	Nombre de la bandera o release.	`support_rag_v19`.
H	Función hash estable.	SHA-256, MurmurHash, xxHash.
$b(c, f)$	Bucket entre 0 y 100.	3,71.
w	Peso de tráfico para candidate.	5%.

Si $w = 5$ y el bucket es 3,71, ese contexto entra en candidate. Si mañana subimos a 25%, entrarán contextos nuevos, pero quienes ya estaban en candidate seguirán ahí. Esa propiedad hace que el rollout sea más depurable.

Qué medir en cada paso

El canary de una app clásica suele mirar errores HTTP, latencia, CPU y quizá conversión. En IA necesitamos más señales:

SEÑAL	QUÉ PREGUNTA RESPONDE
`contract_fail_rate`	¿La salida sigue cumpliendo schema, tipos y campos?
`quality_proxy`	¿La calidad online o muestreada no cae?
`acceptance_rate`	¿La salida se acepta, se usa o evita revisión?
`fallback_rate`	¿El sistema cae más en rutas alternativas?
`cost_p95_eur`	¿El coste por run aceptada se mantiene?
`latency_p95_ms`	¿La experiencia interactiva sigue dentro del SLO?
`timeout_rate`	¿Suben timeouts o cortes de generación?
`route_mix_delta`	¿El router deriva demasiado a rutas caras o lentas?
`trace_completeness`	¿La versión candidata deja trazas suficientes?
`manual_review_rate`	¿Aumenta la necesidad de revisión?

Estas señales no deberían vivir en un informe manual. Deben salir de métricas y trazas filtrables por `release_id`, `variant`, `rollout_step`, `task` y `tenant`. OpenTelemetry define las trazas como una forma de seguir el camino de una petición mediante spans relacionados.[6] Sus convenciones para GenAI añaden atributos propios de sistemas generativos.[7]

Para alertar durante canary, no basta con “parece que sube algo”. El SRE Workbook de Google recomienda razonar con SLOs y presupuesto de error.[8] El libro de SRE también recuerda que proteger un servicio bajo carga implica decidir qué trabajo aceptar, retrasar o rechazar.[9]

Consultas observables reales

Decir “mide latencia y coste” es insuficiente. Un equipo necesita consultas reproducibles. Prometheus define PromQL como un lenguaje funcional para seleccionar y agregar series temporales en tiempo real.[10] Con etiquetas como `release_id`, `variant`, `task` y `tenant_tier`, puedes preguntar cosas útiles.

Tasa de fallos de contrato por variante:

```
sum(rate(ai_contract_fail_total{release_id="support-rag@1.9.0-rc1"}[10m])) by (variant)
/
sum(rate(ai_run_total{release_id="support-rag@1.9.0-rc1"}[10m])) by (variant)
```

Latencia p95 de candidate:

```
histogram_quantile(
  0.95,
  sum(rate(ai_run_latency_seconds_bucket{
    release_id="support-rag@1.9.0-rc1",
    variant="candidate"
  }[10m])) by (le, task)
)
```

Coste p95 por tarea:

```
histogram_quantile(
  0.95,
  sum(rate(ai_run_cost_eur_bucket{
    release_id="support-rag@1.9.0-rc1"
  }[30m])) by (le, task, variant)
)
```

Ratio de fallback:

```
sum(rate(ai_fallback_total{release_id="support-rag@1.9.0-rc1"}[15m])) by (variant)
/
sum(rate(ai_run_total{release_id="support-rag@1.9.0-rc1"}[15m])) by (variant)
```

Si además guardas eventos analíticos en una tabla, la pregunta de negocio se puede revisar con SQL:

```
select
  variant,
  task,
  count(*) as runs,
  avg(case when accepted_by_user then 1 else 0 end) as acceptance_rate,
  avg(cost_eur) as avg_cost_eur,
  percentile_cont(0.95) within group (order by latency_ms) as latency_p95_ms
from ai_release_runs
where release_id = 'support-rag@1.9.0-rc1'
  and created_at >= now() - interval '2 hours'
group by variant, task
order by task, variant;
```

La señal importante para un ingeniero: no basta con instrumentar. Hay que diseñar nombres, etiquetas y cardinalidad para que las consultas salgan limpias cuando la release esté viva.

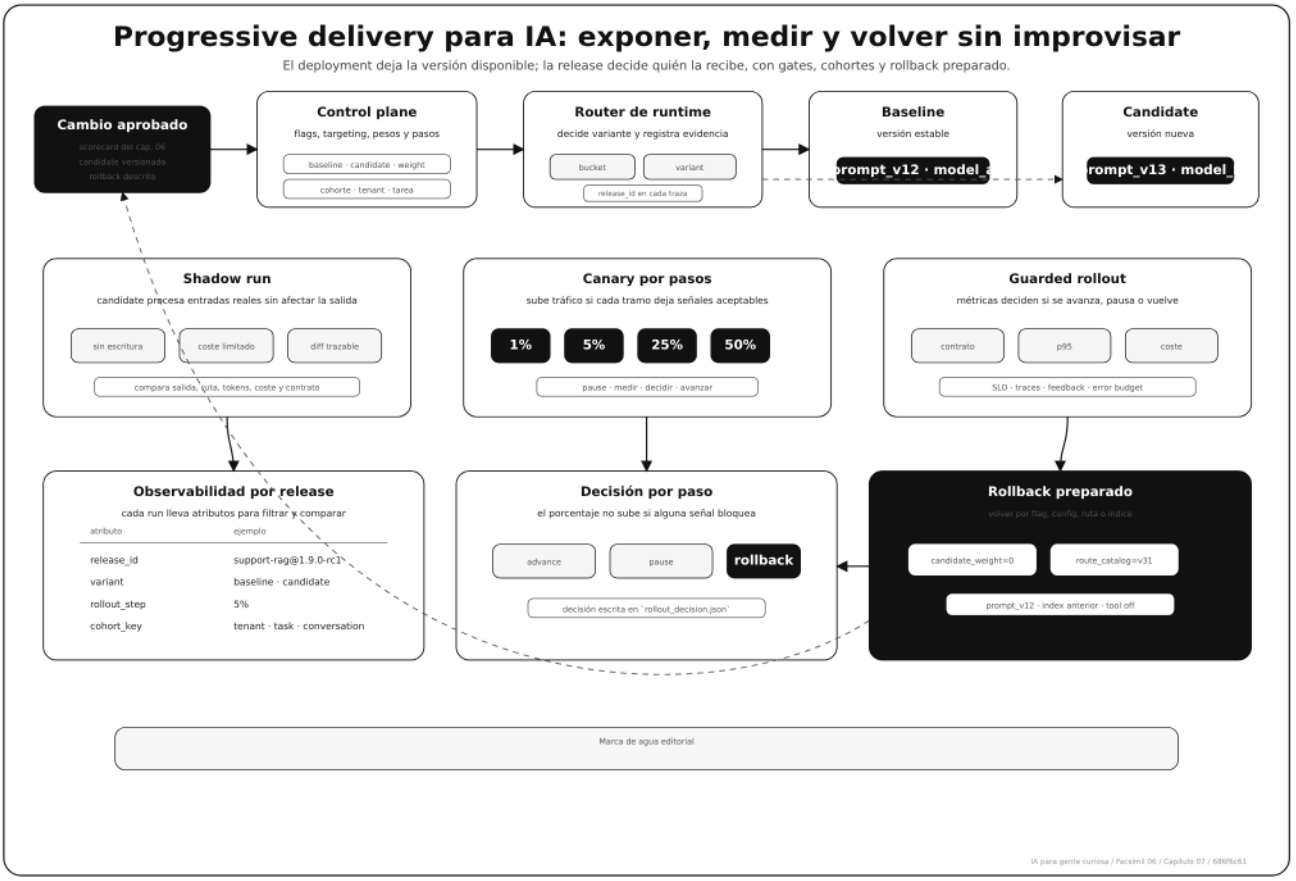
El gate de avance de cada paso es una función indicador: solo se sube al siguiente porcentaje de tráfico si se cumplen todas las condiciones:

$$A_s = 1[K_s \le K_{max} \wedge L_{95,s} \le L_{max} \wedge C_{95,s} \le C_{max} \wedge Q_s \ge Q_{min} \wedge T_s \ge T_{min}]$$

SÍMBOLO	SIGNIFICADO
A_s	Decisión de avanzar en el paso s.
K_s	Tasa de fallos de contrato en el paso.
K_{max}	Máximo tolerado de fallos de contrato.
$L_{95,s}$	Latencia p95 observada.
L_{max}	Límite p95 del SLO.
$C_{95,s}$	Coste p95 por run aceptada.
C_{max}	Presupuesto máximo permitido.
Q_s	Calidad online, evaluación muestreada o proxy aceptado.
Q_{min}	Calidad mínima para avanzar.
T_s	Proporción de trazas completas.
T_{min}	Mínimo de trazabilidad aceptable.

Si $A_s = 1$, se puede avanzar al siguiente porcentaje. Si $A_s = 0$, no se avanza. Dependiendo del motivo, se pausa, se baja porcentaje o se vuelve a baseline.

Anatomía visual de un rollout progresivo de IA



Shadow run sin efectos persistentes

Shadow run significa que la candidata recibe entradas reales, pero su salida no se entrega al usuario. Esto permite comparar sin cambiar la experiencia principal. En IA hay que diseñarlo con cuidado.

Reglas para shadow:

REGLA	MOTIVO
No ejecutar tools con escritura.	Evita duplicar tickets, correos, pagos, publicaciones o cambios de estado.
Usar dry-run cuando exista.	Permite validar argumentos sin tocar sistemas externos.
Marcar cada traza como `shadow=true`.	Facilita filtrar métricas y coste.
Limitar coste por día.	Shadow puede duplicar llamadas de modelo.
No mezclar salida sombra con feedback de usuario.	El usuario no vio esa salida; no puede aceptarla ni rechazarla.
Comparar por caso pareado.	La misma entrada debe tener baseline y candidate.

La salida sombra sí sirve para medir:

COMPARACIÓN	QUÉ PUEDE REVELAR
Respuesta final	Candidate responde más largo, más corto, más caro o menos grounded.
Contrato	Candidate rompe JSON o cambia campos.
Routing	Candidate elige rutas más caras o lentas.
Tools	Candidate intenta una tool que baseline no necesitaba.
Tokens	Candidate consume más contexto o salida.
Latencia	Candidate empeora prefill, decode o validación.

Shadow no prueba aceptación real. Prueba compatibilidad con realidad. Es una fase de ingeniería, no una encuesta.

Canary para IA: qué segmentos elegir

El 1% global puede sonar prudente, pero no siempre enseña lo correcto. Si tu producto tiene tareas muy distintas, el canary debe segmentarse.

SEGMENTO	CUÁNDO USARLO	EJEMPLO
Por tarea	Cada tarea tiene riesgo y coste distintos.	`support_summary`, `policy_qa`, `json_extract`.
Por tenant	Los documentos o normas cambian por cliente.	3 tenants internos antes de externos.
Por idioma	La calidad puede variar mucho.	Español primero si el dataset está mejor cubierto.
Por canal	No es igual chat interactivo que batch.	Canary en batch antes que en chat.
Por contrato	JSON estricto merece gates más duros.	Activar solo en respuestas no estructuradas al principio.
Por criticidad	Tareas de bajo impacto primero.	Resumen interno antes que acción externa.

Un buen canary no busca la muestra más cómoda. Busca la muestra más informativa sin ampliar demasiado el impacto.

Tamaño mínimo de muestra y duración

Una release progresiva necesita suficientes observaciones para no decidir por ruido. El número exacto depende de la varianza, la criticidad y el volumen, pero conviene declarar mínimos antes de empezar.

FACTOR	QUÉ CAMBIA
Tráfico bajo	Necesitas más tiempo para llegar a un mínimo útil.
Salida muy variable	Necesitas más casos o segmentar por tarea.
Tarea crítica	Necesitas gates más estrictos y porcentaje más pequeño.
Coste alto	Shadow y canary deben tener presupuesto diario.
Contrato estricto	Un fallo puede pesar más que una media de calidad.
Tenants heterogéneos	El 1% global puede no representar a nadie.

Una pauta razonable para empezar:

PASO	MÍNIMO ORIENTATIVO	DURACIÓN MÍNIMA	QUÉ MIRARÍA ANTES DE AVANZAR
Shadow	100-300 runs pareadas.	1 ciclo de carga real.	Diffs, contrato, coste y rutas.
Canary 1%	200-500 runs.	30-120 min según tráfico.	Contrato, p95, fallback y trazas.
Canary 5%	500-1500 runs.	1-4 h.	Segmentos por tarea y tenant.
Canary 25%	1000-5000 runs.	Medio día o 1 ciclo de uso.	Presupuesto de error y coste p95.
Canary 50%	Depende del negocio.	1 ciclo relevante.	Comparación estable y sin regresiones nuevas.

Esto no sustituye estadística formal. Es una forma de no fingir precisión: si solo tienes 12 runs, una tasa del 0% no significa “cero problemas”; significa “no hemos mirado bastante”.

Cuándo parar el canary: el test secuencial de Wald. No hace falta fijar de antemano cuántas peticiones observar. El test secuencial de la razón de probabilidad acumula evidencia caso a caso y se detiene en cuanto cruza un umbral, lo que suele necesitar muchas menos muestras que un test de tamaño fijo.[11]

$$\Lambda_n = \prod_{i=1}^{n} \frac{p_1(x_i)}{p_0(x_i)}, \qquad B = \frac{\beta}{1-\alpha} \;<\; \Lambda_n \;<\; \frac{1-\beta}{\alpha} = A$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
Λ_n	Razón de verosimilitud acumulada tras n casos.	Sube si el canary parece malo.
p_1, p_0	Probabilidad del dato bajo «peor» y bajo «igual».	Modelos de error de cada hipótesis.
α, β	Errores tolerados de tipo I y II.	0,05 y 0,10.
A, B	Umbrales de decisión.	Parar arriba (malo) o abajo (bueno).

En palabras: mientras la evidencia esté en la zona dudosa, sigues observando; en cuanto cruza A cortas el canary, y en cuanto cruza B lo promueves. Esto es justo lo que quieres en un rollout, que un canary claramente malo se detenga pronto, sin gastar más usuarios de los necesarios para estar seguro.

¿Es el canary peor que la versión estable? Contraste de dos proporciones. La comparación natural de un canary es contra el grupo de control que sigue en la versión vieja. El contraste de dos proporciones dice si la diferencia de tasa de error entre ambos grupos es real o cabe en el azar del muestreo.[12]

$$z = \frac{\hat{p}_c - \hat{p}_v}{\hat{p}\,(1-\hat{p})\left(\frac{1}{n_c} + \frac{1}{n_v}\right)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\hat{p}_c, \hat{p}_v$	Tasa de error del canary y de la versión vieja.	0,04 y 0,03.
n_c, n_v	Tamaño de cada grupo.	2.000 y 18.000.
$\hat{p}$	Tasa combinada de ambos grupos.	0,031
z	Estadístico; $	z

En palabras: el canary y el control comparten el mismo tráfico y el mismo momento, así que comparar sus tasas aísla el efecto del cambio mejor que comparar contra el histórico. Si $|z|$ no supera 1,96, la «subida de errores» del canary puede ser ruido; si lo supera, es señal de rollback.

¿A qué velocidad gasto el margen? La tasa de consumo del presupuesto de error. Un canary puede no fallar el contraste y aun así estar quemando el presupuesto de error demasiado rápido. La tasa de consumo (burn rate) mide a qué ritmo se agota el margen del SLO, y permite alertar antes de quedarte sin él.[13]

$$b = \frac{1 - \mathrm{SLI}}{1 - \mathrm{SLO}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
SLI	Indicador medido en la ventana actual.	0,94
SLO	Objetivo comprometido.	0,99
b	Tasa de consumo del presupuesto.	$0{,}06/0{,}01 = 6$.

En palabras: una tasa de 1 gasta el presupuesto justo a tiempo; una tasa de 6 lo agota seis veces más rápido de lo sostenible. Alertar por tasa de consumo, combinando una ventana corta y una larga, dispara el rollback cuando el daño se acumula deprisa pero ignora picos

breves e inofensivos. Es la regla que evita tanto el rollback histérico como el tardío.

¿Se ha salido la métrica de lo normal? El gráfico de control de Shewhart. En sombra comparas una métrica nueva contra su comportamiento normal. El gráfico de control separa la variación natural de la anómala con una regla simple y centenaria: marca como fuera de control lo que se aleja más de tres desviaciones de la media.[14]

$$\mathrm{LCS},\ \mathrm{LCI} = \mu \pm 3\,\sigma$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
μ	Media histórica de la métrica en sombra.	1,8 s de latencia.
σ	Desviación típica histórica.	0,3 s.
$\mathrm{LCS}, \mathrm{LCI}$	Límites de control superior e inferior.	0,9 s y 2,7 s.

En palabras: mientras la métrica baile entre los límites, es ruido normal y no hay que tocar nada; si los cruza, ha pasado algo que merece mirarse. Esto evita el error más común al vigilar una sombra, reaccionar a cada subida menor, y a la vez no deja pasar las desviaciones que sí importan.

Rollback no siempre es volver código atrás

En IA, rollback tiene varias capas. A veces no necesitas revertir todo el deployment. Basta con cambiar una bandera o restaurar una configuración.

CAPA	CÓMO VUELVES
Prompt	`prompt_version=prompt_v12`.
Modelo	`model_id=model_a` o ruta anterior.
RAG	`index_version=rag_index_2026_05`.
Router	`route_catalog=route_catalog@31`.
Tool	`tool_enabled=false` o `mode=read_only`.
Contrato	`response_schema=schema_v4`.
Runtime	Enviar tráfico al pool anterior.
Código	Revertir deployment o cambiar imagen.

La decisión entre rollback y roll forward depende de la causa:

SITUACIÓN	MEJOR OPCIÓN
Cambio de prompt rompe contrato.	Rollback de prompt o schema gate.
Nuevo índice RAG trae documentos incorrectos.	Rollback de índice.
Runtime nuevo tiene p95 malo.	Volver al pool anterior.
Falla una condición menor y hay fix claro.	Roll forward con parche y canary nuevo.
No sabemos la causa.	Bajar a baseline, preservar trazas y analizar.

La frase importante: **rollback no es castigo; es una capacidad de diseño**.

Kill switch: parar sin reunión

Un kill switch es una decisión operativa preparada para detener una capacidad sin desplegar código. En IA generativa puede apagar una tool, bajar una variante a 0%, forzar modelo baseline o cambiar a modo solo lectura.

CAMPO	PARA QUÉ SIRVE
`enabled`	Permite cortar una capacidad completa.
`candidate_weight`	Baja exposición sin tocar deployment.
`read_only`	Mantiene lectura y bloquea escritura.
`fallback_model`	Fuerza proveedor o modelo estable.
`max_cost_eur_per_run`	Corta rutas caras antes de agotar presupuesto.
`reason`	Deja trazabilidad de la decisión.
`expires_at`	Evita que un apagado temporal se quede olvidado.

Plantilla mínima:

```
release_id: support-rag@1.9.0-rc1
controls:
  enabled: true
  candidate_weight: 5
  read_only: false
  fallback_model: model_a
  max_cost_eur_per_run: 0.04
  kill_switch:
    enabled: false
    reason: null
    owner: ai-platform
    expires_at: null
```

Ensayo obligatorio antes del canary:

1. Activar `kill_switch.enabled=true` en staging.
2. Comprobar que todo tráfico va a baseline.
3. Comprobar que no se ejecutan tools de escritura.
4. Ver en trazas `kill_switch=true`.
5. Volver a activar candidate y verificar que el sistema recupera el comportamiento esperado.

Lifecycle de flags: limpiar también es operar

Los flags son útiles, pero también crean deuda si se quedan para siempre. LaunchDarkly documenta un ciclo de vida de flags con estados como live, ready for code removal, ready to archive, archived, deprecated y deleted, y recomienda planificar la retirada durante la creación del flag.[15]

Un flag de IA debería nacer con metadatos:

METADATO	EJEMPLO	MOTIVO
`owner`	`ai-platform`	Alguien responde por la bandera.
`created_at`	`2026-05-28`	Permite detectar flags antiguos.
`expected_removal`	`2026-06-15`	La retirada se planifica desde el inicio.
`permanent`	`false`	Distingue rollout temporal de interruptor permanente.
`release_id`	`support-rag@1.9.0-rc1`	Une flag, trazas, scorecard y decisión.
`cleanup_issue`	`OPS-1842`	La limpieza entra en el backlog real.

Estados recomendados:

ESTADO	QUÉ SIGNIFICA	ACCIÓN
`planned`	Está definido, pero no se evalúa todavía.	Revisar contrato y rollback.
`shadow`	Candidate se ejecuta sin salida visible.	Medir coste y compatibilidad.
`canary`	Candidate recibe parte del tráfico.	Vigilar gates.
`launched`	Candidate ya es comportamiento por defecto.	Abrir tarea de retirada si era temporal.
`cleanup`	El flag ya no decide nada útil.	Quitar código muerto y archivar.
`permanent_control`	Se conserva como interruptor.	Documentar quién puede cambiarlo.

La disciplina no termina cuando el rollout llega al 100%. Termina cuando el código, los dashboards y la documentación dejan de arrastrar caminos que nadie usa.

En el día a día

Los cambios progresivos aparecen la primera vez que algo que pasó todas las pruebas falla igualmente en producción, porque la realidad trae entradas, tráfico y casos que ningún dataset cubre del todo. La respuesta operable no es «desplegar y rezar», sino soltar el cambio por etapas: primero en sombra (procesa tráfico real pero su salida no se usa, solo se compara), luego en canary (un porcentaje pequeño de usuarios reales), y solo después al cien por cien. Cada etapa es una pregunta con menos riesgo: ¿se comporta igual que la versión vieja con tráfico real antes de que un solo usuario lo note?

La otra cara es el rollback. En un prototipo, volver atrás es git revert y a correr; en un sistema de IA, el comportamiento depende de modelo, prompt, política, dataset y retrieval, así que «volver» significa restaurar una combinación concreta y probada de todos ellos. El día que algo se tuerce en producción, la diferencia entre tener un rollback de un clic y reconstruir a mano qué versión funcionaba es la diferencia entre cinco minutos y una tarde con el sistema degradado.

Por qué debería importarte

Porque desplegar de golpe un sistema no determinista es una de las formas más caras de aprender. Los cambios progresivos te dan lo que ninguna batería de tests puede dar del todo: evidencia con tráfico real antes de comprometerte, y una salida de emergencia probada para cuando la evidencia sea mala. No eliminan el riesgo, lo acotan a una fracción de usuarios y a una ventana de tiempo corta.

Y hay una promesa que solo puedes hacer si esto está montado: que un cambio malo no se convierte en una incidencia grave. Con sombra, canary y rollback, lo peor que pasa es que un canary se apague antes de extenderse; sin ellos, lo peor que pasa es un fallo total en producción que tardas en diagnosticar y más aún en revertir. Esa diferencia es, casi siempre, la que separa un equipo que duerme tranquilo de uno que no.

Dónde solía tropezar yo

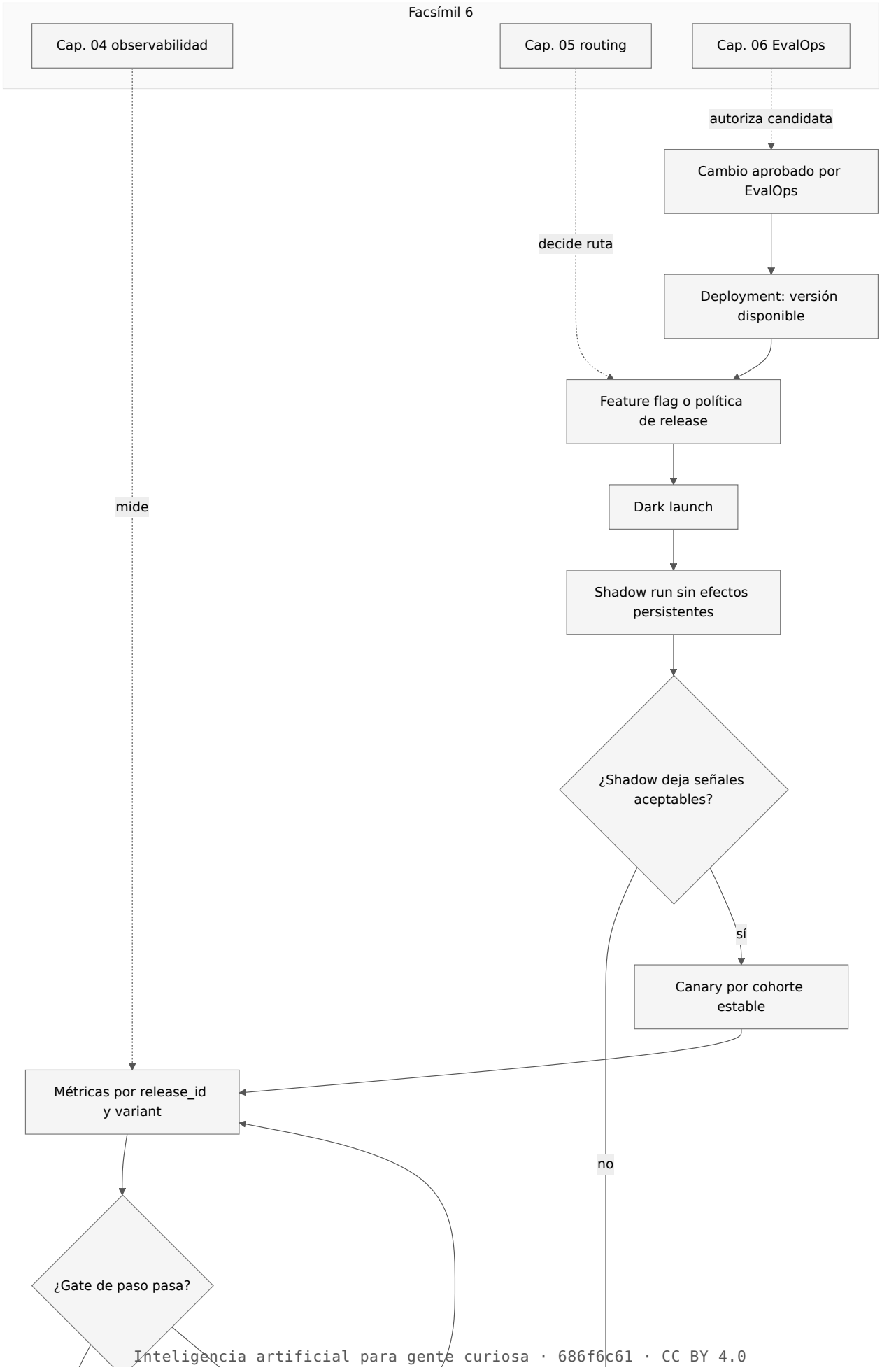
Me costó entender que canary no es una fase decorativa. Si no hay métricas, cohortes, gates y rollback, solo has cambiado el tamaño del salto.

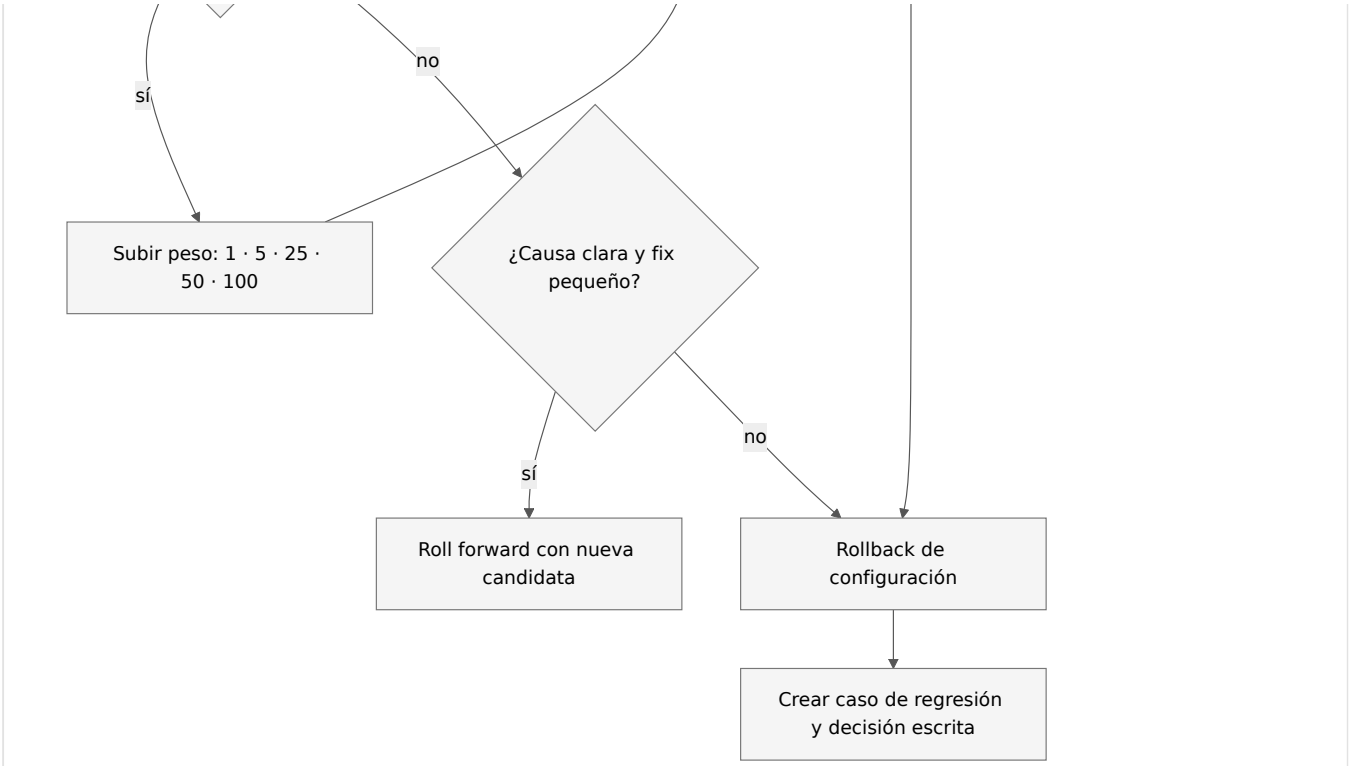
TROPIEZO	ANTÍDOTO
Confundir deployment y release.	Preguntar: ¿la versión está disponible o expuesta?
Usar `random()` por request.	Asignar por hash estable de cohorte.
Hacer shadow con tools de escritura.	Dry-run, solo lectura o bloqueo explícito.
Mirar solo errores técnicos.	Añadir contrato, coste, calidad, fallback y trazas.
No probar rollback.	Ensayar vuelta antes de subir tráfico.

La frase que me habría ahorrado muchos sustos: **un rollout no es una escalera automática; es una serie de decisiones con evidencia**.

Cómo encaja todo

Facsímil 6


Cap. 04 observabilidad
Cap. 05 routing
Cap. 06 EvalOps
autoriza candidata
Cambio aprobado por EvalOps
decide ruta
Deployment: versión disponible
Feature flag o política de release
mide
Dark launch
Shadow run sin efectos persistentes
¿Shadow deja señales aceptables?
sí
Canary por cohorte estable
Métricas por release_id y variant
no
¿Gate de paso pasa?



Relación con otros capítulos

Este capítulo no sustituye a EvalOps. Lo continúa.

CAPÍTULO	QUÉ APORTA AQUÍ
F6 · Capítulo 01	Manifest, versión, release gate y rollback como idea de sistema operable.
F6 · Capítulo 04	SLIs, SLOs, métricas, trazas y alertas para decidir.
F6 · Capítulo 05	Router, rutas, presupuestos, shadow routing y canary de políticas.
F6 · Capítulo 06	Datasets, gates y scorecard antes de exponer tráfico.
F5 · Capítulo 08	Revisión humana y permisos cuando una acción requiere aprobación.

La cadena completa queda así: EvalOps decide que la candidata merece exposición; progressive delivery decide cómo se expone; observabilidad decide si sigue avanzando; rollback conserva una salida de vuelta.

Para entenderlo

Tres escenas sencillas:

SITUACIÓN	MALA DECISIÓN	DECISIÓN PROGRESIVA
Nuevo prompt más claro	100% de tráfico el lunes.	Shadow, 1%, revisar contrato y subir si p95/coste aguantan.
Nuevo índice RAG	Sustituir índice antiguo.	Candidate usa índice nuevo en shadow y compara fuentes recuperadas.
Nuevo modelo barato	Cambiar proveedor por defecto.	Canary por tareas de baja criticidad, midiendo calidad y fallbacks.

No es miedo al cambio. Es respeto por la operación.

Vocabulario aprendido

TÉRMINO	DEFINICIÓN OPERATIVA	ERROR TÍPICO
Deployment	La versión está instalada o disponible.	Creer que ya está expuesta al usuario.
Release	La versión está recibiendo tráfico o afectando decisiones.	Publicarla al 100% sin gates intermedios.
Dark launch	La versión se despliega sin uso visible.	Confundir disponibilidad con validación.
Shadow run	La candidata procesa entradas reales sin producir efectos persistentes.	Dejar que duplique emails, tickets o escrituras.
Canary	Exposición pequeña y medible a una cohorte estable.	Usar tráfico aleatorio por request y mezclar señales.
Cohorte estable	Grupo asignado por hash de tenant, usuario, conversación o tarea.	Cambiar variante en cada llamada y romper comparaciones.
Kill switch	Control para apagar una capacidad sin redesplegar.	Tener que tocar código durante una incidencia.
Rollback	Vuelta a una versión conocida y verificada.	Pensar que solo es revertir código, olvidando prompt, índice o ruta.
Roll forward	Nueva candidata que corrige el problema sin volver atrás.	Usarlo sin evidencia suficiente porque “parece pequeño”.
Blast radius	Porción de usuarios, tenants o tareas expuesta al cambio.	Medir solo porcentaje global y no criticidad.

Antes de pasar página

Comprueba que puedes responder:

1. ¿Cuál es la diferencia entre deployment y release?
2. ¿Por qué un feature flag ayuda a publicar cambios de IA?
3. ¿Qué debe evitar un shadow run para no duplicar efectos persistentes?
4. ¿Por qué conviene asignar canary por cohorte estable?
5. ¿Qué señales mirarías en un canary de RAG?

6. ¿Qué capas puede tocar un rollback en IA además del código?
7. ¿Qué archivo entregarías para demostrar que el rollback está preparado?

En resumen

IDEA	PARA LLEVARTE
Deployment y release no son lo mismo.	Puedes tener una versión desplegada pero no expuesta.
Shadow reduce incertidumbre.	Compara con entradas reales sin afectar la salida visible.
Canary debe tener gates.	Cada porcentaje necesita métricas, duración, dueño y criterio de avance.
Rollback es diseño.	Volver por flag, prompt, modelo, índice o ruta debe estar probado antes.

Para saber más

- Argo Project. *Canary Deployment Strategy*. https://argoproj.github.io/argo-rollouts/features/canary/
- Google Cloud. *Use a canary deployment strategy*. https://cloud.google.com/deploy/docs/deployment-strategies/canary
- Kubernetes. *Deployments*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
- LaunchDarkly. *Reducing Technical Debt from Feature Flags*. https://launchdarkly.com/docs/guides/flags/technical-debt
- LaunchDarkly. *Releasing features with LaunchDarkly*. https://launchdarkly.com/docs/home/releases/releasing
- OpenTelemetry. *Semantic Conventions for Generative AI Systems*. https://opentelemetry.io/docs/specs/semconv/gen-ai/
- OpenTelemetry. *Traces*. https://opentelemetry.io/docs/concepts/signals/traces/
- OpenFeature. *Introduction*. https://openfeature.dev/docs/reference/intro/
- Prometheus. *Querying Basics*. https://prometheus.io/docs/prometheus/latest/querying/basics/
- Wilkinson, J. *Alerting on SLOs*. https://sre.google/workbook/alerting-on-slos/

Notas

1. OpenFeature. (2026). *Introduction*. https://openfeature.dev/docs/reference/intro/. Consultado el 28 de mayo de 2026. ↩

2. Kubernetes. (2026). *Deployments*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Consultado el 28 de mayo de 2026. ↩

3. Argo Project. (2026). *Canary Deployment Strategy*. https://argoproj.github.io/argo-rollouts/features/canary/. Consultado el 28 de mayo de 2026. ↩

4. LaunchDarkly. (2026). *Releasing features with LaunchDarkly*. https://launchdarkly.com/docs/home/releases/releasing. Consultado el 28 de mayo de 2026. ↩

5. Google Cloud. (2026). *Use a canary deployment strategy*. https://cloud.google.com/deploy/docs/deployment-strategies/canary. Consultado el 28 de mayo de 2026. ↩

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

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

8. 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. ↩

9. Beyer, B., Jones, C., Petoff, J. y Murphy, N. R. (2016). *Handling Overload*. En *Site Reliability Engineering*. https://sre.google/sre-book/handling-overload/. Consultado el 27 de mayo de 2026. ↩

10. Prometheus. (2026). *Querying Basics*. https://prometheus.io/docs/prometheus/latest/querying/basics/. Consultado el 28 de mayo de 2026. ↩

11. Wald, A. (1945). Sequential Tests of Statistical Hypotheses. *The Annals of Mathematical Statistics*, 16(2), 117-186. https://doi.org/10.1214/aoms/1177731118. Introduce el test secuencial de la razón de probabilidad (SPRT). ↩

12. El contraste de dos proporciones mediante el estadístico z es un resultado estándar de inferencia; véase Agresti, A. (2013). *Categorical Data Analysis* (3.ª ed.). Wiley. ↩

13. Beyer, B., Murphy, N. R., Rensin, D. K., Kawahara, K., & Thorne, S. (Eds.). (2018). *The Site Reliability Workbook*. O'Reilly. Define la alerta por tasa de consumo del presupuesto de error con ventanas múltiples. ↩

4. Shewhart, W. A. (1931). *Economic Control of Quality of Manufactured Product*. Van Nostrand. Origen del gráfico de control y de los límites a tres sigma. ↩

5. LaunchDarkly. (2026). *Reducing Technical Debt from Feature Flags*. https://launchdarkly.com/docs/guides/flags/technical-debt. Consultado el 28 de mayo de 2026. ↩