Multi-Cloud Engineering Program

Parte 08 — Entrega continua y platform engineering

12 clases · 52 horas

Cuaderno de una sola parte: su introducción, sus clases completas y sus evaluaciones.

Extracto del manual integral, generado desde las mismas fuentes versionadas.

Alcance de este cuaderno

Componente	Cobertura
Parte	08 de 23
Clases	097–108
Evaluaciones	12
Horas estimadas	52

Índice

Parte 08 — Entrega continua y platform engineering

← Parte 07 · Índice completo · Parte 09 →

Descargar: Esta parte en PDF · Manual integral

Nivel: avanzado · Clases: 12 · Duración sugerida: 6–8 semanas

Diseñar un flujo de entrega rápido, seguro y observable para equipos de producto.

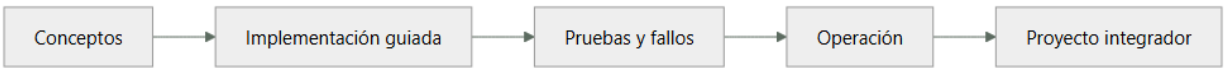
Resultados de la parte

Al completar esta parte podrás explicar el dominio con lenguaje independiente del proveedor, implementar sus mecanismos esenciales, diagnosticar fallos y defender decisiones mediante evidencia, costo, seguridad y objetivos de confiabilidad.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Secuencia



Clases

ID	Clase	Laboratorio	Horas
097	Integración continua, trunk-based development y feedback	delivery	4
098	GitHub Actions: workflows, runners, permisos y caché	delivery	4
099	Artefactos inmutables, semver y promoción	supply-chain	4
100	Pruebas, calidad y puertas de cambio	testing	4
101	SAST, SCA, secretos, SBOM y firma en pipeline	security	4
102	Rolling, blue-green, canary y rollback	delivery	4
103	GitOps con Argo CD o Flux	gitops	4
104	Ambientes efímeros y promoción entre entornos	delivery	4
105	Feature flags y separación deploy-release	delivery	4
106	Platform engineering e Internal Developer Platform	platform	4
107	Developer experience, DORA y carga cognitiva	metrics	4
108	Proyecto: fábrica de software multi-cloud	capstone	8

Evaluación de la parte

- 35 % laboratorios y evidencia reproducible.
- 25 % retos verificables.
- 20 % decisiones y ADR.
- 10 % seguridad, confiabilidad y costo.

- 10 % proyecto integrador de la parte.

Se aprueba con 80 % de clases en nivel B o superior y el proyecto integrador reproducible.

Pauta bibliográfica

- Continuous Delivery — Humble y Farley.
- Accelerate — Forsgren, Humble y Kim.
- Team Topologies — Skelton y Pais.

097 — Integración continua, trunk-based development y feedback

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 4 Laboratorio: delivery · Estado: EXECUTABLECORE

Propósito

Fijar por qué la integración continua es una práctica de organización del trabajo y no una herramienta, y cuál es la magnitud que lo decide todo: cuánto tiempo vive el trabajo sin integrarse. De ahí salen las ramas largas, los conflictos caros y las revisiones que nadie puede evaluar. La clase establece además el número que gobierna esta parte —el tiempo de respuesta de la canalización— y la ley 16 aparece por sexta vez con un mecanismo nuevo: si la canalización tarda demasiado, la gente encuentra cómo saltársela.

Resultados de aprendizaje

Al finalizar podrás:

1. Distinguir integración continua de tener una canalización, con la comprobación que las separa.
2. Medir el tiempo de vida de las ramas y su relación con el coste de integrar.
3. Dividir un cambio grande en incrementos integrables sin dejar de entregar valor.
4. Dimensionar el tiempo de respuesta de la canalización y saber qué ocurre al superarlo.
5. Diagnosticar una prueba intermitente en vez de reintentarla.

Conceptos centrales

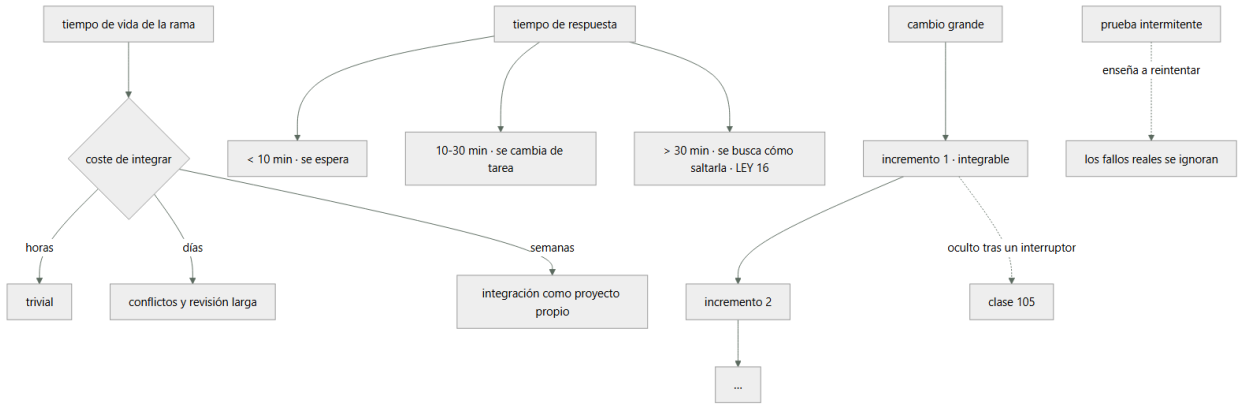
Concepto	Comprensión verificable
integración continua	Práctica en la que todo el mundo integra su trabajo en la línea principal al menos una vez al día. No es tener una canalización: es una decisión sobre cómo se organiza el trabajo.
tiempo de vida de una rama	Horas entre crearla y fusionarla. El coste de integrar crece más deprisa que ese tiempo, porque crecen a la vez el tamaño y la divergencia.
tiempo de respuesta	Minutos desde que se envía un cambio hasta que se sabe si está bien. Por encima de cierto umbral, la gente deja de esperarlo y trabaja sobre supuestos.
prueba intermitente	Prueba que falla sin motivo reproducible. Su daño real no es el fallo: es que enseña a reintentar, y con ello a ignorar los fallos verdaderos.
incremento integrable	Trozo de trabajo que se puede fusionar sin romper nada, aunque la funcionalidad completa no esté. Es lo que hace posible integrar a diario.
línea principal siempre desplegable	Invariante del método: en cualquier momento, lo que hay en la rama principal se puede desplegar. Sin ella, la integración diaria no aporta.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. La práctica no es la herramienta

Casi todas las organizaciones tienen una canalización y muy pocas hacen integración continua. La diferencia es una definición que conviene tomarse en serio:

```
tener una canalización      algo se ejecuta cuando envío código
integración continua        CADA PERSONA integra su trabajo en la línea
                            principal al menos una vez al día,
                            y la línea principal se puede desplegar siempre
```

La comprobación que las separa es una sola pregunta con respuesta medible:

```
$ git for-each-ref --format='%(refname:short) %(committerdate:relative)' refs/remotes/origin \
  | grep -v 'origin/main' | head -20
```

Si hay ramas de más de un día, no hay integración continua. Hay una canalización que se ejecuta sobre ramas que divergen.

Y el motivo por el que la magnitud importa es que el coste de integrar no crece linealmente:

```
una rama de 4 horas       un conflicto trivial, si acaso
una rama de 3 días        varios conflictos, y hay que recordar por qué
una rama de 3 semanas     la integración es un proyecto, con su riesgo propio
                          y la revisión es imposible de evaluar de verdad
```

Crecen dos cosas a la vez —el tamaño del cambio y la divergencia respecto de la línea principal— y su producto es lo que se paga.

Y hay un efecto secundario que se nota en la calidad de la revisión y que rara vez se dice:

```
un cambio de 50 líneas        se revisa de verdad, con comentarios concretos
un cambio de 2.000 líneas     se aprueba
```

La segunda no es una revisión: es un trámite. Y eso convierte todo el trabajo de las clases 091 y siguientes en la única barrera real, porque la humana ha dejado de funcionar.

Y la objeción habitual —«no puedo integrar hasta que la funcionalidad esté completa»— tiene una respuesta que es la mitad de esta clase: integrar no es publicar. Un incremento puede estar en la línea principal, desplegado en producción y apagado, que es la materia de la clase 105. Sin ese mecanismo, la integración diaria es imposible para cualquier cambio que dure más de un día, y con él es casi siempre posible.

Y las tres consecuencias de trabajar así, que son las que justifican el esfuerzo:

```
los conflictos son triviales o no existen
el origen de un fallo es evidente: pocos cambios entre versiones que funcionan
y la vuelta atrás es pequeña, porque el cambio es pequeño
```

2. Dividir sin dejar de entregar

La habilidad que hace posible todo lo anterior es dividir un cambio grande en incrementos que se puedan integrar. Hay cuatro técnicas y conviene tenerlas nombradas:

1. Ocultar tras un interruptor. El código nuevo se integra y no se ejecuta hasta que alguien lo activa. Es la técnica general y la clase 105 la desarrolla.

2. Expandir y contraer. Ya apareció en las clases 071, 079 y 088, y aquí es el mecanismo para cambiar una interfaz sin romper a nadie:

```
1. añadir lo nuevo, conservando lo viejo
2. migrar a los consumidores, uno a uno, integrando cada paso
3. retirar lo viejo cuando nadie lo use
```

Cuarta aparición de la misma técnica en el programa, y siempre con la misma condición: hay que llegar al paso 3. Un expandir sin contraer deja dos formas de hacer lo mismo para siempre.

3. Rama por abstracción. Para sustituir un componente grande sin una rama larga:

```
1. introducir una abstracción delante del componente actual
2. integrar; todo sigue funcionando igual
3. escribir la implementación nueva detrás de la abstracción, integrando a diario
4. cambiar cuál se usa, con un interruptor
5. retirar la vieja y, si sobra, la abstracción
```

Es más trabajo que una rama larga y el trabajo es visible y revisable en trozos, en vez de invisible durante tres semanas y luego imposible de revisar.

4. Entregar en oscuro. El camino nuevo se ejecuta en paralelo con el viejo, sin que su resultado se use, y se comparan. Es caro y es la única forma honesta de sustituir algo cuyo comportamiento no se puede especificar por completo.

Y una advertencia sobre los interruptores que la clase 105 desarrolla y conviene anticipar: cada uno es deuda con fecha. Un interruptor que lleva un año activo al 100 % no es un interruptor, es una rama muerta con un condicional.

Y sobre la revisión, dos prácticas que reducen el tiempo de vida de las ramas más que ninguna herramienta:

```
revisar en horas, no en días
  una revisión que tarda dos días convierte cualquier rama en una rama larga
  → un compromiso explícito de tiempo de respuesta, como el de la canalización

programación en pareja o en grupo
  la revisión ocurre mientras se escribe; la integración es inmediata
  → no es para todo el trabajo, y para lo complejo suele salir más barato
```

La primera es la que más impacto tiene y la que menos se mide. Un equipo con una canalización de cinco minutos y revisiones de dos días no tiene un problema de herramientas.

3. El tiempo de respuesta, y la ley 16

El número que gobierna esta parte es cuánto tarda alguien en saber si su cambio está bien. Y sus umbrales están medidos:

```
< 10 minutos    se espera mirando; el trabajo continúa sobre información cierta
10-30 minutos   se cambia de tarea; volver cuesta el coste de recuperar contexto
> 30 minutos    se deja de esperar; se trabaja sobre supuestos
> 1 hora        se busca cómo saltarla
```

La última línea es la ley 16 de la clase 096 con un mecanismo nuevo, y es su sexta aparición en el programa: un control que hace el trabajo más lento se rodea. Aquí se rodea de formas concretas y reconocibles:

```
fusionar sin esperar al resultado
saltarse comprobaciones con una etiqueta o una opción
acumular varios cambios en una sola ejecución
y, la peor, desactivar pruebas que "tardan mucho"
```

De ahí que reducir el tiempo de respuesta no sea una mejora de comodidad sino la condición para que las comprobaciones sobrevivan.

Las palancas, por orden de efecto:

```
1. dividir en etapas por coste (clase 091)
   lo rápido primero; lo caro solo si lo rápido pasa
2. paralelizar lo independiente
   y no paralelizar lo que comparte estado, que produce intermitencia
3. caché de dependencias y de construcción (clase 062)
   la diferencia entre once minutos y noventa segundos
4. ejecutar solo lo afectado
   con un mapa de dependencias real, no por corazonada
```

```
5. mover lo caro fuera del camino crítico
   pruebas largas y análisis profundos, programados o antes de fusionar
```

Y la cuarta merece una advertencia: seleccionar qué pruebas ejecutar es una optimización con riesgo. Un mapa de dependencias incompleto deja pasar una rotura, y ese fallo aparece más tarde y cuesta más. Solo compensa con un mapa derivado del código, no mantenido a mano.

Y una comprobación que conviene tener, porque el tiempo de respuesta se degrada solo:

```
mediana y percentil 95 del tiempo de la canalización, por semana
  → una tendencia al alza es deuda que se paga en integración diaria
```

Y dos cifras más que la parte 08 va a usar y conviene empezar a medir ya:

```
tiempo de vida de las ramas: mediana y percentil 90
proporción de ejecuciones que fallan por causas no reproducibles
```

4. La prueba intermitente enseña a ignorar

Una prueba que falla sin motivo reproducible parece un problema menor y es uno de los peores, porque su daño no es el fallo:

```
el daño real es que ENSEÑA A REINTENTAR
y quien reintenta por costumbre ignora también los fallos verdaderos
```

La consecuencia se mide y suele sorprender:

```
proporción de ejecuciones fallidas que se reintentan sin investigar
  por encima del 10 %, la canalización ha dejado de ser una señal
```

Es la ley 15 de la clase 096 aplicada a la canalización: una señal que se equivoca a menudo deja de ser una señal.

Las causas, en orden de frecuencia, y qué hacer con cada una:

```
dependencia de tiempo         esperas fijas en vez de esperar a una condición
                              → sustituir cada espera fija por una condición
dependencia de orden          las pruebas comparten estado y el orden varía
                              → aislar el estado; ejecutar en orden aleatorio
                                a propósito para detectarlo
recursos compartidos          un puerto, un fichero, una base de datos común
                              → recursos únicos por ejecución
dependencia externa           una API de terceros dentro de una prueba unitaria
                              → sustituirla; si hace falta de verdad,
                                es otra categoría de prueba
concurrencia real             una condición de carrera en el código
                              → es un HALLAZGO, no un problema de la prueba
```

La última merece subrayarse: una prueba intermitente por concurrencia está encontrando un defecto real que aparecerá en producción. Silenciarla es silenciar el hallazgo.

Y el procedimiento que evita que se acumulen, que hay que decidir antes de tener veinte:

```
1. detectar: ejecutar la suite varias veces sobre el mismo commit
   las que fallan alguna vez son intermitentes
2. cuarentena inmediata: sacarla del camino crítico, con su fallo registrado
3. plazo: una semana para arreglarla o borrarla
4. y un tope: si hay más de N en cuarentena, se para y se arreglan
```

El paso 3 es incómodo y necesario. Una prueba en cuarentena indefinida es una prueba que no existe, con el coste de mantenerla.

Y una comprobación barata que detecta la mayoría antes de que molesten:

```
# ejecutar la suite cinco veces sobre el mismo commit
$ for i in 1 2 3 4 5; do npm test -- --seed=$RANDOM 2>&1 | tail -1; done
```

La aleatorización de la semilla es lo que destapa las dependencias de orden, que son la segunda causa y la más difícil de encontrar cuando ya está instalada.

5. Lo que la canalización debe garantizar

Con lo anterior, la canalización de integración tiene un contrato corto y verificable:

```
en cada cambio, antes de fusionar
  ■ construir el artefacto UNA vez (clase 062)
  ■ comprobaciones rápidas: formato, análisis estático, pruebas unitarias
```

```
  ■ pruebas de integración con dependencias reales acotadas
  ■ análisis de seguridad y de dependencias (clase 101)
  ■ y el resultado en menos de diez minutos

al fusionar
  ■ publicar el artefacto por huella, firmado y con procedencia (clases 061, 067)
  ■ desplegar a un entorno donde se pueda comprobar de verdad
```

Y tres propiedades que hacen que ese contrato se sostenga:

La línea principal siempre desplegable. Sin ella, la integración diaria no aporta: se integra en algo que no funciona. Y el mecanismo que la protege es una cola de fusión que verifica el resultado combinado antes de fusionar:

```
dos cambios que pasan por separado pueden romper al combinarse
  → sin cola de fusión, eso rompe la línea principal
  → con ella, se detecta antes de fusionar
```

Es un mecanismo que solo hace falta con cierto volumen, y a partir de ahí es imprescindible.

La construcción es reproducible. La regla de la clase 062: se construye una vez y se promueve. Una canalización que construye de nuevo en cada entorno rompe la garantía de que lo probado es lo desplegado.

La canalización se define en el repositorio. Configurada en la interfaz de una herramienta, es un estado que nadie revisa y que se desvía —lo mismo que la infraestructura de la parte 07, con el mismo remedio.

Y el cierre que enlaza con la clase siguiente: todo lo anterior necesita ejecutarse en algún sitio, con alguna identidad y con acceso a algún artefacto. Ese ejecutor es el actor más privilegiado del sistema, y su tratamiento es la materia de la clase 098 y una de las dos predicciones de la hipótesis que abrió esta parte.

Ejemplo trabajado

CloudShop tiene una canalización desde hace tres años y ninguna de las propiedades de esta clase. La medición previa da cuatro números que explican por qué los despliegues son difíciles.

La medición.

```
tiempo de vida de las ramas    mediana 6 días · percentil 90: 23 días
tamaño del cambio             mediana 340 líneas · percentil 90: 2.900
tiempo de respuesta           mediana 34 min · percentil 95: 71 min
ejecuciones reintentadas sin investigar   31 %
revisiones que tardan más de un día       58 %
```

Y las cuatro consecuencias, todas visibles en el historial:

```
conflictos de integración al mes                   41
reversiones tras fusionar                           9
cambios fusionados sin esperar el resultado        14
pruebas desactivadas en los últimos 12 meses       23
```

La última cifra es la ley 16: con un tiempo de respuesta de 71 minutos en el percentil 95, desactivar pruebas lentas era la forma racional de trabajar.

Corrección 1 — el tiempo de respuesta.

El desglose de los 34 minutos:

```
instalar dependencias                   9 min 20 s   sin caché
construir                               6 min 10 s   sin caché
pruebas unitarias                       4 min 40 s
pruebas de integración                 11 min 30 s   secuenciales
análisis de seguridad                   2 min 20 s
```

```
caché de dependencias y construcción        ninguna          compartida (062)
pruebas de integración                    secuenciales     4 en paralelo
análisis profundo                      en cada cambio     programado y antes
                                                          de fusionar
tiempo de respuesta (mediana)              34 min             6 min 40 s
percentil 95                               71 min           9 min 10 s
```

Y el efecto sobre el comportamiento, medido dos meses después:

```
cambios fusionados sin esperar el resultado    14 → 0
pruebas desactivadas                            0 nuevas
```

Corrección 2 — las pruebas intermitentes.

Ejecutar la suite cinco veces sobre el mismo commit:

```
pruebas que fallaron alguna vez                19 de 1.240
causa
  esperas fijas                                 8
  dependencia de orden                          6
  puerto o fichero compartido                   3
  condición de carrera real en el código        2   ← hallazgo
```

Las dos últimas eran defectos reales: una condición de carrera en la actualización de un contador de inventario que producía, en producción, discrepancias de stock que se atribuían a errores de conteo manual.

```
pruebas intermitentes                          19              0
defectos reales encontrados por ellas           —              2
reintentos sin investigar                     31 %            3 %
cuarentena con plazo de una semana          no había        política activa
aleatorización del orden                    no había        en cada ejecución
```

Corrección 3 — el tiempo de vida de las ramas.

Dos cambios, y el segundo fue el que de verdad movió la cifra:

```
1. compromiso de revisión en menos de 4 horas laborables
2. formación en las cuatro técnicas de división, con un caso real:
   la sustitución del motor de precios, planificada como rama de 6 semanas,
   se hizo con rama por abstracción en 19 incrementos integrados a diario
```

```
mediana del tiempo de vida                   6 días          7 horas
percentil 90                                23 días         1,5 días
mediana del tamaño del cambio               340 líneas      95 líneas
revisiones de más de un día                   58 %            6 %
conflictos de integración al mes               41              3
```

Y la sustitución del motor de precios merece detalle porque es el argumento contra las ramas largas:

```
duración                                      6 semanas         7 semanas
revisiones significativas                     1, de 4.800 líneas  19, de ~250
defectos encontrados durante el desarrollo    —                  6
defectos encontrados tras publicar            —                  1
riesgo de la integración final               alto               ninguno
posibilidad de parar a mitad                  ninguna            en cualquier punto
```

Una semana más de calendario, y seis defectos encontrados mientras se escribía en vez de al final. La última fila fue la que convenció: el trabajo se pudo pausar dos veces por urgencias sin dejar nada a medias.

Corrección 4 — la línea principal desplegable.

```
veces que la línea principal estuvo rota en 12 meses     34
tiempo medio de rotura                                   2 h 40 min
causa más frecuente        dos cambios que pasaban por separado y no juntos
```

```
cola de fusión                              no había          activa
roturas de la línea principal (6 meses)        17                1
la única                                        —          una dependencia
                                                           externa retirada
```

Resumen:

```
tiempo de respuesta (mediana)                 34 min          6 min 40 s
mediana del tiempo de vida de una rama        6 días          7 horas
mediana del tamaño de un cambio             340 líneas      95 líneas
pruebas intermitentes                          19               0
reintentos sin investigar                     31 %             3 %
conflictos de integración al mes               41               3
roturas de la línea principal (6 meses)        17               1
pruebas desactivadas en el periodo             23               0
```

La lección que esta clase traslada al resto de la parte 08: las veintitrés pruebas desactivadas no eran negligencia sino la respuesta racional a una canalización de 71 minutos, que es la ley 16 en su sexta aparición. Y el cambio con más efecto no fue técnico: pasar de ramas de seis días a siete horas redujo los conflictos de 41 a 3 al mes y convirtió las revisiones en algo que se puede hacer de verdad. Una revisión de 2.900 líneas no es una revisión: es una aprobación.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/097-integracion-continua-trunk-based-development-y-feedback/lab.py
```

El laboratorio selecciona el motor de práctica delivery y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa flujo-ci en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es un pipeline con gates, promoción y rollback. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye flujo-ci para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
Se desactivan pruebas porque tardan demasiado	Es la ley 16: la canalización hace el trabajo más lento y se rodea	Reduce el tiempo de respuesta por debajo de diez minutos antes de exigir que nadie desactive nada.
Los conflictos de integración son frecuentes y caros	Las ramas viven días o semanas, así que crecen a la vez el tamaño y la divergencia	Integra a diario dividiendo el trabajo, y compromete un tiempo de revisión en horas.
Las revisiones se aprueban sin comentarios de fondo	Un cambio de miles de líneas no se puede evaluar	Divide en incrementos integrables; un cambio de menos de cien líneas se revisa de verdad.
Los fallos de la canalización se reintentan por costumbre	Hay pruebas intermitentes y han enseñado a no confiar en la señal	Detéctalas ejecutando la suite varias veces, ponlas en cuarentena con plazo y arréglalas o bórralas.
La línea principal se rompe con cambios que pasaban por separado	No se verifica el resultado combinado antes de fusionar	Cola de fusión que comprueba la combinación; a partir de cierto volumen es imprescindible.
No se puede integrar hasta que la funcionalidad esté completa	Se confunde integrar con publicar	Integra el incremento apagado tras un interruptor, o usa rama por abstracción para sustituciones grandes.

■ Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

■ Preguntas de comprobación

1. ¿Qué comprobación distingue tener una canalización de hacer integración continua?
2. ¿Por qué el coste de integrar no crece linealmente con el tiempo de vida de una rama?
3. Enumera las cuatro técnicas de división y para qué sirve cada una.
4. ¿Qué umbrales tiene el tiempo de respuesta y qué ocurre al superar el último?
5. ¿Por qué el daño de una prueba intermitente no es el fallo, y qué procedimiento evita que se acumulen?

Referencias

- Jez Humble, David Farley (2010). Continuous Delivery, cap. 3 — integración continua y línea principal desplegable. <https://continuousdelivery.com/>
- Martin Fowler (2023). Patterns for managing source code branches — tiempo de vida de ramas y técnicas de división. <https://martinfowler.com/articles/branching-patterns.html>
- Paul Hammant (2020). Branch by abstraction — sustituir un componente sin rama larga. <https://www.branchbyabstraction.com/>
- Google (2020). Flaky tests at scale — causas, cuarentena y coste de la intermitencia. <https://testing.googleblog.com/2020/12/test-flakiness-one-of-main-challenges.html>
- Nicole Forsgren et al. (2018). Accelerate, cap. 4 — tiempo de entrega, lotes pequeños y su efecto medido. <https://itrevolution.com/product/accelerate/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 08 en PDF · Manual integral

Anterior	Índice	Siguiente
← 096 · Proyecto: infraestructura multiambiente promovible	Parte 08 · Programa	098 · GitHub Actions: workflows, runners, permisos y caché →

Evaluación — 097 Integración continua, trunk-based development y feedback

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega flujo-ci con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.

098 — GitHub Actions: workflows, runners, permisos y caché

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 4 Laboratorio: delivery · Estado: EXECUTABLECORE

Propósito

Operar la canalización sabiendo lo que es: el actor más privilegiado del sistema. Despliega en producción, lee secretos, publica artefactos y ejecuta código que llega de fuera. Esa combinación la convierte en el objetivo más valioso, que es la primera predicción de la hipótesis que abrió esta parte. La clase cubre el modelo de ejecución, los cuatro caminos por los que un cambio ajeno consigue ejecutar código con privilegios, y la disciplina de versionado que las clases 061 y 088 ya exigieron —aquí con una razón nueva y más aguda.

Resultados de aprendizaje

Al finalizar podrás:

1. Describir el modelo de ejecución y dónde entra código que no controla la organización.
2. Reconocer los cuatro caminos de escalada característicos y cerrarlos.
3. Acotar los permisos del testigo y el alcance de los secretos por entorno.
4. Fijar acciones de terceros por revisión exacta y justificar por qué no basta la etiqueta.
5. Configurar ejecutores propios y caché sin abrir un camino hacia la red interna.

Conceptos centrales

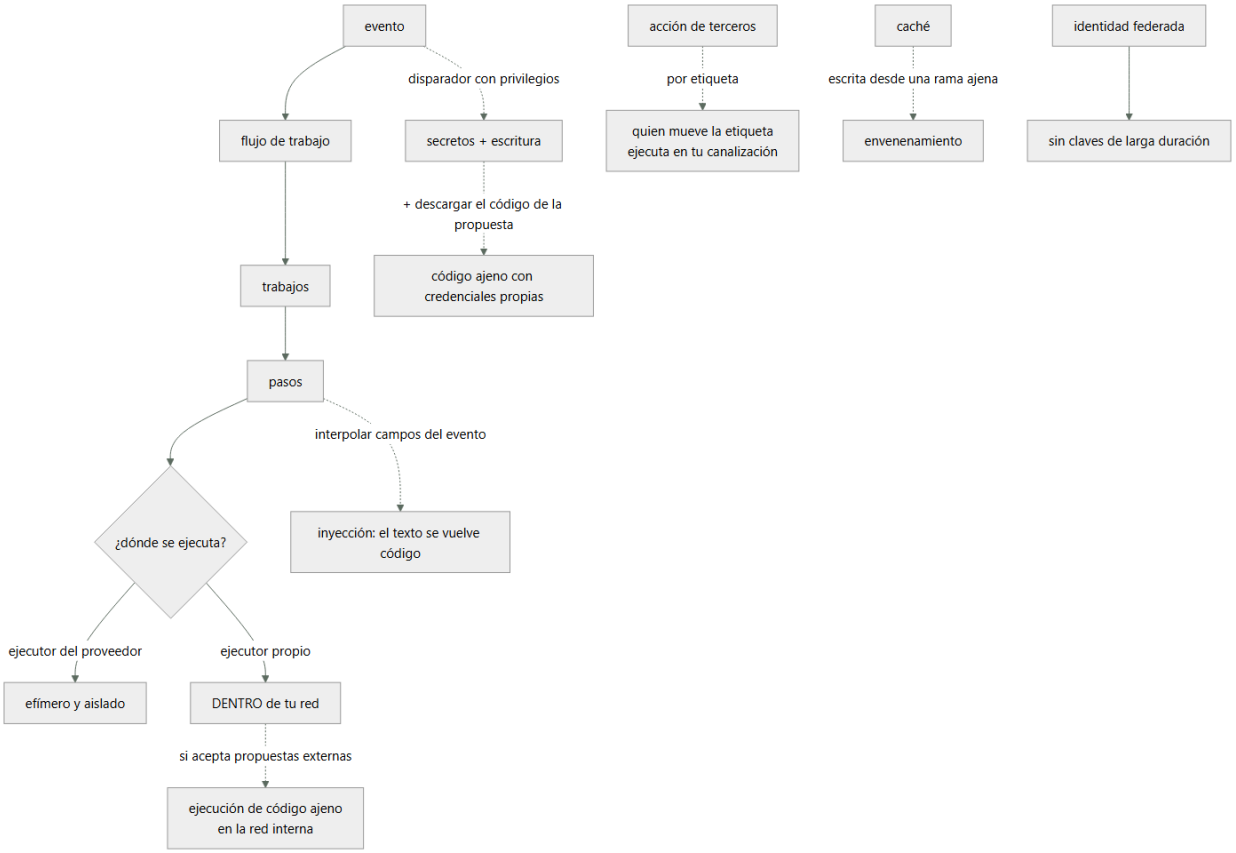
Concepto	Comprensión verificable
ejecutor	Máquina que ejecuta los pasos. Si es propia y está en la red de la organización, cualquier código que ejecute está dentro.
disparador con privilegios	Evento que ejecuta con permisos de escritura y acceso a los secretos del repositorio base. Combinado con descargar el código de la propuesta, ejecuta código ajeno con credenciales propias.
inyección por expresión	Interpolar un campo del evento dentro de una orden. El texto lo escribe quien abre la propuesta, así que se convierte en código.
permisos del testigo	Alcance de la credencial automática del repositorio. Por defecto puede ser amplio; debe declararse mínimo y por trabajo.
fijación por revisión	Referenciar una acción por su identificador exacto. Una etiqueta se puede mover, y quien la mueve ejecuta código en tu canalización.
envenenamiento de caché	Escribir en una caché desde una rama no confiable y que otra ejecución la lea. Convierte una optimización en un camino de ejecución de código.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. El modelo, y dónde entra código ajeno

La estructura es sencilla y lo importante es dónde se cruza con código que la organización no controla:

```
evento          alguien envía código, abre una propuesta, publica una versión,
                o llega la hora
  flujo de trabajo
    trabajo     se ejecuta en un ejecutor; los trabajos son paralelos por defecto
      paso      una orden o una acción de terceros
```

Y los cuatro sitios por donde entra algo que no ha escrito la organización:

```
1. el código de una propuesta ajena     lo escribe quien la abre
2. los campos del evento                título, rama, descripción: texto libre
3. las acciones de terceros             código de otros, ejecutado con tus permisos
4. las dependencias que se instalan     y sus guiones de instalación
```

Los cuatro son legítimos y necesarios. El problema aparece cuando se cruzan con credenciales, y ahí está toda la clase.

Y el primer punto de decisión es el ejecutor:

```
del proveedor    máquina efímera, fuera de tu red, destruida al terminar
                 → el código ajeno se ejecuta en una máquina que no es tuya
propio           máquina tuya, normalmente en tu red
                 → el código ajeno se ejecuta DENTRO
```

La segunda opción existe por motivos reales —acceso a redes privadas, hardware concreto, coste— y trae una consecuencia que hay que asumir antes:

```
un ejecutor propio que atiende propuestas de un repositorio público
es ejecución de código arbitrario dentro de la red de la organización
```

Y con un agravante: por defecto, un ejecutor no efímero conserva el sistema de ficheros entre ejecuciones, así que un trabajo puede dejar preparado algo que el siguiente ejecute.

Las tres medidas, y hacen falta las tres:

```
efímeros        una máquina nueva por ejecución, destruida al terminar
aislados        en su propia red, con acceso solo a lo que necesitan
                y nunca a la red de producción
nunca para propuestas de fuera de la organización
```

La tercera es la que no se puede negociar. Y para el caso legítimo —contribuciones externas a un repositorio público— la respuesta es que esas ejecuciones vayan a ejecutores del proveedor y sin ningún secreto.

2. Los cuatro caminos de escalada

Camino 1: el disparador con privilegios más el código ajeno.

Hay un evento que existe para poder comentar o etiquetar propuestas externas: se ejecuta con permisos de escritura y con acceso a los secretos del repositorio base. Y es seguro mientras no ejecute el código de la propuesta.

```
# PELIGROSO
on: pull_request_target
jobs:
  build:
    steps:
      - uses: actions/checkout@v4
        with:
          ref: ${{ github.event.pull_request.head.sha }}   # ← código ajeno
      - run: npm ci && npm test                            # ← ejecutado con secretos
```

Cualquiera que abra una propuesta puede modificar los guiones del proyecto y ejecutarlos con las credenciales de la organización. Es el fallo más grave y más común de esta clase.

La regla es corta:

```
con el disparador privilegiado: NO se descarga el código de la propuesta
con el disparador normal: no hay secretos ni permisos de escritura
```

Y para lo que de verdad necesita ambas cosas —publicar un entorno de vista previa, comentar resultados—, el patrón correcto separa en dos flujos:

```
flujo 1 · disparador normal   construye y prueba, SIN secretos
                              deja el resultado como artefacto
flujo 2 · al terminar el 1    con privilegios, LEE el artefacto
                              y publica o comenta; nunca ejecuta el código
```

Camino 2: la inyección por expresión.

```
# PELIGROSO
- run: echo "Revisando ${{ github.event.pull_request.title }}"
```

Ese título lo escribe quien abre la propuesta, y se interpola antes de ejecutar la orden. Un título con la sintaxis adecuada se convierte en órdenes.

La corrección es no interpolar nunca dentro de una orden:

```
- env:
    TITULO: ${{ github.event.pull_request.title }}
  run: echo "Revisando $TITULO"
```

La variable de entorno se pasa como dato y no como texto que se sustituye. Y la lista de campos que hay que tratar como texto ajeno es más larga de lo que parece: título, cuerpo, nombre de rama, mensaje de commit, nombre y correo del autor, y el contenido de cualquier etiqueta.

Camino 3: las acciones de terceros.

Una acción es código que se ejecuta con los permisos del trabajo. Referenciarla por etiqueta significa que quien controle esa etiqueta puede cambiar lo que se ejecuta:

```
- uses: alguien/accion@v3                          # ← la etiqueta se mueve
- uses: alguien/accion@a1b2c3d4e5f6…               # ← revisión exacta
```

Es la disciplina de las clases 061, 062, 081 y 088 —fijar versiones— con una razón más aguda: aquí la versión no solo cambia el comportamiento, ejecuta código con acceso a los secretos del trabajo.

Y dos medidas complementarias:

```
lista de acciones permitidas a nivel de organización
un robot que proponga la actualización de las revisiones fijadas
  → sin él, fijar congela también las correcciones (clase 062)
```

Camino 4: el envenenamiento de caché.

Una caché escrita desde la ejecución de una propuesta puede ser leída por una ejecución posterior con más privilegios, según cómo estén organizadas las claves y las ramas. Y lo que hay en una caché se usa como si fuera propio: dependencias, binarios, artefactos intermedios.

```
regla   la caché de las ejecuciones de propuestas NO debe ser legible
        por las ejecuciones de la rama principal
        → separar por clave, y no restaurar cachés de ramas ajenas
        en trabajos con permisos
```

3. Permisos, secretos e identidad

El testigo del repositorio es una credencial que la plataforma inyecta en cada trabajo, y su alcance por defecto puede ser mucho mayor del necesario:

```
# en la raíz del flujo: mínimo por defecto
permissions:
  contents: read

jobs:
  publicar:
    permissions:
      contents: read
      packages: write
      id-token: write        # solo donde haga falta federar
```

Declararlo por trabajo, y no por flujo, es lo que evita que el trabajo de pruebas tenga permiso de escritura porque el de publicación lo necesita.

Los secretos se acotan por entorno, con aprobación cuando corresponde:

```
jobs:
  desplegar-produccion:
    environment: produccion       # con revisores obligatorios y espera
    steps:
      - run: ./desplegar.sh
```

Eso da tres cosas a la vez: los secretos de producción solo existen en ese trabajo, hay aprobación humana antes de ejecutarlo, y queda registrado quién aprobó.

Y la regla que este programa lleva ocho apariciones repitiendo, ahora en su forma definitiva:

```
la canalización NO tiene claves de la nube: tiene identidad
```

```
- uses: aws-actions/configure-aws-credentials@a1b2c3d4…
  with:
    role-to-assume: arn:aws:iam::418293047512:role/despliegue-tienda
    aws-region: eu-west-1
```

Y del lado de la nube, la condición de confianza acotada al repositorio y a la rama o al entorno, que es el campo que ha fallado en tres de las cuatro veces que este programa lo ha comprobado:

```
sub = repo:cloudshop/tienda:environment:produccion
```

Sin acotar, cualquier rama de ese repositorio —o de otro— obtiene credenciales de producción.

Y dos separaciones más que la clase 059 estableció y aquí se concretan:

```
identidad de LECTURA     para planificar en propuestas
identidad de ESCRITURA   solo desde la rama principal, y con entorno protegido
```

Y un mecanismo que evita dos despliegues simultáneos, que es el bloqueo de la clase 087 en su versión de canalización:

```
concurrency:
  group: desplegar-${{ github.ref }}
  cancel-in-progress: false
```

cancel-in-progress: false importa en un despliegue: cancelar uno a mitad deja el sistema en un estado intermedio. En cambio, en una construcción de propuesta, cancelar la anterior sí ahorra tiempo y no rompe nada.

4. Caché, reutilización y tiempo de respuesta

La clase 097 fijó el objetivo: por debajo de diez minutos. La caché es la palanca principal y tiene tres reglas.

Clave exacta y claves de respaldo:

```
- uses: actions/cache@a1b2c3d4…
  with:
    path: ~/.npm
    key: npm-${{ runner.os }}-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      npm-${{ runner.os }}-
```

La clave incluye la huella del fichero de dependencias, así que cambia solo cuando cambian. Y la clave de respaldo permite reutilizar la caché anterior cuando la exacta no existe, que es lo que evita empezar de cero tras cada actualización de dependencias.

Lo que se cachea y lo que no:

```
sí    dependencias descargadas, caché de compilación, capas de imagen (062)
no    artefactos de salida — esos se publican, no se cachean
no    nada que dependa de un secreto o que lo contenga
```

El ciclo de vida: las cachés caducan y tienen un tope de tamaño total, con expulsión de las menos usadas. Una caché que supera el tope expulsa a las demás, así que cachear un directorio enorme puede empeorar el tiempo de todos los flujos del repositorio.

Y la reutilización de flujos aplica aquí la disciplina de la clase 088:

```
# .github/workflows/servicio.yml — en cada servicio
jobs:
  ci:
    uses: cloudshop/plataforma/.github/workflows/servicio-ci.yml@v3.2.0
    with:
      lenguaje: go
      criticidad: 2
    secrets: inherit
```

Y con las mismas reglas que un módulo: interfaz pequeña, versionado, notas de cambio y política de soporte. Un flujo reutilizable con veinte entradas es la clase 088 otra vez.

Y dos precisiones sobre la reutilización:

```
heredar secretos es cómodo y los comparte ENTEROS
  → mejor pasar solo los que hagan falta
un flujo reutilizable de la plataforma se ejecuta con los permisos
  del repositorio que lo llama
  → declararlos mínimos dentro del propio flujo
```

Y el tiempo de respuesta se gobierna con lo de la clase 097 más dos mecanismos propios:

```
trabajos en paralelo con dependencias declaradas
  → solo espera lo que de verdad depende
matriz para lo que se repite
  → y con la advertencia: una matriz de 30 combinaciones
    consume 30 ejecutores y puede ser más lenta por la cola
```

Y una comprobación que conviene tener, porque el tiempo se degrada solo:

```
$ gh run list --workflow ci.yml --limit 100 --json durationMs \
  | jq '[.[].durationMs] | sort | .[length/2|floor] / 60000 | round'
```

5. La auditoría de la canalización

Con lo anterior, hay una lista de comprobaciones que se pueden ejecutar sobre los flujos de una organización:

```
# 1 · disparadores privilegiados que descargan código ajeno
$ grep -rlZ 'pull_request_target' .github/workflows/ \
  | xargs -0 grep -l 'head.sha\|head.ref' \
  && echo "ESCALADA: revisar de inmediato"

# 2 · interpolación de campos del evento dentro de órdenes
$ grep -rnE 'run:.*\$\{\{\s*github\.event\.' .github/workflows/

# 3 · acciones de terceros sin fijar por revisión
$ grep -rhoE 'uses:\s*[^ ]+@[^ ]+' .github/workflows/ \
  | grep -vE '@[0-9a-f]{40}' | grep -v '^uses: \./' | sort -u

# 4 · permisos no declarados o amplios
```

```
$ for f in .github/workflows/*.yml; do
    grep -q '^permissions:' "$f" || echo "sin permisos declarados: $f"
    grep -q 'permissions:.*write-all' "$f" && echo "permisos amplios: $f"
  done

# 5 · ejecutores propios en flujos que atienden propuestas
$ grep -rlZ 'self-hosted' .github/workflows/ \
  | xargs -0 grep -l 'pull_request'

# 6 · secretos de larga duración en vez de identidad federada
$ gh secret list --json name -q '.[].name' | grep -Ei 'aws_secret|azure_client_secret|gcp_sa_key'
```

Seis comprobaciones que caben en un guion y que corresponden a los cuatro caminos más los permisos y la identidad.

Y la lista de comprobación de la clase:

- ningún disparador privilegiado que descargue el código de la propuesta
- ningún campo del evento interpolado dentro de una orden
- acciones de terceros fijadas por revisión, con robot que proponga actualizar
- lista de acciones permitidas a nivel de organización
- permisos declarados por trabajo, mínimos
- secretos acotados por entorno, con aprobación en producción
- identidad federada acotada a repositorio y entorno; cero claves de nube
- ejecutores propios efímeros, aislados y nunca para propuestas externas
- caché separada por confianza; nada sensible cacheado
- concurrencia declarada, sin cancelar despliegues a mitad
- flujos reutilizables versionados, con interfaz pequeña
- tiempo de respuesta vigilado, con la mediana por debajo de diez minutos

Doce puntos, de los cuales siete son de seguridad. Esa proporción es la tesis de la clase y responde a la hipótesis que abrió la parte: la canalización es el actor más privilegiado del sistema, y se audita como tal.

Y una consecuencia que conviene enunciar porque cambia la prioridad de un equipo: comprometer la canalización equivale a comprometer todo lo que despliega. Los controles de las partes 05, 06 y 07 —firma de imágenes, política sobre el plan, admisión— siguen valiendo, y el actor que los ejecuta puede saltárselos si es él quien está comprometido. Por eso la separación entre la identidad que planifica y la que aplica, y la aprobación humana en producción, no son formalismos: son lo único que queda cuando el ejecutor deja de ser de fiar.

Ejemplo trabajado

CloudShop audita sus flujos de canalización por primera vez. Los cinco hallazgos son de los cuatro caminos de escalada, y el primero permitía a cualquier persona de internet desplegar en producción.

Hallazgo 1 — cualquiera podía obtener las credenciales de producción.

```
$ grep -rlZ 'pull_request_target' .github/workflows/ | xargs -0 grep -l 'head.sha'
.github/workflows/vista-previa.yml
```

El flujo construía un entorno de vista previa para cada propuesta. Se ejecutaba con el disparador privilegiado —para poder comentar el enlace— y descargaba el código de la propuesta.

```
on: pull_request_target
permissions: write-all
jobs:
  vista-previa:
    steps:
      - uses: actions/checkout@v4
        with: { ref: "${{ github.event.pull_request.head.sha }}" }
      - run: make vista-previa           # ← ejecuta el Makefile de la propuesta
```

El repositorio es público. Cualquiera podía abrir una propuesta que modificara el Makefile y ejecutar código con los secretos de la organización, incluida la identidad de despliegue.

```
estructura                          un flujo privilegiado   dos flujos separados
                                    que ejecuta código ajeno
flujo 1                                    —                construye sin secretos,
                                                            deja artefacto
flujo 2                                    —                con privilegios, LEE el
                                                            artefacto, no ejecuta código
permisos                            write-all               mínimos por trabajo
tiempo expuesto                    14 meses                       —
credenciales rotadas                   —                      todas
```

Se revisó el registro de auditoría de los catorce meses buscando ejecuciones anómalas. No se encontró ninguna, y las credenciales se rotaron igualmente: la exposición se trata por lo que permitía, no por lo que ocurrió.

Hallazgo 2 — 61 acciones de terceros sin fijar.

```
$ grep -rhoE 'uses:\s*[^ ]+@[^ ]+' .github/workflows/ \
  | grep -vE '@[0-9a-f]{40}' | sort -u | wc -l
61
```

Sesenta y una referencias por etiqueta, de diecinueve autores distintos. Cada una es código de terceros ejecutado con los permisos del trabajo, y cada etiqueta la puede mover su autor.

```
acciones fijadas por revisión                0 de 61        61 de 61
autores distintos                               19              6
lista de permitidas en la organización       no había         activa
robot que propone actualizar                 no había         semanal
```

La reducción de diecinueve autores a seis salió de revisar para qué servía cada una: nueve hacían algo que se resolvía con tres líneas de orden.

Hallazgo 3 — inyección por el título de la propuesta.

```
$ grep -rnE 'run:.*\$\{\{\s*github\.event\.' .github/workflows/
.github/workflows/ci.yml:41:      - run: echo "PR: ${{ github.event.pull_request.title }}" >> notas.
txt
.github/workflows/etiquetar.yml:18: - run: gh pr edit ${{ github.event.pull_request.number }} …
```

Dos casos. El primero permitía ejecutar órdenes con solo poner el texto adecuado en el título de una propuesta.

```
campos del evento interpolados en órdenes         2               0
mecanismo                              interpolación directa   variable de entorno
comprobación en la canalización            no había      falla si aparece el patrón
```

Hallazgo 4 — un ejecutor propio atendiendo propuestas del repositorio público.

```
$ grep -rlZ 'self-hosted' .github/workflows/ | xargs -0 grep -l 'pull_request'
.github/workflows/integracion-lenta.yml
```

El ejecutor estaba en la red de preproducción, con acceso a la base de datos de integración y visibilidad de la red interna. Y no era efímero: conservaba el sistema de ficheros entre ejecuciones.

```
ejecutores propios                        2, persistentes   3, efímeros
ubicación                               red de preproducción  red aislada propia
acceso a la red interna                       completo        solo al registro
                                                              de artefactos
atienden propuestas externas                     sí              no
coste mensual                                ~90 USD          ~140 USD
```

Cincuenta dólares más al mes por eliminar un camino de ejecución de código arbitrario dentro de la red.

Hallazgo 5 — tres claves de nube de larga duración.

```
$ gh secret list --json name -q '.[].name' | grep -Ei 'aws_secret|gcp_sa_key'
AWS_SECRET_ACCESS_KEY
AWS_SECRET_ACCESS_KEY_PRE
GCP_SA_KEY
```

Las tres eran anteriores a la federación de las clases 050 y 059, y ninguna se había retirado al migrar.

```
claves de nube en secretos                       3              0
identidad federada                            parcial         en los 22 flujos
condición de confianza acotada a entorno      2 de 3          3 de 3
secretos totales en la organización              41              12
```

Y la prueba negativa, quinta vez en el programa:

```
desde una rama de trabajo, pedir la identidad de producción   denegado   ✓
desde la rama principal sin el entorno protegido              denegado   ✓
desde la rama principal con el entorno y la aprobación        concedido  ✓
```

Y el efecto sobre el tiempo de respuesta, que se midió antes y después:

```
mediana                                      6 min 40 s       7 min 10 s
```

Treinta segundos más por las comprobaciones añadidas. Se aceptó porque sigue por debajo del umbral de la clase 097, y porque la alternativa —quitarlas— es la ley 16 en su versión más cara.

Resumen:

```
caminos de escalada abiertos                    4             0
acciones fijadas por revisión                 0 de 61       61 de 61
autores de acciones de terceros                 19             6
claves de nube en secretos                       3             0
secretos totales                                41            12
ejecutores propios efímeros y aislados        0 de 2         3 de 3
permisos declarados por trabajo                0 de 22       22 de 22
mediana del tiempo de respuesta             6 min 40 s     7 min 10 s
```

La lección que esta clase traslada al resto de la parte 08: los cinco hallazgos existían desde hacía entre catorce meses y tres años, en un repositorio con revisión obligatoria y comprobaciones de seguridad — porque ninguna de esas comprobaciones miraba los propios flujos de la canalización. Y el primero permitía a cualquiera de internet ejecutar código con las credenciales de despliegue de producción, lo que confirma la primera predicción de la hipótesis de esta parte: la canalización es el objetivo más valioso del sistema, y hasta esta auditoría era el menos vigilado.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/098-github-actions-workflows-runners-permisos-y-cache/lab.py
```

El laboratorio selecciona el motor de práctica delivery y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa pipeline-github-actions en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es un pipeline con gates, promoción y rollback. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye pipeline-github-actions para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

Errores frecuentes

Síntoma	Causa probable	Corrección
Cualquiera que abra una propuesta puede ejecutar código con los secretos del repositorio	Un disparador privilegiado descarga y ejecuta el código de la propuesta	Separa en dos flujos: uno construye sin secretos y deja un artefacto; el otro, con privilegios, solo lee ese artefacto.
Un texto en el título de una propuesta ejecuta órdenes	El campo del evento se interpola dentro de la orden antes de ejecutarla	Pásalo como variable de entorno; nunca interpoles campos del evento dentro de un run.
Una acción de terceros cambia de comportamiento sin que nadie lo decida	Está referenciada por etiqueta y quien la controla puede moverla	Fija por revisión exacta, mantén una lista de acciones permitidas y un robot que proponga actualizaciones.
Un ejecutor propio ejecuta código de propuestas externas dentro de la red	Se usa un ejecutor de la organización en flujos abiertos a contribuciones	Ejecutores efímeros, en red aislada, y nunca para propuestas de fuera; esas van a ejecutores del proveedor sin secretos.
Una caché escrita desde una propuesta se usa en la rama principal	Las claves no separan por confianza	Separa las cachés por origen y no restaures cachés de ramas ajenas en trabajos con permisos.
Existen claves de nube de larga duración en los secretos del repositorio	Quedaron al migrar a identidad federada y nadie las retiró	Migra los flujos restantes, acota la confianza a repositorio y entorno, y retira las claves comprobando con la prueba negativa.

Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

Preguntas de comprobación

1. ¿Cuáles son los cuatro sitios por los que entra código que la organización no controla?
2. ¿Por qué un disparador privilegiado combinado con descargar el código de la propuesta es una escalada, y cuál es el patrón correcto?
3. ¿Por qué fijar una acción por etiqueta es insuficiente, y qué razón lo hace más grave aquí que en un módulo?
4. ¿Qué tres medidas hacen aceptable un ejecutor propio, y cuál no se puede negociar?
5. ¿Por qué la separación entre identidad de lectura y de escritura no es un formalismo?

Referencias

- GitHub (2025). Security hardening for GitHub Actions — disparadores, inyección, permisos y acciones de terceros. <https://docs.github.com/en/actions/security-for-github-actions/security-guides/security-hardening-for-github-actions>
- GitHub (2025). About OpenID Connect in GitHub Actions — identidad federada y condiciones de confianza. <https://docs.github.com/en/actions/security-for-github-actions/security-hardening-your-deployments/about-security-hardening-with-openid-connect>
- GitHub (2025). Self-hosted runners: security considerations — riesgos y ejecutores efímeros. <https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/about-self-hosted-runners>
- GitHub (2025). Caching dependencies — claves, claves de respaldo, alcance y expulsión. <https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/caching-dependencies-to-speed-up-workflows>
- OpenSSF (2025). Scorecard checks for CI/CD — fijación por revisión, permisos y automatización de la revisión. <https://github.com/ossf/scorecard/blob/main/docs/checks.md>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Anterior	Índice	Siguiente
← 097 · Integración continua, trunk-based development y feedback	Parte 08 · Programa	099 · Artefactos inmutables, semver y promoción →

Evaluación — 098 GitHub Actions: workflows, runners, permisos y caché

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega pipeline-github-actions con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.

099 — Artefactos inmutables, semver y promoción

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 4 Laboratorio: supply-chain · Estado: EXECUTABLECORE

Propósito

Convertir el artefacto en la unidad que se promueve, con una pregunta que decide si el sistema es auditable: ¿qué hay ahora mismo en producción, de qué commit salió y qué demuestra que se probó? Si responderla lleva más de un minuto, la promoción no existe: hay reconstrucciones con el mismo nombre. La clase fija qué hace promocionable a un artefacto, qué información viaja con él, y por qué el versionado semántico resuelve un problema que muchos servicios no tienen.

Resultados de aprendizaje

Al finalizar podrás:

1. Distinguir promoción real de reconstrucción por entorno, con la comprobación que las separa.
2. Decidir el esquema de versionado según haya o no consumidores externos.
3. Adjuntar al artefacto la evidencia de lo que ha superado, y consultarla.
4. Responder en segundos qué se ejecuta en producción y qué lo respalda.
5. Definir una política de retención que no borre lo que está en uso.

Conceptos centrales

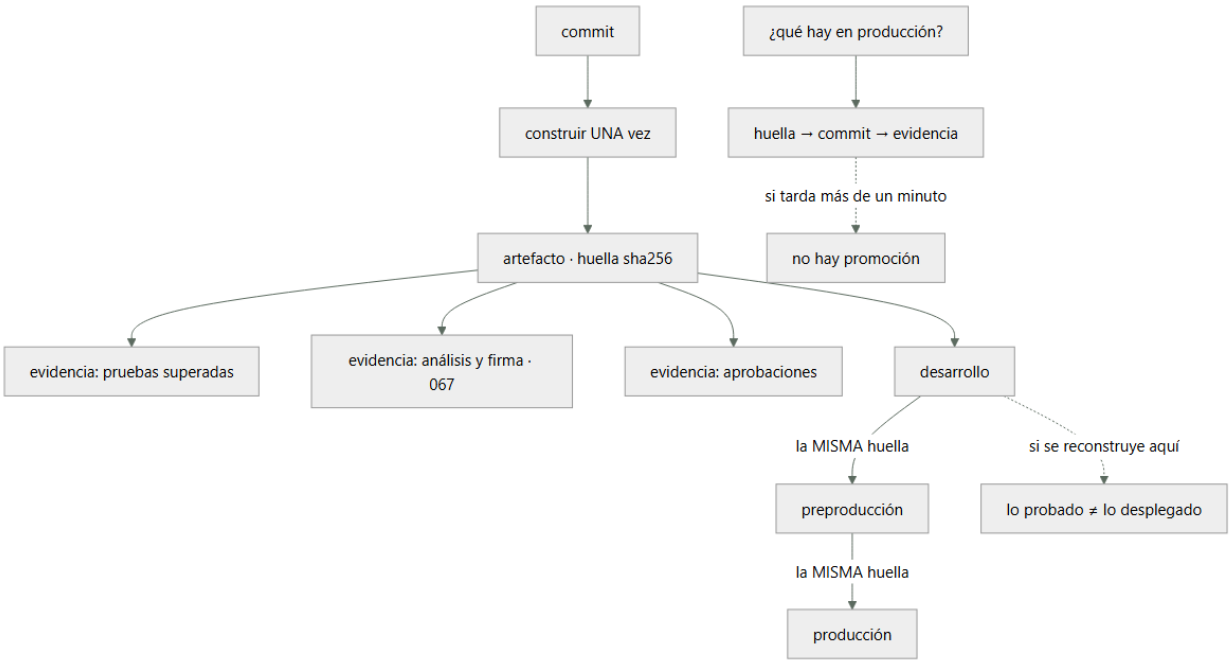
Concepto	Comprensión verificable
artefacto promocionable	Identificado por huella, inmutable, con su origen y su evidencia adjuntos. Lo que lo hace promocionable no es su formato: es que no cambia al pasar de entorno.
promoción	Mover la misma huella al siguiente entorno. Si en algún punto se reconstruye, lo probado y lo desplegado dejan de ser lo mismo.
versionado semántico	Contrato con consumidores: la versión anuncia si un cambio rompe. Resuelve un problema real en bibliotecas y aporta poco en un servicio sin consumidores externos.
trazabilidad	Cadena verificable de commit a artefacto y de artefacto a despliegue. Es lo que un servicio necesita de verdad, tenga o no versión semántica.
evidencia adjunta	Declaraciones asociadas a la huella: qué pruebas superó, qué análisis pasó, quién aprobó. Es la procedencia de la clase 067 extendida al ciclo de entrega.
retención por referencia	No se borra lo que algún entorno referencia, con independencia de su antigüedad. Es la regla de la clase 067 aplicada a todos los artefactos.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. Promover no es reconstruir

La clase 062 estableció el principio y aquí se convierte en una comprobación:

```
# la huella desplegada en cada entorno
$ for e in dev pre pro; do
    echo -n "$e: "; kubectl --context=$e get deploy tienda \
      -o jsonpath='{.spec.template.spec.containers[0].image}'; echo
  done
dev: registro/tienda@sha256:9f2c4a1b…
pre: registro/tienda@sha256:9f2c4a1b…
pro: registro/tienda@sha256:9f2c4a1b…
```

Tres huellas idénticas: hay promoción. Tres huellas distintas del mismo commit: hay tres artefactos que nadie ha comparado.

Y las cuatro formas de romperlo sin darse cuenta, que aparecen todas en organizaciones reales:

```
1. reconstruir por entorno            clase 062: el `ARG ENTORNO` en el Dockerfile
2. desplegar por etiqueta             clase 061: la etiqueta se mueve
3. reconstruir para "actualizar la base de seguridad" antes de producción
   → sale un artefacto distinto del que se probó, con la mejor intención
4. una canalización por entorno, cada una con su paso de construcción
```

La tercera merece detalle porque su motivo es bueno y su efecto es el mismo: si la base ha cambiado y hay que reconstruir, lo que sale es un artefacto nuevo que no ha pasado por preproducción. La forma correcta es reconstruir y volver a promover desde el principio, no inyectar el artefacto nuevo al final.

Y lo que hace promocionable a un artefacto son cuatro propiedades, todas ya establecidas:

```
identificado por huella           clase 061
inmutable en el registro          etiquetas inmutables, clase 061
con su origen verificable         procedencia, clase 067
con la configuración FUERA        clases 062, 076, 089
```

La cuarta es la que hace posibles las tres primeras. Un artefacto que lleva dentro la configuración de un entorno no puede ser el mismo en todos, así que no se puede promover.

Y una consecuencia práctica sobre el registro: la promoción no copia el artefacto. Es un cambio de referencia en el destino, no un movimiento de bytes. Copiar entre registros por entorno es una decisión de aislamiento legítima —producción no descarga del mismo sitio que desarrollo— y hay que hacerla conservando la huella:

```
$ crane copy registro-dev/tienda@sha256:9f2c… registro-pro/tienda@sha256:9f2c…
$ crane digest registro-pro/tienda@sha256:9f2c…
```

```
sha256:9f2c4a1b…                                                ✓
```

Si al copiar cambia la huella, no se ha copiado: se ha reconstruido.

2. Versionar: para quién y para qué

El versionado semántico resuelve un problema concreto —avisar a los consumidores de si un cambio rompe— y se aplica con frecuencia donde ese problema no existe.

```
MAYOR   cambia de forma incompatible: el consumidor tiene que hacer algo
MENOR   añade sin romper
PARCHE  corrige sin cambiar el contrato
```

Y la pregunta que decide si aporta:

```
¿hay consumidores que eligen QUÉ VERSIÓN usar?
  sí  → biblioteca, módulo, acción, imagen base, interfaz publicada
        el versionado semántico es un contrato y hay que respetarlo
  no  → un servicio que se despliega entero y del que solo hay una versión viva
        el número aporta poco; lo que hace falta es TRAZABILIDAD
```

La segunda situación es la de la mayoría de los servicios de una organización, y ahí el esquema útil es otro:

```
versión derivada del origen
  2026.08.03-a1b2c3d      fecha y revisión
  o simplemente la huella, con la revisión en una etiqueta estándar (clase 061)
```

Lo que hay que poder responder no es «qué versión es» sino de qué commit salió y qué ha superado, y eso lo dan las etiquetas y la evidencia, no el número.

Y para lo que sí tiene consumidores, la disciplina de la clase 088 se aplica entera, con una precisión que allí se estableció y conviene repetir:

```
un cambio es INCOMPATIBLE aunque la interfaz no cambie
si su efecto destruye o recrea algo en el consumidor
```

Y tres prácticas que hacen que el versionado signifique algo:

```
notas por versión, con las incompatibilidades señaladas
política de soporte: cuántas versiones mayores se mantienen
y un plazo de retirada anunciado, no descubierto
```

Y una advertencia sobre el versionado automático a partir de los mensajes de commit: funciona y traslada una decisión de contrato a una convención de escritura. Quien escribe el mensaje decide si la versión sube de mayor, y eso solo es aceptable si el equipo entiende que está declarando un contrato. Sin esa comprensión, produce versiones mayores por errores tipográficos y cambios incompatibles publicados como parche.

Y el caso de las versiones previas, que resuelve un problema real de la promoción:

```
3.2.0-rc.1     candidata: se puede promover hasta preproducción
3.2.0          publicada: la misma huella, con otra etiqueta
```

Con la condición de siempre: la huella no cambia al publicar. Si publicar reconstruye, se ha vuelto al problema del apartado anterior.

3. La evidencia viaja con el artefacto

La clase 067 estableció que un artefacto puede llevar asociados objetos por su huella: inventario, firma y procedencia. La misma mecánica sirve para la evidencia del ciclo de entrega:

```
qué pruebas superó, y con qué resultado
qué análisis de seguridad pasó, y en qué versión de las reglas
qué entornos ha recorrido, y cuándo
quién aprobó su paso a producción
```

Y el valor de adjuntarla en vez de guardarla en la canalización es que sobrevive a la canalización: los registros de las ejecuciones caducan, y la evidencia asociada a la huella no.

```
$ cosign attest --predicate pruebas.json --type https://cloudshop.example/pruebas/v1 \
    registro/tienda@sha256:9f2c…

$ cosign verify-attestation --type https://cloudshop.example/pruebas/v1 \
    registro/tienda@sha256:9f2c… … | jq -r '.payload' | base64 -d \
  | jq '{suite: .predicate.suite, superadas: .predicate.superadas, commit: .predicate.commit}'
{
```

```
  "suite": "integracion-v4",
  "superadas": 1284,
  "commit": "a1b2c3d4e5f6…"
}
```

Y la promoción pasa a tener una condición verificable en vez de un paso manual:

```
para promover a preproducción   debe existir evidencia de la suite de integración
para promover a producción      además, evidencia de preproducción y una aprobación
```

Eso convierte la puerta de promoción en una comprobación de admisión —la misma idea de la clase 067— y tiene una propiedad importante: no depende de que la canalización se comporte bien. Un artefacto que llegara a producción por otra vía no tendría la evidencia, y la admisión lo rechazaría.

Y el registro de promociones, que es lo que responde la pregunta que abre la clase:

```
$ ./que-hay-en-produccion.sh
servicio   huella          commit    desde        evidencia
tienda     sha256:9f2c…    a1b2c3d   hace 3 días  pruebas ✓ análisis ✓ aprobó: ana
api        sha256:c74e…    e5f6a7b   hace 6 h     pruebas ✓ análisis ✓ aprobó: luis
informes   sha256:81ab…    ?         hace 4 meses  ninguna              ← revisar
```

La última fila es el tipo de hallazgo que este ejercicio produce siempre: un artefacto en producción sin evidencia y sin commit conocido, que en la clase 067 resultó ser una imagen construida desde un portátil.

Y una precisión sobre la aprobación: registrarla junto al artefacto y no solo en la canalización responde a la pregunta que aparece en cualquier auditoría —quién autorizó lo que hay ahora mismo en producción— sin depender de la retención del sistema de integración continua.

4. Retención: lo que no se puede borrar

Un registro de artefactos crece sin parar, y las políticas de limpieza por antigüedad producen un fallo concreto y grave:

```
se borra por antigüedad una imagen de hace ocho meses
y producción la está usando, porque ese servicio no ha cambiado
→ cualquier nodo nuevo no puede arrancar
→ y la vuelta atrás a esa versión deja de existir
```

La regla es la de la clase 067 y aplica a todos los artefactos:

```
nunca se borra lo que algún entorno referencia
nunca se borra lo que sea el destino de una vuelta atrás plausible
el resto, por antigüedad
```

Y su implementación es una comparación entre dos listas, la misma que las clases 049, 087 y 095 usaron para otras cosas:

```
# lo que está desplegado en algún entorno
$ for e in dev pre pro; do
    kubectl --context=$e get deploy -A -o jsonpath='{..image}' | tr ' ' '\n'
  done | grep -o 'sha256:[0-9a-f]*' | sort -u > en-uso.txt

# lo que hay en el registro
$ crane ls registro/tienda | while read t; do crane digest registro/tienda:$t; done \
  | sort -u > en-registro.txt

# candidatos a borrar: en el registro y no en uso
$ comm -13 en-uso.txt en-registro.txt | head
```

Y dos matices que evitan borrar de más:

```
conservar las N últimas de cada servicio, aunque no estén en uso
  → para poder volver atrás varios pasos
conservar lo que tenga evidencia de haber estado en producción
  → para poder auditar hacia atrás
```

Y el coste de no limpiar, que también es real:

```
un registro con decenas de miles de artefactos
  → listados lentos, búsquedas inútiles, y almacenamiento que se paga
```

Y la comprobación que dice si la política funciona:

```
# ningún artefacto en uso figura como candidato a borrar
$ comm -12 en-uso.txt candidatos.txt | wc -l
0                                                                    ✓
```

Y un caso especial que conviene anticipar: las imágenes base y los módulos. Su retención no se decide por lo que está desplegado sino por lo que puede necesitar reconstruirse. Una imagen base retirada impide reconstruir una versión antigua de cualquier servicio que la usara, lo que convierte una vuelta atrás en imposible aunque el artefacto final siga existiendo.

5. La pregunta que decide si el sistema es auditable

Todo lo anterior existe para responder cuatro preguntas en segundos. Conviene tenerlas escritas y comprobar cuánto se tarda:

```
1. ¿qué hay ahora mismo en producción?
2. ¿de qué commit salió?
3. ¿qué demuestra que se probó, y quién lo aprobó?
4. ¿a qué versión anterior puedo volver, y sigue existiendo?
```

Y el orden en que se responden, que es siempre el mismo:

```
huella desplegada → etiquetas del artefacto → commit
                  → evidencia adjunta → pruebas, análisis, aprobación
                  → historial de despliegues → versión anterior
```

Si alguna de las cuatro exige entrar en el sistema de integración continua, buscar en registros o preguntar a alguien, la cadena está rota en ese punto.

Y la lista de comprobación de la clase:

- [ ] una sola construcción por commit, y la misma huella en los tres entornos
- [ ] despliegue por huella, nunca por etiqueta
- [ ] etiquetas inmutables en el registro
- [ ] configuración fuera del artefacto
- [ ] copia entre registros conservando la huella, verificada
- [ ] versionado semántico solo donde hay consumidores que eligen versión
- [ ] trazabilidad de commit a artefacto en etiquetas estándar
- [ ] evidencia adjunta al artefacto: pruebas, análisis y aprobaciones
- [ ] promoción condicionada a la evidencia, verificada en la admisión
- [ ] retención por referencia, con las N últimas conservadas
- [ ] las cuatro preguntas respondidas en menos de un minuto

Y el cierre que enlaza con las clases siguientes: este artefacto, con su evidencia, es lo que las clases 102 y 103 van a mover a producción. Lo que decide si un despliegue es reversible es que el artefacto anterior siga existiendo y siga siendo desplegable, y eso no lo garantiza la estrategia de despliegue sino la política de retención de esta clase.

Ejemplo trabajado

CloudShop responde por primera vez las cuatro preguntas de esta clase. Tarda dos días y medio, y el ejercicio destapa que lo que llamaban promoción eran cuatro construcciones distintas.

La pregunta, y lo que costó responderla.

```
pregunta                                  tiempo
¿qué hay en producción?                   20 min (mirando cada despliegue)
¿de qué commit salió?                     1 día  (correlacionando por fecha)
¿qué demuestra que se probó?              no se pudo responder
¿a qué versión puedo volver?              1 día  (buscando en el registro)
```

La tercera no se pudo responder porque los registros de las ejecuciones de hace más de noventa días ya no existían.

Hallazgo 1 — cuatro construcciones por commit.

```
$ for e in dev pre pro; do
    echo -n "$e: "; kubectl --context=$e get deploy tienda \
      -o jsonpath='{.spec.template.spec.containers[0].image}'; echo
  done
dev: registro/tienda:2026.07.28-a1b2c3d
pre: registro/tienda:2026.07.28-a1b2c3d
pro: registro/tienda:2026.07.28-a1b2c3d
```

Misma etiqueta en los tres. Y las huellas:

```
$ for e in dev pre pro; do crane digest registro-$e/tienda:2026.07.28-a1b2c3d; done
sha256:9f2c4a1b…
sha256:71ae0c9d…
sha256:c74e0182…
```

Tres huellas distintas con la misma etiqueta y el mismo commit. Cada entorno tenía su canalización con su paso de construcción, y una cuarta reconstruía antes de producción «para actualizar la base de seguridad».

```
construcciones por commit                      4                1
canalizaciones de construcción                 4                1
despliegue por                              etiqueta            huella
reconstrucción previa a producción           sí, siempre    volver a promover
                                                            desde el principio
lo probado y lo desplegado                  artefactos      la misma huella
                                            distintos
```

Y la comparación de las tres imágenes explicó dos incidentes archivados sin causa:

```
diferencias entre la de preproducción y la de producción
  versión de una biblioteca del sistema:  2
  versión de una dependencia transitiva:  1
```

Una de las dos bibliotecas era la que causaba un fallo intermitente que solo aparecía en producción y que nunca se había podido reproducir.

Hallazgo 2 — la evidencia no sobrevivía a la canalización.

```
retención de los registros de ejecución        90 días
servicios en producción desplegados hace más   6 de 15
evidencia disponible para esos seis            ninguna
```

```
evidencia                            en los registros    adjunta al artefacto
                                     de la canalización   por huella
sobrevive a la retención                    no                  sí
tipos adjuntos                               —          pruebas, análisis,
                                                         entornos recorridos,
                                                         aprobaciones
promoción condicionada a la evidencia       no          verificada en la admisión
```

Y al aplicar la condición de admisión, un servicio dejó de poder desplegarse:

```
informes: sin evidencia de pruebas ni procedencia
```

Era la imagen que la clase 067 había encontrado construida desde un portátil catorce meses atrás. Seguía en producción.

Hallazgo 3 — la política de retención iba a romper dos servicios.

```
$ comm -12 en-uso.txt candidatos-a-borrar.txt
sha256:81ab…    (informes, desplegado hace 4 meses)
sha256:3c4d…    (exportador-fiscal, desplegado hace 7 meses)
```

La política borraba por antigüedad a los 180 días. Dos artefactos en producción estaban en la lista de candidatos, y uno de ellos a doce días de cumplir el plazo.

```
política                                  por antigüedad    por referencia +
                                                            las 10 últimas
artefactos en uso en la lista de borrado        2                 0
comprobación antes de ejecutar la limpieza   no había          obligatoria
imágenes base con retención propia           no había         conservadas mientras
                                                              algo pueda reconstruirse
```

La última fila salió de una pregunta que nadie se había hecho: volver atrás a una versión de hace ocho meses exige que su imagen base siga existiendo si hubiera que reconstruirla.

Hallazgo 4 — el versionado que no significaba nada.

```
servicios con versión semántica                15 de 15
servicios con consumidores que eligen versión   2 de 15
versiones mayores publicadas en un año          31
de ellas, cambios realmente incompatibles        3
```

El versionado automático a partir de los mensajes de commit producía versiones mayores por convenciones mal escritas. Trece servicios mantenían un contrato que nadie consumía.

```
servicios con versión semántica              15 de 15         2 de 15
los otros trece                                 —          fecha y revisión,
                                                            con trazabilidad
versiones mayores al año                        31                2
notas por versión en los dos con consumidores  no había           obligatorias
```

Y las cuatro preguntas, medidas de nuevo:

```
$ time ./que-hay-en-produccion.sh
servicio   huella        commit    desde       pruebas  análisis  aprobó
tienda     sha256:9f2c…  a1b2c3d   3 días      ✓        ✓         ana
api        sha256:c74e…  e5f6a7b   6 h         ✓        ✓         luis
… (15 servicios)

real    0m4,2s
```

Resumen:

```
construcciones por commit                        4             1
huellas distintas del mismo commit               3             1
evidencia disponible tras 90 días          0 de 6         15 de 15
artefactos en uso en la lista de borrado         2             0
servicios con versión semántica sin consumidores 13            0
tiempo para responder las cuatro preguntas  2,5 días         4,2 s
artefactos en producción sin procedencia         1             0
```

La lección que esta clase traslada al resto de la parte 08: lo que el equipo llamaba promoción eran cuatro construcciones con el mismo nombre, y la comprobación que lo destapa es comparar tres huellas — quince segundos de trabajo que nadie había hecho en tres años. Y una de las diferencias entre la imagen de preproducción y la de producción explicaba un fallo intermitente archivado sin causa. Cuando lo probado y lo desplegado son artefactos distintos, cualquier prueba habla de otra cosa.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/099-artefactos-inmutables-semver-y-promocion/lab.py
```

El laboratorio selecciona el motor de práctica supply-chain y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa registro-artefactos en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es procedencia, inventario y verificación del artefacto. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye registro-artefactos para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

Errores frecuentes

Síntoma	Causa probable	Corrección
El mismo commit produce artefactos distintos en cada entorno	Hay una construcción por entorno, o se reconstruye antes de producción	Una sola construcción por commit; si hay que actualizar la base, se vuelve a promover desde el principio.
Un fallo solo aparece en producción y no se puede reproducir	El artefacto de producción no es el que se probó	Compara las huellas de los tres entornos: si difieren, la causa está ahí antes que en el código.
No se puede demostrar qué pruebas superó lo que está en producción	La evidencia vive en los registros de la canalización, que caducan	Adjunta la evidencia al artefacto por su huella; sobrevive a la retención del sistema de integración.
Una limpieza del registro deja un servicio sin poder escalar	La retención es por antigüedad y ese artefacto llevaba meses desplegado sin cambios	Retención por referencia: nunca se borra lo que un entorno usa ni lo que sea destino de una vuelta atrás.
Las versiones mayores se publican constantemente sin cambios incompatibles	El versionado automático traslada una decisión de contrato a una convención de escritura	Usa versionado semántico solo donde hay consumidores que eligen versión; para el resto, trazabilidad.
Copiar un artefacto entre registros cambia su identificador	Se ha reconstruido en vez de copiado	Copia conservando la huella y verifícala en el destino antes de dar la promoción por buena.

Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

Preguntas de comprobación

1. ¿Qué comprobación de quince segundos distingue una promoción real de cuatro reconstrucciones?
2. Enumera las cuatro formas de romper la promoción sin darse cuenta, incluida la de mejor intención.
3. ¿Cuándo aporta el versionado semántico y qué necesita un servicio que no tiene consumidores externos?
4. ¿Por qué la evidencia se adjunta al artefacto en vez de guardarse en la canalización?
5. ¿Qué regla de retención evita que una limpieza deje un servicio sin poder escalar?

Referencias

- Jez Humble, David Farley (2010). Continuous Delivery, cap. 13 — gestión de artefactos y promoción entre entornos. <https://continuousdelivery.com/>
- Semantic Versioning (2013). SemVer 2.0.0 — el contrato que expresa una versión. <https://semver.org/>
- Sigstore (2025). In-toto attestations with cosign — evidencia adjunta a un artefacto por su huella. <https://docs.sigstore.dev/cosign/verifying/attestation/>
- OCI (2025). Referrers API — objetos asociados a un artefacto en el registro. <https://github.com/opencontainers/distribution-spec/blob/main/spec.md#listing-referrers>
- Google (2025). Artifact Registry cleanup policies — retención y exclusión de lo que está en uso. <https://cloud.google.com/artifact-registry/docs/repositories/cleanup-policy>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 08 en PDF · Manual integral

Anterior	Índice	Siguiente
← 098 · GitHub Actions: workflows, runners, permisos y caché	Parte 08 · Programa	100 · Pruebas, calidad y puertas de cambio →

Evaluación — 099 Artefactos inmutables, semver y promoción

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega registro-artefactos con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.

100 — Pruebas, calidad y puertas de cambio

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 4 Laboratorio: testing · Estado: EXECUTABLECORE

Propósito

Decidir qué se prueba, a qué nivel y qué detiene un cambio, con una distinción que ordena el resto: una prueba es una puerta solo si su fallo detiene el cambio; las demás son documentación con coste de mantenimiento. La clase trata la forma de la pirámide como un compromiso entre coste y confianza y no como un dogma, explica por qué la cobertura dice lo que no está probado y nada sobre lo que sí, y llega al límite honesto: hay comportamientos que solo se pueden comprobar en producción, y para eso están las clases 102 y 105.

Resultados de aprendizaje

Al finalizar podrás:

1. Elegir el nivel de prueba por lo que puede detectar y por lo que cuesta mantener.
2. Sustituir pruebas frágiles entre servicios por contratos verificados en ambos lados.
3. Interpretar la cobertura sin convertirla en objetivo.
4. Clasificar cada comprobación en puerta o informe, y justificarlo.
5. Reconocer lo que no se puede probar antes de desplegar y qué mecanismo lo cubre.

Conceptos centrales

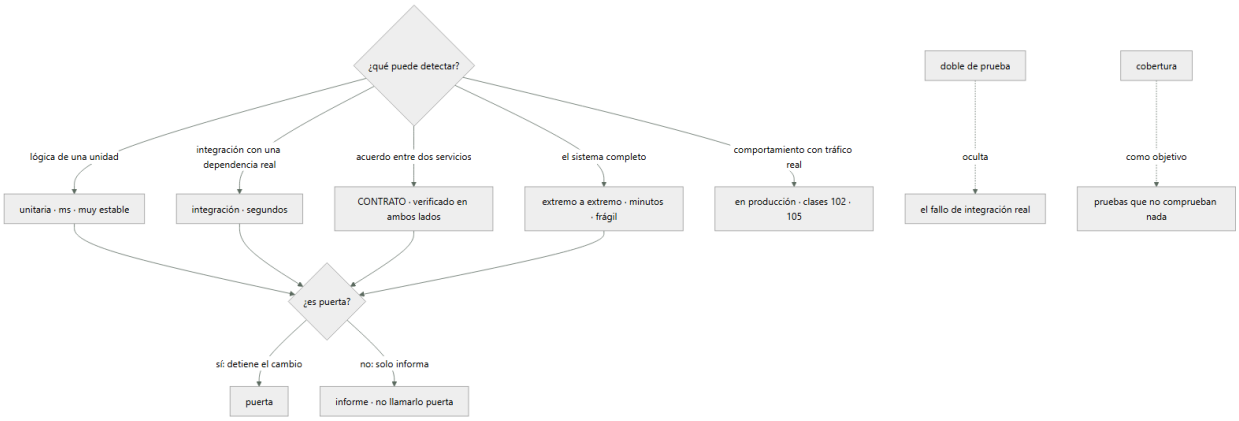
Concepto	Comprensión verificable
puerta	Comprobación cuyo fallo detiene el cambio. Si no detiene, es un informe; llamarla puerta produce una falsa sensación de control.
doble de prueba	Sustituto de una dependencia. Hace la prueba rápida y estable, y oculta precisamente el fallo de integración que a veces es el que importa.
prueba de contrato	Acuerdo entre quien consume y quien provee, verificado por separado en ambos lados. Da confianza de integración sin desplegar los dos sistemas juntos.
cobertura	Proporción de código ejecutado por las pruebas. Dice qué no está probado; no dice nada sobre la calidad de lo que sí se ejecutó.
prueba que nunca falla	La que no ha detectado nada en años. Cuesta mantenimiento y tiempo de ejecución, y su valor hay que justificarlo como el de cualquier otra.
verificación en producción	Comprobar en el sistema real lo que ningún entorno previo puede reproducir. No sustituye a las pruebas: cubre lo que queda fuera de su alcance.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. La forma la decide la arquitectura, no el dogma

La pirámide de pruebas se enseña como una proporción fija y es un compromiso entre dos magnitudes:

```
coste       tiempo de ejecución, fragilidad y mantenimiento
confianza   qué clase de fallo puede detectar
```

Y su forma correcta depende de qué hace el servicio:

```
un servicio con lógica de negocio densa
  → muchas unitarias: ahí está el riesgo

un servicio que sobre todo INTEGRA
  llama a tres APIs, transforma y publica un mensaje
  → sus unitarias comprueban muy poco: la lógica es la integración
  → el peso se desplaza a integración y contrato

una aplicación de interfaz
  → el riesgo está en el recorrido completo, no en las funciones
```

Aplicar la misma proporción a los tres produce el resultado que se ve a menudo: un servicio de integración con el 90 % de cobertura de unitarias y ningún fallo detectado por ellas, porque todo lo que puede romperse ocurre en las fronteras.

Y lo que cada nivel puede y no puede detectar, que es el criterio real de elección:

```
unitaria        lógica, casos límite, errores de cálculo
                NO detecta: contratos rotos, configuración, concurrencia real

integración     el uso real de una dependencia: consultas, esquemas, tiempos
                NO detecta: el comportamiento del otro servicio

contrato        que dos partes siguen entendiendo lo mismo
                NO detecta: que el sistema completo funcione

extremo a extremo   un recorrido real completo
                NO detecta: casi nada más, y cuesta mucho por lo que da

en producción   carga, datos reales, comportamiento de usuarios
                es lo ÚNICO que detecta esos tres
```

La última fila es la que se resiste a aceptarse y es la más honesta: hay clases enteras de fallo que ningún entorno previo puede reproducir, porque dependen del volumen, de la variedad de los datos reales o del comportamiento de los usuarios.

Y sobre las pruebas de extremo a extremo, una posición que conviene defender con datos: son las más caras de mantener, las más frágiles (clase 097) y las que menos fallos únicos detectan. La regla práctica:

```
unas pocas, sobre los recorridos que de verdad no pueden fallar
y cada una justificada: qué detecta que no detecte nada más barato
```

Una suite de doscientas pruebas de extremo a extremo es casi siempre una suite de veinte pruebas y ciento ochenta duplicados frágiles de lo que ya cubren niveles inferiores.

2. Los dobles y el contrato

Un doble de prueba sustituye una dependencia para que la prueba sea rápida y estable. Y tiene un coste que hay que nombrar:

```
el doble se comporta como YO CREO que se comporta la dependencia
y si mi creencia es errónea, la prueba pasa y el sistema falla
```

Ese es el fallo característico: una suite verde con una integración rota, porque el doble devolvía lo que el autor esperaba y no lo que el otro sistema devuelve de verdad.

Y la respuesta no es eliminar los dobles —sin ellos las pruebas son lentas e inestables— sino verificar el contrato por separado:

```
el consumidor declara qué espera        "si pido /precios/42, recibo {id, importe}"
esa expectativa se guarda como contrato
y el proveedor verifica, en SU canalización, que lo cumple
```

```
# lo que el consumidor espera, generado desde sus propias pruebas
interacciones:
  - descripcion: precio de un artículo existente
    peticion:  { metodo: GET, ruta: /precios/42 }
    respuesta:
      estado: 200
      cuerpo: { id: 42, importe: 1990, moneda: "EUR" }
```

Y las dos propiedades que lo hacen valioso:

```
no hace falta desplegar los dos sistemas juntos
y el proveedor se entera de que ROMPE a alguien antes de publicar
```

La segunda es la que resuelve el problema real. Sin contratos, un cambio incompatible del proveedor se detecta cuando el consumidor falla en un entorno compartido, y para entonces ya está fusionado.

Y la disciplina que hace que funcione:

```
el contrato lo genera el consumidor desde sus pruebas, no se escribe a mano
el proveedor lo verifica en cada cambio, y su fallo es una PUERTA
y un contrato que ningún consumidor usa se retira
```

La tercera evita el problema que estas herramientas tienen a los dos años: decenas de contratos de consumidores que ya no existen, bloqueando cambios legítimos.

Y una decisión práctica sobre qué dependencias merecen una prueba de integración real en vez de un doble:

```
la base de datos                sí, casi siempre: el esquema y las consultas
                                son donde más se rompe
el sistema de mensajería        sí, si la lógica depende de su semántica
un servicio propio              contrato, no integración
un servicio de terceros         doble en las pruebas, y una comprobación
                                sintética contra el real, programada
```

La última fila resuelve un caso frecuente: no se puede depender de la disponibilidad de un tercero para fusionar un cambio, y tampoco se puede ignorar que cambie. Una comprobación programada contra el servicio real detecta el cambio sin bloquear a nadie.

3. La cobertura dice lo que no está probado

La cobertura mide qué proporción del código ejecutan las pruebas. Y lo que se puede concluir de ella es asimétrico:

```
cobertura baja en un módulo    ese código NO está probado: es información útil
cobertura alta en un módulo    ese código se EJECUTÓ durante las pruebas
                               y no dice nada sobre si se comprobó algo
```

La asimetría importa porque una prueba puede ejecutar código sin comprobar nada:

```
def test_calcular_precio():
    calcular_precio(articulo, cliente)      # ninguna comprobación
```

Esa prueba da cobertura y solo detecta que la función no lanza una excepción. Y cuando la cobertura es un objetivo, ese es exactamente el tipo de prueba que aparece — la ley de Goodhart en su forma más literal: una medida que se convierte en objetivo deja de ser una buena medida.

La forma útil de usarla:

```
como mapa      ¿qué partes críticas no tienen ninguna prueba?
sobre el cambio  ¿este cambio añade código sin probar?
nunca como objetivo global   "llegar al 80 %" produce pruebas sin valor
```

Y la segunda es la única que funciona bien como puerta:

```
# la cobertura del código NUEVO, no la global
$ diff-cover cobertura.xml --compare-branch=origin/main --fail-under=80
```

Eso exige que lo que se añade esté probado sin obligar a nadie a subir la cobertura del código heredado, que es lo que hace que la regla se acepte.

Y dos medidas que dicen más que la cobertura y casi nunca se usan:

```
pruebas de mutación   introduce cambios pequeños en el código y comprueba
                      si alguna prueba falla
                      → mide si las pruebas COMPRUEBAN, no si ejecutan
                      → es cara: para el código crítico, no para todo

fallos detectados por nivel  de los defectos que llegaron a producción,
                      ¿qué nivel de prueba debería haberlos detectado?
                      → dice dónde invertir, con datos del propio equipo
```

La segunda es la más barata y la más útil: revisar los últimos veinte defectos de producción y clasificar cuál habría sido el nivel adecuado da un plan de trabajo mejor que cualquier objetivo de cobertura.

Y el otro lado del coste, que se ignora: una prueba tiene mantenimiento. Una que no ha fallado nunca en tres años, que se rompe en cada refactor y que duplica lo que otra cubre es un pasivo. Retirarla es una decisión legítima y hay que poder tomarla:

```
se conserva si    detecta algo que ninguna otra detecta
                  y ese algo importa
se retira si      es un duplicado frágil de un nivel inferior
                  o comprueba una decisión que ya no está vigente
```

4. Puerta o informe: decidirlo explícitamente

La distinción que ordena esta clase:

```
PUERTA    su fallo detiene el cambio
INFORME   su fallo se anota y el cambio continúa
```

Y el error más común es tener informes llamados puertas: comprobaciones que fallan, nadie mira y todo el mundo cree que protegen algo. Es la séptima aparición de la familia de la clase 060 —un mecanismo que parece estar haciendo algo y no lo está—.

La clasificación que funciona:

```
PUERTAS
  compilación y análisis estático
  pruebas unitarias y de integración
  contratos con consumidores existentes
  cobertura del código nuevo
  análisis de seguridad: crítico y corregible (clase 091)
  política sobre el resultado (clase 091)
  ninguna credencial en el cambio (clase 092)

INFORMES
  cobertura global
  deuda técnica y complejidad
  hallazgos de seguridad sin corrección disponible
  estimación de coste, salvo por encima de un umbral (clase 091)
  duración de la canalización
```

Y la secuencia de adopción, que es la quinta aparición en el programa:

```
avisar → bloquear lo que empeora → bloquear siempre
```

Con la misma razón que en las clases 046, 049, 080 y 091: empezar por el bloqueo total con una base incumplidora produce que alguien desactive la puerta, que es la ley 16.

Y dos propiedades que una puerta debe tener para sobrevivir:

```
rápida     por debajo del umbral de la clase 097; si no, se rodea
precisa    un falso positivo frecuente enseña a saltársela (ley 15)
```

Y el mecanismo de excepción, con la disciplina de las clases 046, 067 y 091:

```
motivo, responsable y fecha de caducidad
y la caducidad rompe la canalización
```

Y una puerta que conviene tener y casi nunca está: que el cambio no rompa a nadie más. En un repositorio único es directo; con repositorios separados, es la prueba de contrato del apartado anterior. Sin ella, la única forma de saber que se rompió a alguien es que ese alguien falle.

5. Lo que solo se puede comprobar en producción

Tres clases de comportamiento no se pueden reproducir antes, y conviene reconocerlo en vez de intentar simularlo:

```
volumen y concurrencia reales   un entorno con el 1 % del tráfico no reproduce
                                los problemas del 100 %
variedad de los datos           los datos de prueba son los que alguien imaginó;
                                los reales tienen formas que nadie imaginó
comportamiento de los usuarios  qué caminos recorren de verdad, y en qué orden
```

Y los mecanismos que los cubren, todos posteriores al despliegue:

```
despliegue progresivo con criterio de promoción     clase 102
interruptores para separar desplegar de publicar    clase 105
comprobaciones sintéticas periódicas                aquí
observabilidad con objetivos de servicio            clases 057, 082
```

Las comprobaciones sintéticas merecen su lugar en esta clase porque son pruebas, aunque se ejecuten en producción:

```
un recorrido crítico ejecutado cada minuto contra producción
  → detecta lo que ninguna alerta de infraestructura detecta:
    que el sistema funciona pero el recorrido de compra no
```

Y con dos condiciones que las hacen viables:

```
datos de prueba identificables y limpiables
  → una cuenta sintética, marcada, que no ensucie los informes
y la comprobación no debe producir efectos irreversibles
  → nada de cobros reales; y si el recorrido los tiene, un modo declarado
```

Y una advertencia sobre las pruebas de carga: son útiles y no reproducen producción. Un generador de carga produce peticiones uniformes; los usuarios reales producen ráfagas, sesiones largas y combinaciones raras. Sirven para encontrar un techo y para comparar dos versiones, no para afirmar que el sistema aguantará.

Y la lista de comprobación de la clase:

- la forma de la pirámide justificada por lo que hace el servicio
- cada prueba de extremo a extremo justificada por lo que detecta en exclusiva
- contratos verificados en ambos lados, y retirados si nadie los consume
- dependencias de terceros con doble y comprobación sintética programada
- cobertura usada sobre el cambio, nunca como objetivo global
- revisión periódica de los defectos de producción por nivel que los habría detectado
- cada comprobación clasificada explícitamente en puerta o informe
- puertas rápidas y precisas; excepciones con motivo, responsable y fecha
- pruebas retiradas cuando duplican o comprueban decisiones no vigentes
- recorridos críticos con comprobación sintética contra producción

Y el cierre que enlaza con la clase siguiente: de las siete puertas de la lista, tres son de seguridad y de cadena de suministro. Cómo se integran sin repetir el error de las clases 067 y 091 —una señal con ochocientos elementos que acaba desactivada— es la materia de la clase 101.

Ejemplo trabajado

CloudShop tiene 4.100 pruebas, un 87 % de cobertura y once defectos en producción el último trimestre. El análisis de esos once explica dónde estaba invertido el esfuerzo y dónde estaba el riesgo.

Los once defectos, clasificados por el nivel que los habría detectado.

```
nivel adecuado                          defectos
contrato entre servicios                     5
integración con la base de datos             2
en producción (volumen o datos reales)       3
unitaria                                     1
```

Y la distribución del esfuerzo:

```
nivel                   pruebas    tiempo de ejecución
unitarias                3.640          2 min 10 s
integración                190          4 min 40 s
extremo a extremo          270         11 min 30 s
contrato                     0                  —
```

Cinco de los once defectos eran de un nivel que no existía, y las 270 pruebas de extremo a extremo —el 63 % del tiempo de ejecución— habían detectado uno en todo el trimestre.

Corrección 1 — contratos donde estaba el riesgo.

Los cinco defectos de contrato tenían la misma forma: un servicio cambió su respuesta y el consumidor no se enteró hasta que falló en un entorno compartido.

```
contratos entre servicios                       0              14
verificados en la canalización del proveedor    —          puerta obligatoria
defectos de contrato en el trimestre siguiente  5               0
cambios del proveedor detenidos por romper
  a un consumidor                               —               3
```

Los tres cambios detenidos son la medida del valor: tres cambios incompatibles que se habrían fusionado y habrían roto a alguien.

Corrección 2 — las pruebas de extremo a extremo.

Se revisaron las 270 preguntando qué detecta cada una que no detecte nada más barato:

```
duplican una unitaria o de integración          188
comprueban una decisión ya no vigente            41
detectan algo en exclusiva                       23
intermitentes sin arreglar (clase 097)           18
```

```
pruebas de extremo a extremo                   270              23
tiempo de ejecución                        11 min 30 s       1 min 50 s
defectos únicos detectados por trimestre         1                1
intermitentes                                   18                0
```

Mismo valor detectado, con la décima parte de las pruebas y del tiempo. Y la reducción del tiempo de respuesta total permitió añadir los contratos sin superar el umbral de la clase 097.

Corrección 3 — la cobertura que no significaba nada.

```
$ mutmut run --paths-to-mutate src/precios/
1.240 mutantes · 412 sobrevivieron (33 %)
```

Un tercio de las mutaciones en el módulo de precios no hacía fallar ninguna prueba, con el 91 % de cobertura en ese módulo. Al revisarlas:

```
pruebas que ejecutan sin comprobar nada          61
pruebas que comprueban solo que no hay excepción 88
```

```
cobertura global                              87 %            84 %
objetivo de cobertura global                  80 %         retirado
medida sobre el cambio                     no había     80 % del código nuevo
mutantes supervivientes en precios            33 %             8 %
pruebas sin comprobación retiradas o corregidas  —             149
```

La cobertura global bajó tres puntos y la confianza subió. Fue la conversación más difícil del ejercicio y el argumento que la cerró fue el de las mutaciones: el 33 % de los cambios en el código no rompía ninguna prueba.

Corrección 4 — puertas e informes, clasificados.

```
comprobaciones que decían ser puertas          14               7
de ellas, que de verdad detenían el cambio      7               7
las otras siete                          fallaban y nadie     informes
                                         las miraba           declarados
```

Siete comprobaciones fallaban regularmente sin detener nada y todo el mundo creía que protegían. Séptima aparición de la familia de fallos de la clase 060.

Corrección 5 — lo que solo se ve en producción.

Los tres defectos restantes eran de volumen y de datos reales:

```
un tiempo de espera que solo se agota con más de 400 peticiones por segundo
un carácter en un nombre de cliente que rompía la generación del recibo
```

```
un recorrido de usuario que nadie había previsto
```

```
comprobaciones sintéticas                      0              6 recorridos
frecuencia                                     —            cada minuto
datos sintéticos identificables           no había        cuenta marcada,
                                                          excluida de informes
despliegue progresivo con criterio         no había      clase 102
tiempo medio de detección de un defecto
  de producción                             4 h 20 min       6 min
```

Resumen:

```
pruebas totales                              4.100          3.980
pruebas de extremo a extremo                   270             23
contratos                                        0             14
tiempo total de la suite                  18 min 20 s     8 min 40 s
mutantes supervivientes (módulo crítico)      33 %            8 %
defectos en producción por trimestre            11              3
comprobaciones que dicen ser puertas            14              7
tiempo de detección de un defecto          4 h 20 min       6 min
```

La lección que esta clase traslada al resto de la parte 08: el equipo tenía 4.100 pruebas y cinco de los once defectos del trimestre eran de un nivel que no existía. El 63 % del tiempo de ejecución se iba en 270 pruebas de extremo a extremo que detectaron un defecto único en tres meses, y el 33 % de las mutaciones en el módulo más crítico no rompía ninguna prueba pese al 91 % de cobertura. La cobertura decía lo que se ejecutaba; los defectos decían dónde estaba el riesgo.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/100-pruebas-calidad-y-puertas-de-cambio/lab.py
```

El laboratorio selecciona el motor de práctica testing y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa quality-gates en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es pruebas automatizadas con fallos diagnósticos. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye quality-gates para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

Errores frecuentes

Síntoma	Causa probable	Corrección
Alta cobertura y defectos frecuentes en producción	Las pruebas ejecutan el código sin comprobar nada, porque la cobertura era el objetivo	Usa la cobertura sobre el cambio, no como objetivo global, y mide con pruebas de mutación en el código crítico.
Un cambio de un servicio rompe a otro y se descubre en un entorno compartido	No hay contratos verificados en el lado del proveedor	Genera el contrato desde las pruebas del consumidor y verifícalo como puerta en la canalización del proveedor.
La suite pasa y la integración está rota	El doble se comporta como el autor cree que se comporta la dependencia	Prueba de integración real contra la base de datos y contratos frente a otros servicios; los dobles no verifican creencias.
Comprobaciones que fallan y nadie mira	Son informes que todo el mundo llama puertas	Clasifica cada comprobación explícitamente; una que no detiene el cambio no protege nada.
La suite de extremo a extremo tarda más que todo lo demás y detecta poco	Duplica niveles inferiores y acumula pruebas frágiles	Justifica cada una por lo que detecta en exclusiva; retira las duplicadas y las de decisiones no vigentes.
Hay defectos que ningún entorno previo podía reproducir	Dependen del volumen, de la variedad de datos reales o del comportamiento de los usuarios	Comprobaciones sintéticas contra producción y despliegue progresivo con criterio de promoción.

Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

Preguntas de comprobación

1. ¿Qué distingue una puerta de un informe, y por qué confundirlos es peligroso?
2. ¿Por qué la forma de la pirámide depende de lo que hace el servicio? Da dos ejemplos opuestos.
3. ¿Qué oculta un doble de prueba y cómo se recupera esa confianza sin desplegar dos sistemas juntos?
4. ¿Qué se puede concluir de una cobertura alta y qué de una baja?
5. Enumera tres clases de fallo que ningún entorno previo puede reproducir y el mecanismo que cubre cada una.

Referencias

- Martin Fowler (2012). Test Pyramid — coste y confianza por nivel, y por qué la forma varía. <https://martinfowler.com/bliki/TestPyramid.html>
- Ham Vocke (2018). The practical test pyramid — niveles, dobles y contratos con ejemplos. <https://martinfowler.com/articles/practical-test-pyramid.html>
- Pact (2025). Consumer-driven contract testing — generación del contrato y verificación en el proveedor. <https://docs.pact.io/>
- Martin Fowler (2020). Test coverage — por qué la cobertura como objetivo produce malas pruebas. <https://martinfowler.com/bliki/TestCoverage.html>
- Google (2024). Mutation testing at scale — medir si las pruebas comprueban, y su coste. <https://research.google/pubs/state-of-mutation-testing-at-google/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 08 en PDF · Manual integral

Evaluación — 100 Pruebas, calidad y puertas de cambio

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega quality-gates con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.

101 — SAST, SCA, secretos, SBOM y firma en pipeline

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 4 Laboratorio: security · Estado: EXECUTABLECORE

Propósito

Integrar las cinco familias de comprobación de seguridad en la canalización sin repetir el error que este programa ha documentado tres veces: un escáner con ochocientos hallazgos acaba desactivado, y figura como implantado. La clase introduce la herramienta que lo evita —un presupuesto de ruido con un número concreto— y la técnica que hace posible adoptarlas sobre código heredado: que solo bloquee lo nuevo. La clase 067 dejó los artefactos; aquí se decide dónde se ejecuta cada comprobación, qué detiene y quién arregla.

Resultados de aprendizaje

Al finalizar podrás:

1. Situar cada familia de comprobación por lo que detecta y por dónde debe ejecutarse.
2. Fijar un presupuesto de ruido y ajustar las comprobaciones para no superarlo.
3. Adoptar una comprobación sobre código heredado sin bloquear al equipo.
4. Distinguir una dependencia vulnerable de una vulnerabilidad alcanzable.
5. Enrutar cada hallazgo a quien puede arreglarlo, con plazo.

Conceptos centrales

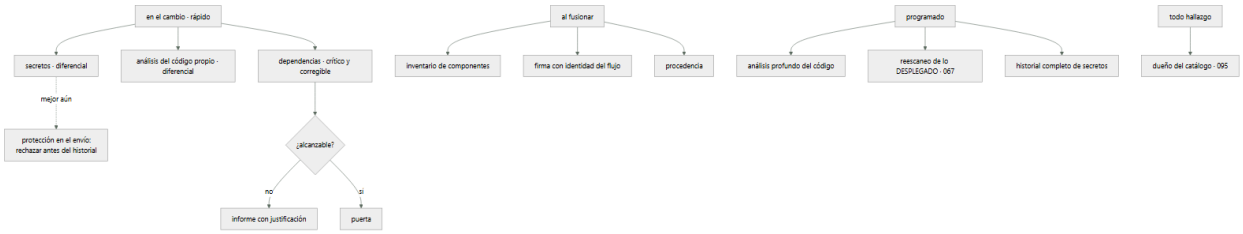
Concepto	Comprensión verificable
presupuesto de ruido	Número máximo de hallazgos que una comprobación puede producir por cambio antes de que la gente deje de leerlos. Es la ley 15 con una cifra.
modo diferencial	La comprobación bloquea solo por hallazgos nuevos respecto de una referencia. Permite adoptar sobre código heredado sin exigir arreglarlo todo primero.
alcanzabilidad	Si el código vulnerable de una dependencia se ejecuta desde la aplicación. Distingue una vulnerabilidad real de una presente pero inerte.
protección en el envío	Rechazar un secreto antes de que entre en el historial. Prevenir es lo único que evita la ley 11: lo que entra en el historial se queda.
firma con identidad del flujo	Firmar con la identidad de la canalización, sin claves. La verificación comprueba qué flujo y qué rama firmaron, no solo que hay firma.
enrutado de hallazgos	Llevar cada hallazgo a quien puede arreglarlo, usando el catálogo de la clase 095. Sin dueño, un hallazgo no se arregla: se acumula.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. Cinco familias, y dónde va cada una

Cada familia detecta una cosa distinta y ninguna sustituye a otra:

```
análisis del código propio   defectos en lo que escribe el equipo
                             inyecciones, criptografía mal usada, rutas inseguras

dependencias                 vulnerabilidades conocidas en lo que se importa
secretos                     credenciales en el código y en el historial
inventario de componentes    qué contiene el artefacto (clase 067)
firma y procedencia          de dónde salió (clases 067, 098)
```

Y la decisión operativa no es cuál usar —hacen falta las cinco— sino dónde se ejecuta cada una, que es lo que decide si sobreviven. El criterio es el de la clase 091: por coste y por lo que puede detectar antes.

```
EN EL CAMBIO, antes de fusionar (rápido, y bloquea)
  secretos, en modo diferencial
  análisis del código propio, solo sobre lo modificado
  dependencias: crítico y corregible (clase 067)

AL FUSIONAR (produce evidencia)
  inventario de componentes, adjunto al artefacto
  firma con la identidad del flujo
  declaración de procedencia

PROGRAMADO (caro, y no bloquea a nadie)
  análisis profundo del código, con todas las reglas
  historial completo de secretos
  reescaneo de lo que está DESPLEGADO
```

La última línea es el paso que la clase 067 numeró como décimo y que casi nadie tiene: una imagen escaneada limpia hace tres meses puede tener hoy una vulnerabilidad crítica publicada la semana pasada. El escaneo en la construcción es una foto; lo que hace falta es reescanear lo que está en ejecución.

```
# huellas realmente desplegadas, y su reescaneo
$ for e in dev pre pro; do kubectl --context=$e get pods -A -o jsonpath='{..imageID}'; done \
  | tr ' ' '\n' | grep -o 'sha256:[0-9a-f]*' | sort -u \
  | while read d; do trivy image --severity CRITICAL --ignore-unfixed -q "registro/…@$d"; done
```

Y una precisión sobre el orden dentro del cambio: los secretos van primero. Es la comprobación más barata y la única cuyo hallazgo exige actuar antes de seguir —rotar—, así que detener el resto de la canalización cuando aparece uno ahorra ejecuciones y evita que la credencial siga viajando.

Y una nota sobre la firma que enlaza con la clase 098: se firma con la identidad del flujo, sin claves, y la verificación comprueba qué flujo y qué rama firmaron. Firmar sin verificar la identidad del firmante es lo que la clase 067 midió: dos imágenes sin firma válida en producción durante dos meses.

2. El presupuesto de ruido

Este programa ha documentado tres veces el mismo final: un escáner con demasiados hallazgos acaba desactivado.

```
escáner de imágenes, 812 hallazgos      desactivado 8 meses      clase 067
analizador de infraestructura, 812      desactivado 14 meses     clase 091
alertas, 340 al mes                     46 silenciadas           clase 057
```

La herramienta que lo evita es fijar un número antes de integrar nada:

```
presupuesto de ruido: 5 hallazgos por cambio, como máximo
```

Y con esa cifra delante, la integración deja de ser «activar la herramienta» y pasa a ser un ejercicio de ajuste con un objetivo medible:

```
si la primera ejecución produce 240 hallazgos por cambio
  → no se activa como puerta
  → se ajusta hasta bajar de 5, y entonces se activa
```

Y los cinco ajustes que bajan el número, en orden de eficacia:

```
1. modo diferencial          solo lo nuevo respecto de la rama principal
2. solo lo corregible        lo que no tiene arreglo no puede ser una puerta
3. solo lo alcanzable        ver el apartado siguiente
4. reglas ajustadas al lenguaje y al marco de trabajo
                             las reglas genéricas producen falsos positivos
5. supresiones en el código, con justificación
                             para el caso concreto que el análisis no entiende
```

El primero es el que más baja y el que hace posible adoptar sobre código heredado:

```
$ semgrep ci --baseline-commit=$(git merge-base HEAD origin/main)
```

Con eso, un repositorio con dos mil hallazgos históricos no bloquea a nadie, y ningún cambio puede añadir uno nuevo. Es la fase 2 de la secuencia de adopción que este programa ha usado ya cinco veces —avisar, no empeorar, bloquear siempre— y aquí es la sexta.

Y la parte incómoda de esa fase: el inventario histórico sigue ahí. Lo que la hace honesta es tratarlo como trabajo planificado y no como algo que se arreglará solo:

```
línea base congelada, con su cifra
un objetivo de reducción por trimestre, con dueño
y la cifra publicada: si sube, la fase 2 no está funcionando
```

Y una comprobación que conviene tener sobre las propias comprobaciones, porque es la que detecta el fracaso antes de que se convierta en un escáner desactivado:

```
hallazgos por cambio, mediana y percentil 90
proporción de ejecuciones en las que alguien suprime o salta la puerta
tiempo añadido a la canalización por cada comprobación
```

La segunda por encima del 10 % significa que el presupuesto está mal fijado, no que el equipo sea indisciplinado.

3. Vulnerable no es alcanzable

Un escáner de dependencias informa de que una versión vulnerable está presente. No informa de que la vulnerabilidad sea explotable en este sistema, y la diferencia es grande:

```
la biblioteca tiene una vulnerabilidad en su analizador de XML
y esta aplicación no analiza XML en ningún punto
→ presente, y no alcanzable
```

Y tratar los dos casos igual produce el problema del apartado anterior. Las tres formas de distinguirlos, de más a menos automática:

```
análisis de alcanzabilidad     la herramienta comprueba si hay un camino de
                               llamadas desde la aplicación al código vulnerable
                               → reduce mucho, y no es infalible

declaración de explotabilidad  un documento verificable que dice, para un
                               componente y una vulnerabilidad, si afecta y por qué
                               → se adjunta al artefacto, como el inventario

lista de exclusiones           con motivo, responsable y caducidad
                               → funciona y no se puede verificar
```

La segunda es la que escala en una organización, porque la evaluación se hace una vez y viaja con el artefacto:

```
{
  "vulnerability": "CVE-2026-1234",
  "product": "registro/tienda@sha256:9f2c…",
  "status": "not_affected",
  "justification": "vulnerable_code_not_in_execute_path",
  "detail": "El analizador de XML no se invoca desde ninguna ruta de la aplicación.",
  "responsable": "equipo-pedidos",
  "revisar": "2026-12-01"
}
```

Y una advertencia honesta sobre la alcanzabilidad automática: puede equivocarse en las dos direcciones. Marca como inalcanzable código que se invoca por reflexión o por configuración, y marca como alcanzable caminos que en la

práctica no se dan. Sirve para priorizar, no para descartar sin mirar.

Y la regla que ordena todo esto:

```
crítico + corregible + alcanzable     PUERTA
crítico + corregible + no alcanzable  informe, con declaración justificada
crítico + sin corrección              informe, y seguimiento con fecha
alto y medio                          informe, y objetivo de reducción
```

Y la palanca que más reduce sin analizar nada, que la clase 067 ya midió: actualizar la imagen base. La mayoría de los hallazgos de un artefacto vienen de ahí, y una cadencia de actualización resuelve en bloque lo que de otro modo se persigue uno a uno.

Y sobre las dependencias directas, dos comprobaciones que valen más que el escáner y cuestan menos:

```
dependencias nuevas en este cambio     ¿alguien la ha pedido? ¿qué aporta?
dependencias sin actualizar en un año  suelen ser las que traen los hallazgos
```

4. Secretos: prevenir en vez de detectar

La ley 11 de la clase 072 es especialmente cruel aquí: lo que entra en el historial se queda, aunque se borre del fichero. Por eso el orden de preferencia es distinto que en las demás familias:

```
1. PREVENIR      rechazar el envío antes de que entre en el historial
2. detectar en el cambio    antes de fusionar, sobre el diferencial
3. detectar en el historial  programado, sobre todo el árbol
```

La primera es la única que evita el trabajo de rotar:

```
protección en el envío: el servidor rechaza el envío si detecta un secreto
  → el secreto nunca llega al historial
  → y quien lo intentó se entera al instante
```

Y conviene medir su efecto, porque es el único mecanismo de esta clase que reduce el trabajo en vez de crearlo:

```
secretos rechazados en el envío       trabajo evitado: ninguna rotación
secretos detectados tras fusionar     rotar, corregir, purgar (clase 092)
```

Y el procedimiento cuando ya ha entrado, que este programa ha repetido cinco veces y no cambia:

```
1. rotar    estuvo expuesto
2. corregir el mecanismo
3. purgar lo que se pueda purgar
```

Y una advertencia sobre la detección: los escáneres encuentran patrones conocidos —claves de proveedores, testigos con formato reconocible— y no encuentran una contraseña que parece una palabra. Complementarlas con reglas propias para los formatos internos de la organización es barato y suele dar resultados el primer día.

Y el rastro que la clase 092 enumeró y que aquí conviene automatizar entero, porque el repositorio es solo uno de seis:

```
repositorio y su historial      escáner, en el cambio y programado
registros de la canalización    comprobación de que el registro detallado
                                está desactivado
artefactos publicados           que no se publiquen ficheros de plan
estado de infraestructura       recuento de campos sensibles
salidas                         nombres que coincidan con patrones
```

Y una comprobación sobre la propia canalización que la clase 098 dejó y que pertenece a esta lista:

```
$ gh secret list --json name -q '.[].name' | grep -Ei 'aws_secret|azure_client_secret|gcp_sa_key' \
  && echo "claves de larga duración: migrar a identidad federada"
```

5. Quién arregla, y en cuánto tiempo

Un hallazgo sin dueño no se arregla: se acumula. Y el catálogo de la clase 095 es lo que permite enrutarlo:

```
$ trivy image --format json registro/tienda@sha256:9f2c… \
  | jq -r '.Results[].Vulnerabilities[]? | select(.Severity=="CRITICAL") | .VulnerabilityID' \
  | while read cve; do
      equipo=$(yq -r '.spec.equipo' catalogo/tienda.yaml)
      echo "$cve → $equipo"
    done
```

Y los plazos, que hay que fijar antes de tener hallazgos y no después:

```
crítico y alcanzable, con corrección       7 días
crítico sin corrección disponible          seguimiento semanal
```

```
alto y alcanzable                       30 días
resto                                   objetivo de reducción trimestral
secreto expuesto                        rotación inmediata; el resto, después
```

Y el mecanismo que hace que los plazos signifiquen algo, con la disciplina de las clases 046, 067 y 091:

```
cada excepción con motivo, responsable y caducidad
y la caducidad rompe la canalización
```

Y una decisión organizativa que decide si esto funciona: quién puede aceptar el riesgo. Un hallazgo crítico sin corregir a los siete días no lo decide el equipo que lo tiene: lo escala a quien pueda decidir que se acepta, con su nombre.

Y la lista de comprobación de la clase:

- presupuesto de ruido fijado, con una cifra, antes de integrar nada
- secretos primero en el orden, y con protección en el envío
- análisis del código propio en modo diferencial sobre el cambio
- dependencias: puerta solo por crítico, corregible y alcanzable
- declaraciones de explotabilidad adjuntas al artefacto, con revisión
- inventario, firma y procedencia al fusionar
- verificación de firma que comprueba el flujo y la rama, no solo la existencia
- análisis profundo y reescaneo de lo desplegado, programados
- hallazgos enrutados al dueño del catálogo, con plazo
- excepciones con motivo, responsable y caducidad que rompe la canalización
- línea base histórica congelada, con objetivo de reducción y cifra publicada
- vigilancia de hallazgos por cambio y de supresiones, para detectar el fracaso

Y el cierre que enlaza con la clase siguiente: todas estas comprobaciones protegen lo que se despliega. Ninguna dice nada sobre cómo llega a producción, y esa es la parte donde un error se convierte en un incidente visible — la materia de la clase 102.

Ejemplo trabajado

CloudShop integra las cinco familias tras el fracaso documentado en las clases 067 y 091 —dos escáneres desactivados durante ocho y catorce meses—. Esta vez se fija el presupuesto de ruido antes de activar nada, y el ejercicio dura seis semanas.

El presupuesto, y la primera medición.

```
presupuesto acordado: 5 hallazgos por cambio

primera ejecución de cada familia, sin ajustar:
  análisis del código propio        240 por cambio
  dependencias                       97
  secretos                            0
  infraestructura (clase 091)         6
```

Con esas cifras, activar como puerta habría reproducido el fracaso anterior. El ajuste, familia por familia:

Análisis del código propio: de 240 a 3.

```
sin ajustar                                          240
modo diferencial sobre el cambio                      11
reglas ajustadas al lenguaje y al marco                4
supresiones justificadas en 2 puntos                   3
```

La línea base histórica quedó congelada en 1.840 hallazgos, con su cifra publicada y un objetivo de reducción de 150 por trimestre. En seis meses bajó a 1.210, y la cifra es visible: si sube, la fase de no empeorar no está funcionando.

Dependencias: de 97 a 2.

```
sin ajustar                                           97
solo crítico                                          31
solo corregible                                       12
solo alcanzable                                        4
tras actualizar la imagen base                         2
```

La última línea confirma lo que la clase 067 midió: la actualización de la base resuelve en bloque. Y de los ocho descartados por alcanzabilidad, se revisaron a mano los ocho:

```
confirmados como no alcanzables                        6
marcados mal por la herramienta                        2   ← invocación por reflexión
```

Los dos errores son el argumento para no descartar sin mirar. Se escribieron declaraciones de explotabilidad para los seis, con responsable y fecha de revisión.

Secretos: cero hallazgos y aun así el mayor cambio.

El escaneo del árbol actual daba cero. El del historial completo, seis (los de la clase 092). Y lo que se añadió fue la prevención:

```
protección en el envío                     no había          activa
secretos rechazados en el envío (6 meses)      —                4
secretos que llegaron al historial             —                0
rotaciones evitadas                            —                4
```

Cuatro credenciales que nunca llegaron al historial, y por tanto cuatro rotaciones que no hubo que hacer. Es el único mecanismo de la clase que reduce trabajo.

Y dos de los cuatro no los habría detectado el escáner genérico: eran testigos de un sistema interno con un formato propio, y se detectaron por una regla escrita el primer día.

Firma y procedencia: la verificación que faltaba.

La clase 067 había dejado la firma activa y la verificación a medias. Al completarla:

```
imágenes firmadas                          15 de 15         15 de 15
verificación comprueba la existencia de firma   sí               sí
verificación comprueba QUÉ flujo firmó          no               sí
verificación comprueba la rama                  no               sí
imágenes rechazadas al activar                  —                2
```

Las dos rechazadas se habían firmado desde una rama de trabajo durante una urgencia, siete meses atrás, y seguían en producción.

El reescaneo de lo desplegado.

```
primera ejecución sobre las 15 huellas en producción
  vulnerabilidades críticas descubiertas                    3
  todas en imágenes que pasaron limpias en su construcción
  la más antigua, publicada hace 41 días
```

Tres vulnerabilidades críticas en producción que ninguna comprobación de la canalización podía detectar, porque las imágenes no habían cambiado: lo que cambió fue lo que se sabe de ellas.

El coste en tiempo de respuesta.

```
comprobaciones en el cambio                    2                5
tiempo añadido                                 —          1 min 20 s
mediana de la canalización                 7 min 10 s       8 min 30 s
umbral de la clase 097                        10 min           10 min
```

Dentro del umbral. Y las dos comprobaciones caras —análisis profundo y reescaneo— fuera del camino crítico.

A los seis meses.

```
escáneres activos                            0 de 5          5 de 5
hallazgos por cambio (mediana)                 —               3
supresiones o saltos de puerta                 —              2,1 %
vulnerabilidades críticas en producción      no se sabía        0
secretos que llegaron al historial            6 en 3 años       0
imágenes en producción sin firma verificada       2             0
línea base histórica                          1.840          1.210
mediana de la canalización                  7 min 10 s      8 min 30 s
```

La lección que esta clase traslada al resto de la parte 08: la diferencia con los dos intentos anteriores no fue de herramientas —son las mismas— sino de haber fijado una cifra antes de activar nada. Con 240 hallazgos por cambio, cualquier equipo desactiva la puerta y tiene razón; con tres, nadie lo intenta. Y el mecanismo que más valió no detecta nada: la protección en el envío evitó cuatro rotaciones al impedir que los secretos llegaran al historial, que es lo único que resuelve de verdad la ley 11.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/101-sast-sca-secretos-sbom-y-firma-en-
```

```
pipeline/lab.py
```

El laboratorio selecciona el motor de práctica security y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa pipeline-devsecops en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es un control con amenaza, mitigación y evidencia. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye pipeline-devsecops para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
Un escáner acaba desactivado y figura como implantado	Produce cientos de hallazgos por cambio y bloquear con esa cifra es inviable	Fija un presupuesto de ruido antes de activar, y ajusta con modo diferencial, corregibilidad y alcanzabilidad hasta bajar de él.
Un repositorio con miles de hallazgos históricos no puede adoptar la comprobación	Se intenta bloquear por todo desde el principio	Modo diferencial: bloquea solo lo nuevo, congela la línea base con su cifra y fija un objetivo de reducción con dueño.
Se bloquean cambios por vulnerabilidades que el sistema no puede ejecutar	Se trata igual una dependencia vulnerable y una vulnerabilidad alcanzable	Analiza alcanzabilidad y adjunta declaraciones de explotabilidad justificadas; revisa a mano, porque la herramienta se equivoca en ambas direcciones.
Un secreto llega al historial y hay que rotarlo	Solo hay detección; la ley 11 dice que lo que entra se queda	Activa protección en el envío y añade reglas para los formatos internos de la organización, que los escáneres genéricos no reconocen.
Las imágenes están firmadas y hay artefactos no autorizados en producción	La verificación comprueba que hay firma, no qué flujo y qué rama firmaron	Acota la identidad del firmante en la política de admisión y verifica con una imagen firmada desde otra rama.
Aparecen vulnerabilidades críticas en imágenes que pasaron limpias	El escaneo de construcción es una foto; lo que cambia es lo que se sabe	Reescanea periódicamente las huellas desplegadas, no solo las que se construyen.

■ Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

■ Preguntas de comprobación

1. ¿Qué detecta cada una de las cinco familias y dónde debe ejecutarse cada una?
2. ¿Qué es un presupuesto de ruido y por qué se fija antes de integrar nada?
3. ¿Qué cinco ajustes bajan el número de hallazgos, y cuál permite adoptar sobre código heredado?
4. ¿Por qué la protección en el envío es cualitativamente distinta de la detección de secretos?
5. ¿Por qué hay que reescanear lo desplegado si la imagen no ha cambiado?

Referencias

• OWASP (2025). DevSecOps guideline: tooling and placement — familias de comprobación y dónde ejecutarlas. <https://owasp.org/www-project-devsecops-guideline/>

• OpenSSF (2025). Scorecard and CI/CD security checks — comprobaciones automatizables sobre la canalización. <https://github.com/ossf/scorecard/blob/main/docs/checks.md>

• CISA (2025). Vulnerability Exploitability eXchange — declarar si una vulnerabilidad afecta y por qué. <https://www.cisa.gov/resources-tools/resources/minimum-requirements-vulnerability-exploitability-exchange-vex>

• GitHub (2025). Push protection for secrets — rechazar el envío antes de que el secreto entre en el historial. <https://docs.github.com/en/code-security/secret-scanning/push-protection-for-repositories-and-organizations>

• Semgrep (2025). Diff-aware scanning — bloquear solo hallazgos nuevos respecto de una referencia. <https://semgrep.dev/docs/deployment/customize-ci-jobs>

• Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 08 en PDF · Manual integral

Anterior	Índice	Siguiente
← 100 · Pruebas, calidad y puertas de cambio	Parte 08 · Programa	102 · Rolling, blue-green, canary y rollback →

Evaluación — 101 SAST, SCA, secretos, SBOM y firma en pipeline

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega pipeline-devsecops con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.

102 — Rolling, blue-green, canary y rollback

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 4 Laboratorio: delivery · Estado: EXECUTABLECORE

Propósito

Elegir cómo llega a producción un artefacto ya verificado, entendiendo lo que una estrategia de despliegue sí controla y lo que no. Ninguna reduce la probabilidad de que el cambio tenga un defecto: reducen cuánta gente lo ve y en cuánto tiempo se revierte. La clase demuestra por qué las cuatro estrategias comparten un mismo requisito —que dos versiones convivan—, por qué un canario de cinco minutos con el 5 % del tráfico no puede detectar casi nada, y por qué la reversión del código es fácil y la de los datos no.

Resultados de aprendizaje

Al finalizar podrás:

1. Distinguir lo que una estrategia de despliegue controla de lo que no.
2. Elegir entre recreación, progresiva, azul-verde y canario con un criterio explícito.
3. Diseñar un cambio para que dos versiones convivan, que es el requisito común de todas.
4. Dimensionar un canario para que su veredicto signifique algo.
5. Separar lo que se puede revertir de lo que no, y planear en consecuencia.

Conceptos centrales

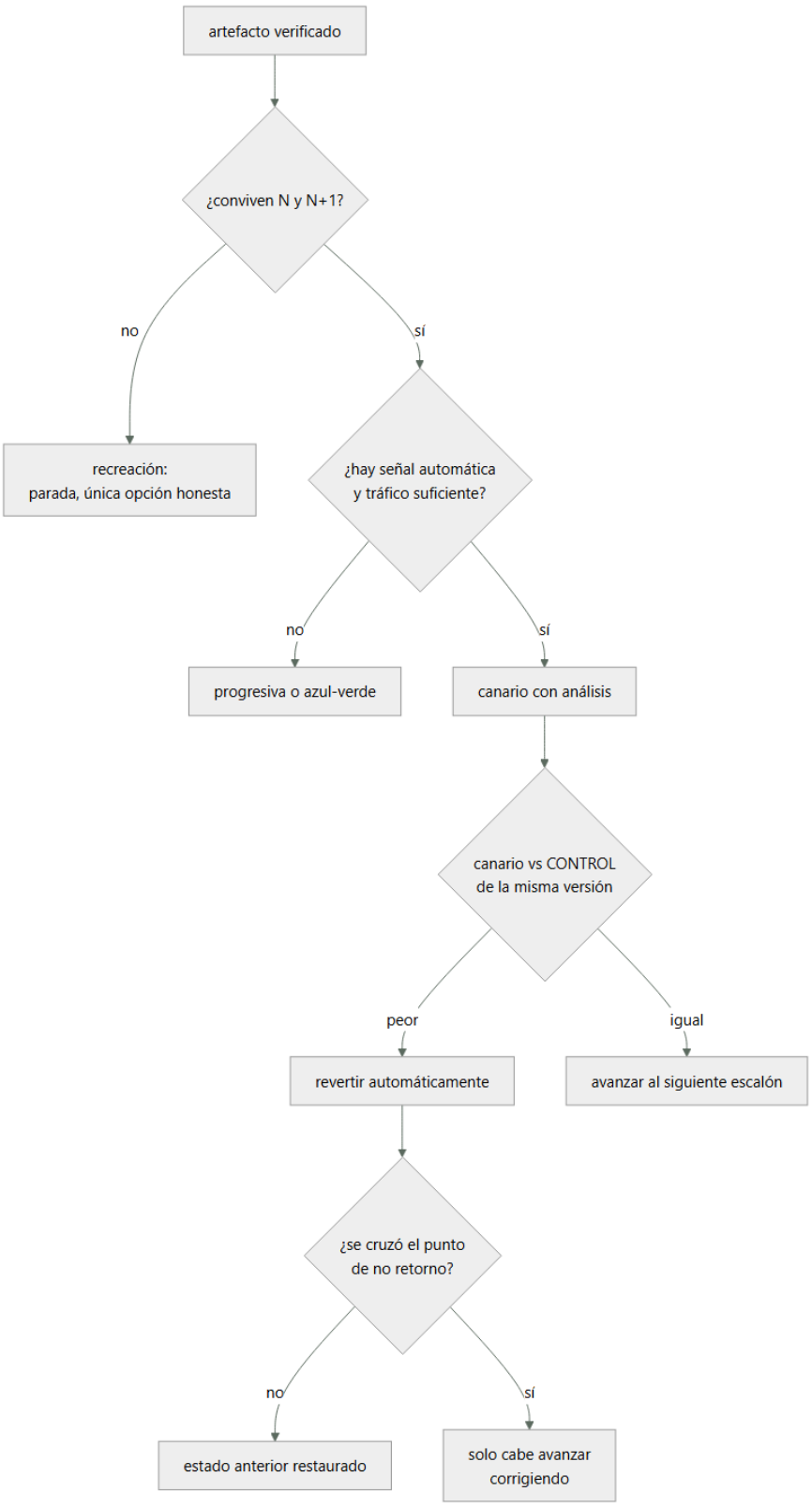
Concepto	Comprensión verificable
radio de exposición	Proporción de usuarios que ve la versión nueva mientras se decide si está bien. Es lo que la estrategia controla de verdad.
convivencia N y N+1	Requisito común a las cuatro estrategias: durante el despliegue hay dos versiones en producción, y todo lo compartido debe funcionar con ambas.
expandir y contraer	Patrón para cambiar un esquema o un contrato en tres pasos —añadir, migrar, retirar— de modo que en ningún momento se rompa la convivencia.
análisis automático del canario	Comparar métricas del canario con las de un control de la misma versión y decidir sin intervención humana. Sin él, un canario es un despliegue progresivo más lento.
potencia del canario	Capacidad estadística de detectar el empeoramiento que se busca. Depende del tráfico que recibe y del tiempo que dura, no de la voluntad de detectarlo.
punto de no retorno	Momento a partir del cual revertir la versión ya no restaura el estado anterior, porque algo irreversible ha ocurrido en los datos.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. Lo que una estrategia controla, y lo que no

Conviene decirlo antes de comparar nada, porque es la confusión más cara de este tema:

```
lo que NO cambia con la estrategia
  la probabilidad de que el cambio tenga un defecto
  → eso lo bajan las pruebas y las puertas de la clase 100
```

```
lo que SÍ cambia
  cuánta gente ve el defecto mientras se decide
  cuánto tarda en dejar de verlo
```

Y con eso, la magnitud que las cuatro estrategias mueven es una sola:

```
impacto ≈ usuarios expuestos × tiempo hasta revertir
```

Las cuatro estrategias, y qué hace cada una con esos dos factores:

```
recreación                100 %         redesplegar   ninguno      sí
progresiva                creciente     redesplegar   ninguno      no
azul-verde                0 o 100 %     cambiar ruta  2× durante   no
canario                   1-5-25 %      cambiar ruta  1 extra      no
```

Y las dos columnas de la derecha explican por qué no hay una respuesta única:

```
azul-verde     reversión en segundos, y hay que pagar dos entornos completos
               salta de 0 % a 100 % de golpe: no hay exposición gradual
canario        exposición gradual, y necesita señal para decidir
               si nadie mira la señal, es una progresiva más lenta
progresiva     barata y sin corte, y revertir exige redesplegar la anterior
recreación     hay corte, y es la única honesta cuando no cabe convivencia
```

Y una combinación frecuente que conviene nombrar porque resuelve la debilidad de dos de ellas: azul-verde con desplazamiento gradual de tráfico —los dos entornos completos, y la ruta moviéndose del 1 % al 100 %— da reversión instantánea y exposición gradual, al precio de mantener los dos entornos.

Y una pregunta que decide antes que cualquier otra, y que la clase 079 dejó planteada en Kubernetes:

```
¿el servicio guarda estado en la instancia?
  sí  → ninguna estrategia funciona bien; el problema es el estado, no el despliegue
  no  → siguen las cuatro sobre la mesa
```

2. El requisito común: que N y N+1 convivan

Progresiva, azul-verde y canario tienen una cosa en común que a menudo se pasa por alto: durante el despliegue hay dos versiones en producción a la vez. Y todo lo que comparten tiene que funcionar con ambas.

```
esquema de la base de datos      lo leen y escriben las dos
mensajes en la cola              los produce una y los consume la otra
caché compartida                 la escribe una y la lee la otra
ficheros en almacenamiento       los escribe una y los lee la otra
contrato de la API               lo llaman clientes de ambas edades
```

Y el fallo típico no es que el cambio esté mal, sino que es incompatible consigo mismo:

```
la versión nueva renombra una columna
→ la versión vieja, que sigue sirviendo el 95 % del tráfico, falla
→ y el canario parece bien, porque el que falla es el control
```

El patrón que lo evita se llama expandir y contraer, y son tres despliegues, no uno:

```
1. EXPANDIR   añadir la columna nueva; la aplicación escribe en las dos
              y sigue leyendo de la vieja
              → compatible con la versión anterior

2. MIGRAR     rellenar la columna nueva con los datos históricos
              cambiar la lectura a la nueva
              → compatible con la versión anterior

3. CONTRAER   dejar de escribir en la vieja y retirarla
              → SOLO cuando ninguna versión anterior está en producción
```

Y la regla que resume el patrón:

```
ninguna migración de esquema puede ser destructiva
mientras exista una versión anterior en ejecución
```

Y lo que hace que el paso 3 se olvide sistemáticamente: es el único que no aporta funcionalidad. Anotarlo como trabajo con fecha, igual que las excepciones de las clases 091 y 101, es lo que evita que la base de datos acumule columnas muertas durante años.

Y el mismo patrón vale para las colas y para las API:

```
colas   el consumidor nuevo entiende el formato viejo y el nuevo
        se despliegan primero TODOS los consumidores, y luego el productor
```

```
API     añadir campos, no renombrar; los campos nuevos, opcionales
        retirar solo cuando ningún cliente los usa, y eso se mide
```

La segunda línea de las colas es la que ordena el despliegue cuando hay dos servicios: primero el que lee, después el que escribe. Al revés, hay mensajes que nadie sabe interpretar.

3. Un canario que no puede detectar nada

Este es el apartado que cambia la práctica de la mayoría de los equipos. Un canario decide comparando métricas, y para que la comparación signifique algo hacen falta suficientes eventos.

La cuenta, con números realistas:

```
tráfico total                    1.000 peticiones/s
al canario                         5 %  →  50 peticiones/s
tasa de error de referencia       0,1 %  →  0,05 errores/s
duración del canario              5 min

errores esperados en el canario, si todo va bien:      15
errores esperados si el cambio DUPLICA la tasa:        30
```

Y con 15 frente a 30 eventos, la variación normal de un sistema real hace que esa diferencia sea perfectamente compatible con el azar. Un canario de cinco minutos al 5 % no puede detectar que la tasa de error se ha duplicado. Lo que hace es dar permiso para avanzar.

Las tres palancas, y cuál conviene mover:

```
más tráfico al canario    detecta antes, y expone a más gente
más duración              detecta lo mismo, exponiendo a los mismos
umbral menos exigente     detecta solo desastres, no degradaciones
```

Y la conclusión práctica: el escalón inicial debe durar más de lo que la intuición sugiere, y los siguientes pueden ser rápidos porque el tráfico ya es mayor.

```
5 %   30 min     ← el escalón que de verdad decide
25 %  10 min
50 %   5 min
100 %  —
```

Y una segunda corrección, tan importante como la anterior: contra qué se compara.

```
mal   canario de hoy contra la media de ayer
      → la carga, la hora y la composición del tráfico son distintas

bien  canario contra un CONTROL de la versión anterior
      desplegado a la vez, con el mismo tamaño y el mismo tipo de tráfico
      → la única diferencia entre los dos grupos es la versión
```

Y qué se mira, en orden de utilidad:

```
tasa de error del propio servicio         siempre
latencia, percentil 95 y 99               siempre
errores de sus dependencias               casi siempre
una métrica de negocio del recorrido      cuando exista
CPU y memoria                             como apoyo, no como veredicto
```

Y la trampa que la ley 13 vuelve a poner aquí: si el análisis no consulta ninguna métrica, no falla — aprueba. Un canario mal configurado no da error: da luz verde. La comprobación que lo detecta es la de siempre:

```
desplegar a propósito una versión rota y confirmar que el canario la para
y repetirlo cada cierto tiempo, porque las métricas se renombran
```

4. Revertir: lo que se puede y lo que no

La reversión del código es un problema resuelto: se vuelve a la etiqueta anterior, que sigue existiendo porque la clase 099 la hizo inmutable. Lo que no está resuelto es todo lo demás.

```
se revierte bien
  el binario y la imagen
  la configuración versionada
  la ruta del tráfico

no se revierte
  los datos que la versión nueva escribió en un formato nuevo
  los mensajes que ya consumió
```

```
  los correos que envió
  los pagos que cobró
  las llamadas que hizo a terceros
```

La frontera entre las dos listas es el punto de no retorno, y conviene identificarlo por adelantado para cada cambio:

```
¿este cambio escribe datos que la versión anterior no sabe leer?
¿consume mensajes que no podrá reprocesar?
¿tiene efectos hacia fuera que no se pueden deshacer?

tres noes  → revertir es seguro; automatízalo
algún sí   → revertir no basta; hay que planear la corrección hacia delante
```

Y cuando la respuesta es «algún sí», la estrategia cambia de forma:

```
reducir el radio todavía más (1 %, no 5 %)
poner un interruptor de funcionalidad para apagar sin desplegar (clase 105)
y tener escrito el procedimiento de corrección, no solo el de reversión
```

Y los dos umbrales que hacen que la reversión ocurra de verdad:

```
automática   el análisis del canario revierte sin preguntar
manual       cualquiera del equipo puede revertir sin pedir permiso
```

La segunda es organizativa y suele ser el cuello de botella real: si revertir exige una aprobación, la mediana de tiempo hasta revertir se mide en horas. Y con la fórmula del primer apartado, eso multiplica el impacto por el mismo factor.

Y una precaución sobre la reversión automática, porque el exceso también hace daño: si revierte por cualquier oscilación, el equipo pierde la confianza y la desactiva —ley 16 otra vez—. El ajuste es el mismo que en la clase 101: medir cuántas reversiones fueron correctas.

```
reversiones automáticas por trimestre         12
  correctas (el cambio era malo)               9
  innecesarias (falsa alarma)                  3   ← si sube, sube el umbral
```

Y la lista de comprobación de la clase:

- [ ] el servicio no guarda estado en la instancia
- [ ] el cambio es compatible con la versión anterior en esquema, colas y API
- [ ] las migraciones destructivas están separadas, con fecha y dueño
- [ ] los consumidores se despliegan antes que los productores
- [ ] el canario se compara con un control de la versión anterior, no con ayer
- [ ] el primer escalón dura lo suficiente para que su veredicto signifique algo
- [ ] el análisis consulta métricas que existen, comprobado con una versión rota
- [ ] el punto de no retorno del cambio está identificado antes de desplegar
- [ ] revertir no requiere aprobación de nadie
- [ ] se mide cuántas reversiones automáticas fueron innecesarias

Y el cierre que enlaza con la clase siguiente: en todo esto, alguien o algo ejecuta el despliegue. Cuando ese algo es un proceso que compara continuamente lo declarado con lo real —la predicción escrita al cerrar la parte 06— aparecen problemas nuevos, y son la materia de la clase 103.

Ejemplo trabajado

CloudShop tenía despliegue progresivo en los quince servicios y una reversión que exigía aprobación. En seis meses hubo cuatro incidentes causados por despliegues; el ejercicio consiste en clasificarlos y cambiar solo lo que los cuatro casos justifican.

Los cuatro incidentes.

```
A  fallo del 100 % de peticiones           detectado en 4 min, revertido en 51 min
   causa: excepción en el arranque con la configuración de producción

B  latencia ×3 en el percentil 99          detectado en 3 h, revertido en 4 h 10
   causa: una consulta sin índice, visible solo con datos reales

C  pedidos duplicados durante 2 h 40       no se revirtió
   causa: la versión nueva reprocesó mensajes que la vieja había marcado

D  el 5 % de peticiones fallando 6 días    detectado por un cliente
   causa: la versión nueva renombró un campo; el consumidor viejo lo ignoraba
```

Lo primero que salta: el tiempo hasta revertir, no la detección.

```
A                  4 min           51 min                    43 min
```

```
B                    3 h           1 h 10                      38 min
C                     —              —                          —
D                   6 días           —                          —
```

En los dos casos donde se revirtió, la aprobación fue la mayor parte del tiempo. Se eliminó el requisito: cualquiera del equipo revierte sin pedir permiso, y se avisa después.

```
mediana de tiempo hasta revertir         1 h 10           6 min
```

El incidente A: un canario lo habría parado, y no había canario.

Un fallo del 100 % de peticiones es lo más fácil de detectar: con el 5 % del tráfico durante 30 minutos, un canario lo habría parado en el primer minuto exponiendo al 5 %. Se activó canario con análisis automático en los cinco servicios de cara al cliente.

Y la primera configuración fue la que casi todo el mundo escribe:

```
5 % durante 5 min, comparado con la media de la semana anterior
```

Se hizo la cuenta del apartado tercero con el tráfico real del servicio de pedidos:

```
tráfico                          420 peticiones/s
al canario, 5 %                   21 peticiones/s
tasa de error de referencia          0,08 %
errores esperados en 5 min             5
errores si la tasa se duplica         10
```

Cinco frente a diez. Se cambió a 30 minutos en el primer escalón y a comparar con un control de la versión anterior desplegado a la vez. Y se comprobó desplegando a propósito una versión con un 2 % de error inyectado:

```
5 min, contra la semana anterior             no
30 min, contra la semana anterior            no consistentemente
30 min, contra control simultáneo            sí, en el minuto 11
```

La tercera línea es la que justifica el control simultáneo: sin él, la variación entre días tapaba la señal.

El incidente B: el canario tampoco lo habría parado, y eso hay que decirlo.

La consulta sin índice degradaba solo con el volumen de datos completo, y a bajo tráfico el percentil 99 del canario no se separaba del control. Lo que lo detectó al reproducirlo fue una métrica distinta:

```
latencia del servicio, p99          sin diferencia significativa a 5 %
duración de consulta a la base      +310 % desde el primer minuto
```

Se añadió la latencia de las dependencias al análisis. El canario no es una red universal: cada incidente que se escapa señala una métrica que falta.

El incidente C: no había estrategia posible, porque el punto de no retorno se cruzó en el minuto uno.

Los pedidos duplicados ya estaban cobrados. Revertir la versión no los deshacía. La revisión del cambio con las tres preguntas habría dado dos síes:

```
¿escribe datos que la versión anterior no sabe leer?    no
¿consume mensajes que no podrá reprocesar?              SÍ
¿tiene efectos hacia fuera irreversibles?               SÍ  (cobros)
```

Se añadió esa revisión de tres preguntas a la plantilla de cambio. Con dos síes, el despliegue pasa a 1 % durante una hora y con interruptor de funcionalidad.

El incidente D: el problema no era el despliegue, era la convivencia.

El campo renombrado rompió al consumidor viejo. El diagnóstico es el del apartado segundo, y la corrección también:

```
cambios de esquema y contrato       en un despliegue   expandir/migrar/contraer
orden de despliegue                 sin regla          consumidores primero
comprobación de compatibilidad      no había           puerta en la canalización
```

La puerta compara el contrato del cambio con el de la versión en producción y bloquea si elimina o renombra algo. En seis meses paró once cambios; los once eran renombrados en despliegue único, exactamente el patrón del incidente D.

Y el paso de contraer, que es el que se olvida:

```
columnas duplicadas pendientes de retirar, al empezar        0
creadas por el patrón en 6 meses                             9
retiradas en plazo                                           7
```

```
pendientes con fecha y dueño                                  2
```

A los seis meses.

```
incidentes causados por despliegues        4 / 6 meses    1 / 6 meses
mediana de tiempo hasta revertir              1 h 10          6 min
usuarios expuestos en el peor caso             100 %           5 %
reversión requiere aprobación                    sí            no
reversiones automáticas del canario               —         9 correctas
                                                            3 innecesarias
cambios parados por incompatibilidad              —            11
prueba de canario con versión rota          no existía      trimestral
```

La lección que esta clase traslada al resto de la parte 08: de las cuatro causas, solo una se resolvía con la estrategia de despliegue. La segunda pedía una métrica que no se miraba, la tercera era irreversible por naturaleza y la cuarta era un problema de compatibilidad entre versiones que ninguna estrategia arregla. Y el cambio de mayor efecto medido no fue el canario: fue quitar la aprobación para revertir, que dividió por once el tiempo de exposición sin costar nada.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/102-rolling-blue-green-canary-y-rollback/lab.py
```

El laboratorio selecciona el motor de práctica delivery y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa estrategia-despliegue en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es un pipeline con gates, promoción y rollback. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye estrategia-despliegue para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

Errores frecuentes

Síntoma	Causa probable	Corrección
El canario aprueba cambios que luego fallan en producción	Con el tráfico y la duración configurados no hay eventos suficientes para detectar el empeoramiento	Calcula los eventos esperados en el escalón; alarga el primer escalón hasta que la diferencia buscada sea distinguible.
El canario da veredictos erráticos según el día y la hora	Se compara con datos históricos, y la carga y el tráfico no son comparables	Compara con un control de la versión anterior desplegado a la vez, con el mismo tamaño y el mismo tipo de tráfico.
La versión antigua empieza a fallar cuando se despliega la nueva	El cambio es incompatible con su propia versión anterior en esquema, cola o contrato	Aplica expandir y contraer, despliega los consumidores antes que los productores y pon una puerta que bloquee eliminaciones y renombrados.
Se revierte la versión y el problema persiste	Se cruzó el punto de no retorno: hay datos, mensajes o efectos externos que la reversión no deshace	Identifica el punto de no retorno antes de desplegar con las tres preguntas y planea la corrección hacia delante, no solo la reversión.
El tiempo hasta revertir se mide en horas aunque la detección sea de minutos	Revertir requiere una aprobación	Autoriza a cualquiera del equipo a revertir sin permiso previo, con aviso posterior.
Un canario mal configurado aprueba todo y nadie se entera	Ley 13: un análisis que no consulta ninguna métrica no falla, aprueba	Despliega a propósito una versión rota cada trimestre y confirma que el canario la para.

Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

Preguntas de comprobación

1. ¿Qué controla una estrategia de despliegue y qué no controla?
2. ¿Qué requisito comparten progresiva, azul-verde y canario, y qué patrón lo satisface?
3. ¿Por qué un canario de cinco minutos con el 5 % del tráfico puede no detectar nada?
4. ¿Contra qué debe compararse el canario y por qué no contra la semana anterior?
5. ¿Qué tres preguntas identifican el punto de no retorno de un cambio?

Referencias

- Humble, J. y Farley, D. (2010). Continuous Delivery, cap. 10 — despliegues azul-verde, canario y reversión. <https://www.oreilly.com/library/view/continuous-delivery-reliable/9780321670250/>
- Google SRE (2025). Canarying releases — control simultáneo, potencia del análisis y automatización del veredicto. <https://sre.google/workbook/canarying-releases/>
- Kubernetes (2025). Deployment strategies and rollbacks — mecánica de progresiva y reversión. <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>
- Argo Rollouts (2025). Analysis and progressive delivery — escalones, métricas y reversión automática. <https://argo-rollouts.readthedocs.io/en/stable/features/analysis/>
- Fowler, M. (2025). Parallel change (expand and contract) — cambiar un contrato sin romper la convivencia. <https://martinfowler.com/bliki/ParallelChange.html>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 08 en PDF · Manual integral

Evaluación — 102 Rolling, blue-green, canary y rollback

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega estrategia-despliegue con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.

103 — GitOps con Argo CD o Flux

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 4 Laboratorio: gitops · Estado: EXECUTABLECORE

Propósito

Poner en marcha el bucle continuo que la parte 06 predijo y que la parte 07 no llegó a construir: un agente dentro del entorno que compara sin parar lo declarado con lo real y corrige la diferencia. La clase muestra lo que ese bucle resuelve —la canalización deja de tener credenciales del clúster, y la deriva deja de acumularse— y, con el mismo detalle, lo que rompe si se activa sin decidir antes qué campos no le pertenecen.

Resultados de aprendizaje

Al finalizar podrás:

1. Explicar en qué cambia el modelo cuando el agente tira en vez de que la canalización empuje.
2. Organizar los repositorios de código y de entorno, y qué confirmación despliega.
3. Delimitar qué campos gobierna el bucle y cuáles debe ignorar.
4. Promocionar un cambio entre entornos con evidencia y sin reconstruir el artefacto.
5. Reconocer los modos de fallo propios del bucle, empezando por el silencioso.

Conceptos centrales

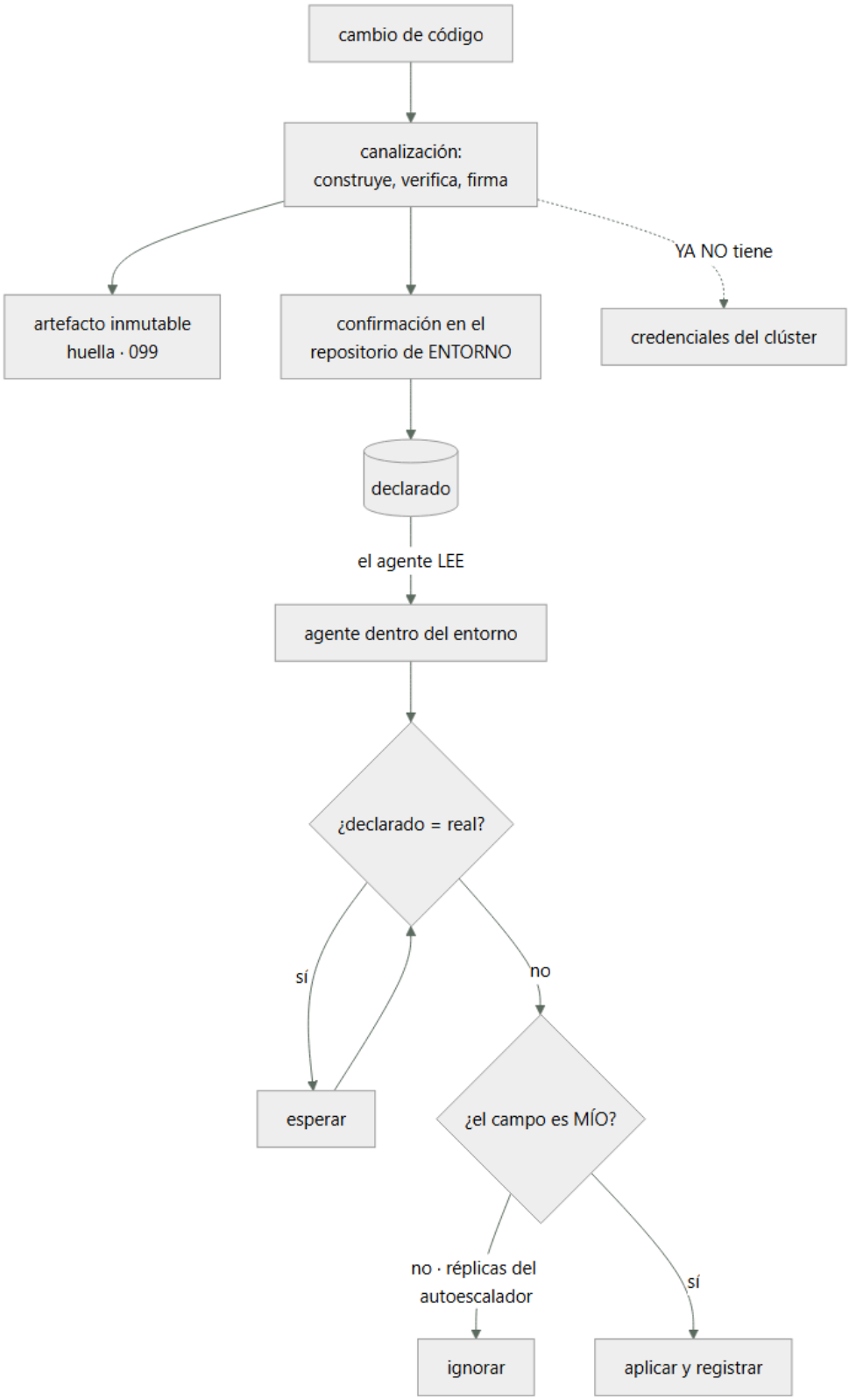
Concepto	Comprensión verificable
reconciliación continua	Un agente compara periódicamente el estado declarado con el real y aplica la diferencia. No es un despliegue: es un bucle que no termina.
modelo de tirar	El agente vive dentro del entorno y lee el repositorio. La canalización deja de necesitar credenciales del clúster.
repositorio de entorno	Repositorio separado que declara qué versión corre en cada entorno. La confirmación que lo modifica es la que despliega.
propiedad de campo	Decisión explícita sobre qué partes de un recurso gobierna el bucle. Sin ella, el bucle pelea con los autoescaladores y los operadores.
poda	Borrar lo que existe en el entorno y no está declarado. Es lo que cierra el círculo de la ley 13, y también lo más peligroso del bucle.
ondas de sincronización	Orden explícito de aplicación cuando unos recursos dependen de otros. Sin ellas el bucle converge igual, pero con errores transitorios ruidosos.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. Qué cambia cuando el agente tira

Hasta aquí, la canalización se conectaba al entorno y aplicaba. Eso obliga a que la canalización tenga credenciales con permiso para modificar producción, y la clase 098 ya explicó lo que eso significa: quien controla el flujo controla el entorno.

El cambio del modelo:

```
EMPUJAR                              TIRAR
canalización → entorno               entorno → repositorio
necesita credenciales del clúster    no las necesita
despliega cuando se ejecuta          reconcilia sin parar
la deriva se acumula                 la deriva se corrige
si el flujo no corre, no pasa nada   si el bucle no corre, tampoco (ley 13)
```

Y las tres consecuencias que importan, en orden de valor:

```
1. la canalización pierde el permiso más peligroso que tenía
   → deja de ser el objetivo que la clase 098 describió

2. la deriva deja de acumularse
   → la clase 090 la detectaba una vez al día; aquí se corrige sola

3. el estado del entorno es legible sin entrar en él
   → lo declarado está en un repositorio, con historial y revisión
```

Y una consecuencia menos citada y muy útil en la práctica: reconstruir un entorno entero pasa a ser apuntar un agente nuevo al mismo repositorio. Es la recuperación que la clase 088 pedía, sin procedimiento aparte.

Y lo que no cambia, para no venderlo de más:

```
no reduce los defectos del cambio        eso siguen siendo las pruebas
no sustituye la estrategia de la 102     el bucle aplica; el escalonado decide
no resuelve los secretos                 ver el apartado cuarto
no impide que alguien modifique a mano   lo detecta y lo revierte, que no es lo mismo
```

La última línea es importante: el bucle no bloquea el cambio manual. Lo deshace unos minutos después. Y eso es una mejora enorme frente a que se quede, y a la vez un problema nuevo, que es el del apartado tercero.

2. Dos repositorios, y qué confirmación despliega

La separación que hace funcionar esto es sencilla de enunciar y se equivoca a menudo:

```
repositorio de CÓDIGO      lo que se construye
                           su confirmación produce un artefacto, no un despliegue

repositorio de ENTORNO     qué versión corre en cada entorno
                           su confirmación SÍ despliega
```

Y con esa separación, promocionar entre entornos deja de reconstruir nada —la regla de la clase 099— y pasa a ser un cambio de una línea:

```
entornos/pre/tienda.yaml:  image: registro/tienda@sha256:9f2c…
entornos/pro/tienda.yaml:  image: registro/tienda@sha256:9f2c…   ← misma huella
```

Y el detalle que evita el fallo más común: por huella, no por etiqueta móvil. Una etiqueta como v2.3 puede reapuntarse; la huella no. Si el repositorio de entorno declara una etiqueta móvil, el bucle no puede saber si hay deriva, porque lo declarado no identifica un artefacto.

La estructura que sostiene esto sin duplicar todo tres veces es la de la clase 083:

```
base/                    lo común
entornos/dev/            lo que cambia en dev
entornos/pre/
entornos/pro/
```

Y una decisión organizativa que decide la fricción diaria: quién aprueba cada entorno.

```
dev   confirmación directa, sin revisión
pre   automática al pasar las puertas de la clase 100
pro   revisión de una persona del equipo dueño (catálogo, clase 095)
```

Y el registro que esto produce sin esfuerzo adicional, que es lo que la clase 090 tenía que reconstruir a mano:

```
$ git log --oneline entornos/pro/tienda.yaml
a91c4e2  promociona tienda a sha256:9f2c…   (aprobó: equipo-pedidos)
7b30f18  revierte a sha256:41ab…           (incidente INC-2291)
```

Y una precaución sobre las ondas de sincronización: cuando unos recursos dependen de otros —espacio de nombres antes que lo que contiene, definición de recurso antes que su instancia—, el bucle acaba convergiendo igual a base de reintentos, pero produce errores transitorios que se confunden con fallos reales. Declarar el orden es barato y limpia el

ruido.

3. Qué campos NO son del bucle

Este es el apartado que la clase 096 anticipó al escribir la lista de lo que el bucle no debe revertir, y es donde fallan casi todas las adopciones.

El bucle compara lo declarado con lo real. Pero en un entorno vivo hay cosas reales que nadie declaró y que están bien:

```
número de réplicas          lo fija el autoescalador (clase 078)
                            declarar 3 y que el bucle lo devuelva a 3
                            cada dos minutos anula el autoescalado

etiquetas y anotaciones     las añaden operadores y mallas de servicio
inyectadas

certificados renovados      los rota un operador

campos rellenados por       direcciones asignadas, identificadores generados
el propio sistema
```

Y el síntoma cuando esto no se decide es característico: el bucle informa de deriva permanente y nadie la mira, que es la ley 15 otra vez —una señal que siempre está encendida deja de ser señal—.

La corrección es declarar la propiedad de cada campo:

```
# el bucle gobierna todo el recurso MENOS las réplicas
ignoreDifferences:
  - group: apps
    kind: Deployment
    jsonPointers: ["/spec/replicas"]
```

Y la regla general que ordena la decisión:

```
si otro controlador escribe ese campo por diseño → no es del bucle
si lo escribió una persona a mano                → sí es del bucle
```

Y la segunda línea es la que da valor: la corrección automática de lo que alguien tocó a mano es exactamente lo que la clase 090 no podía hacer.

La poda, que es lo que cierra el círculo. Sin ella, borrar algo del repositorio no lo borra del entorno, y la ley 13 vuelve: un recurso que se quitó de lo declarado sigue corriendo y nadie da error.

```
sin poda   el entorno acumula lo que ya nadie declara
con poda   lo declarado es la verdad completa
```

Y a la vez es lo más peligroso del bucle, porque un error en el repositorio se convierte en un borrado:

```
un fichero mal movido en una confirmación
→ el bucle interpreta que esos recursos ya no se declaran
→ y los borra
```

Las tres protecciones, y conviene tener las tres:

```
finalizadores y anotación de no-podar en lo que no debe borrarse nunca
  → bases de datos, volúmenes persistentes, espacios de nombres
umbral de seguridad: si la diferencia supera un porcentaje, parar y avisar
poda manual en producción durante la adopción, automática después
```

Y el modo de fallo silencioso que hay que vigilar desde el primer día, porque es el de siempre:

```
el agente deja de sincronizar —permisos caducados, repositorio inaccesible—
→ no da error: deja de haber cambios
→ y el entorno parece estable cuando en realidad está congelado
```

La comprobación que lo detecta es la de la clase 090: alertar por antigüedad de la última reconciliación correcta, no por fallo.

```
time() - max(argocd_app_reconcile_timestamp) > 1800
```

4. Secretos, incidentes y lo que el bucle no debe deshacer

Dos problemas prácticos que deciden si esto sobrevive al primer mes.

Secretos. Si lo declarado vive en un repositorio, un secreto en lo declarado es un secreto en el repositorio, y la ley 11 dice lo que pasa entonces. Las dos salidas honestas:

```
cifrado en el repositorio      el fichero está cifrado; el agente descifra
                               con una clave del entorno
                               → simple, y rotar exige recifrar

referencia externa             el repositorio declara DÓNDE está el secreto,
                               y un operador lo trae del almacén (clases 046, 054)
                               → el secreto nunca entra en el repositorio
```

La segunda es la que encaja con todo lo anterior de este programa, porque conserva la propiedad de la clase 054: el secreto se resuelve con la identidad de la carga, no con una clave guardada.

```
# lo que SÍ entra en el repositorio
spec:
  secretStoreRef: { name: almacen-pro }
  data:
    - secretKey: cadena-conexion
      remoteRef: { key: pedidos/base-datos }
```

Incidentes. Aquí está la tensión que la clase 096 previó. Durante un incidente alguien cambia algo a mano para parar la hemorragia, y el bucle lo revierte a los dos minutos.

La respuesta mala y la buena:

```
mala   desactivar el bucle durante el incidente
       → y olvidarse de volver a activarlo (ley 13 y ley 16 a la vez)

buena  que el cambio de emergencia sea una confirmación
       → con revisión posterior en vez de previa, y aviso automático
```

Y para que la buena sea viable, el camino de emergencia tiene que ser más rápido que tocar a mano, no más lento; si no, nadie lo usa. Eso significa: rama directa, sin aprobación, sincronización inmediata y un aviso al canal del equipo.

Y conviene medir si funciona:

```
cambios de emergencia por confirmación        11
cambios a mano en el entorno                   2   ← si sube, el camino es lento
bucles desactivados y no reactivados           0   ← si no es cero, hay un problema
```

Y la lista de comprobación de la clase:

- la canalización ya no tiene credenciales de escritura en el entorno
- repositorio de entorno separado, y su confirmación es la que despliega
- se declara la huella del artefacto, no una etiqueta móvil
- los campos de otros controladores están excluidos del bucle
- la poda está activa, con protecciones sobre lo que no debe borrarse
- hay umbral de seguridad que para la poda si la diferencia es grande
- alerta por ANTIGÜEDAD de la última reconciliación, no por fallo
- los secretos se resuelven por referencia, no se guardan en el repositorio
- el camino de emergencia es una confirmación y es más rápido que tocar a mano
- se mide cuántos cambios manuales quedan y cuántos bucles se desactivaron

Y el cierre que enlaza con la clase siguiente: con lo declarado en un repositorio y un agente que lo materializa, crear un entorno completo deja de ser un proyecto y pasa a ser una confirmación. Qué se puede hacer con eso —y qué cuesta— es la materia de la clase 104.

Ejemplo trabajado

CloudShop lleva los quince servicios al modelo de tirar. La parte 06 predijo que un bucle continuo resolvería la deriva; la parte 07 solo construyó detección programada. Este ejercicio es la primera vez que el bucle existe, y lo interesante son los tres problemas que aparecen y que la detección no tenía.

Punto de partida.

```
despliegue           la canalización se conecta al clúster y aplica
credenciales         un rol con permiso de escritura en los tres entornos,
                     guardado en la canalización
deriva               detectada una vez al día (clase 090); 2 cambios manuales/mes
reconstruir entorno  procedimiento de 4 páginas, probado una vez
```

Semana 1: el bucle se activa en dev y aparece deriva permanente.

```
recursos declarados                       412
marcados como derivados en la primera hora 209
```

Doscientos nueve de cuatrocientos doce. Al mirarlos, ninguno era un cambio manual:

```
réplicas fijadas por el autoescalador            61
anotaciones inyectadas por la malla de servicio  88
campos rellenados por el propio sistema          47
cambios manuales reales                          13
```

Con el 51 % de los recursos siempre en rojo, la señal no sirve —ley 15, séptima aparición—. Se declaró la propiedad de campo:

```
sin excluir nada                                 209
excluyendo réplicas                              148
excluyendo anotaciones inyectadas                 60
excluyendo campos del sistema                     13
```

Trece, y los trece eran cambios manuales reales. El bucle los corrigió, y de los trece hubo uno que no debía corregirse: un límite de memoria subido a mano durante un incidente tres semanas antes, que nadie había llevado al repositorio. Se llevó al repositorio antes de que el bucle lo revirtiera.

Semana 3: la poda borra un espacio de nombres en preproducción.

Una confirmación movió un directorio y el bucle interpretó que sesenta recursos ya no se declaraban.

```
recursos borrados               60
tiempo hasta detectarlo          4 min
tiempo hasta restaurarlo        11 min   ← revirtiendo la confirmación
datos perdidos                   0
```

Los once minutos de restauración son, en realidad, el argumento a favor del modelo: se restauró revirtiendo una confirmación, sin procedimiento. Pero podía haber pasado en producción, y ahí sí había volúmenes. Se añadieron las tres protecciones:

```
umbral de seguridad: la poda se detiene si afecta a más del 10 % de recursos
anotación de no-podar en volúmenes, bases de datos y espacios de nombres
poda manual en producción durante los tres primeros meses
```

El umbral se activó dos veces en seis meses; las dos eran errores de confirmación, no cambios intencionados.

Semana 5: el bucle de un servicio lleva nueve días sin sincronizar y nadie lo sabe.

El testigo de acceso al repositorio había caducado. El estado del servicio no era «error»: era «desconocido», y el panel lo pintaba en gris.

```
días sin reconciliar              9
confirmaciones sin aplicar        6
qué mostraba el panel        sincronizado (con la última información conocida)
```

Es la ley 13 por séptima vez en este programa, y en su forma más pura: el bucle que no corre no da error. La alerta que lo detecta no mira fallos, mira antigüedad:

```
time() - max by (app) (argocd_app_reconcile_timestamp) > 1800
```

Desde entonces, tres detecciones más: dos permisos caducados y un repositorio renombrado.

Semana 8: el primer incidente con el bucle activo.

Alguien subió las réplicas a mano para absorber un pico. El bucle no las tocó —estaban excluidas— pero el cambio siguiente sí fue a mano: un límite de memoria. El bucle lo revirtió en dos minutos, en mitad del incidente.

```
reacción inicial propuesta   desactivar el bucle durante incidentes
decisión tomada              camino de emergencia por confirmación
```

Y el camino se diseñó con el criterio del apartado cuarto —más rápido que tocar a mano—:

```
tocar a mano en el clúster                       40 s
confirmación de emergencia (sin aprobación,
sincronización inmediata, aviso al canal)        95 s
```

Noventa y cinco segundos frente a cuarenta. Se aceptó, y se midió el uso:

```
cambios de emergencia por confirmación       4       11
cambios a mano en el clúster                 3        0
bucles desactivados por un incidente         1        0
```

El único bucle desactivado, el del mes 1, estuvo apagado seis días después del incidente.

La canalización pierde el permiso.

```
credenciales de escritura en el clúster    en la canalización   ninguna
lo que la canalización puede hacer         desplegar en pro   confirmar en
                                                              el repositorio
quien aprueba producción                   nadie              equipo dueño (095)
```

Y el efecto sobre la clase 098: el flujo más peligroso del inventario dejó de serlo.

A los seis meses.

```
recursos en deriva permanente                    —        13 → 0-2
tiempo hasta corregir un cambio manual        1 día        2 min
cambios manuales en el entorno              2 / mes       0 / mes
credenciales de clúster en la canalización     sí           no
reconstruir un entorno completo           4 páginas    apuntar un agente
prueba de reconstrucción                  1 vez         cada despliegue de dev
bucles parados sin que nadie lo supiera       —      4 detectados por antigüedad
podas detenidas por el umbral                 —        2, ambas errores
```

La lección que esta clase traslada al resto de la parte 08: el bucle hizo lo que la parte 06 predijo —la deriva pasó de un día a dos minutos, y los cambios manuales a cero—, y a cambio trajo tres problemas que la detección programada no tenía: deriva permanente por campos ajenos, borrados por poda y un bucle parado que no da error. Los tres tienen la misma forma que ya conocemos: el primero es la ley 15, el tercero es la ley 13, y el segundo es la ley 14 —una decisión automática sobre lo que existe o no existe, tomada a partir de una confirmación equivocada—.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/103-gitops-con-argo-cd-o-flux/lab.py
```

El laboratorio selecciona el motor de práctica gitops y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa reconciliacion-gitops en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es reconciliación declarativa y evidencia de drift. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye reconciliacion-gitops para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

Errores frecuentes

Síntoma	Causa probable	Corrección
La mitad de los recursos aparecen en deriva permanente y nadie mira el panel	El bucle compara campos que escriben otros controladores por diseño	Declara la propiedad de campo y excluye réplicas del autoescalador, anotaciones inyectadas y campos rellenados por el sistema.
Una confirmación mal hecha borra recursos del entorno	La poda interpreta que lo que ya no se declara debe desaparecer	Umbral de seguridad por porcentaje, anotación de no-podar en volúmenes y bases de datos, y poda manual en producción durante la adopción.
Un servicio lleva días sin recibir cambios y el panel lo muestra bien	Ley 13: el agente dejó de sincronizar y eso no produce error	Alerta por antigüedad de la última reconciliación correcta, no por fallo.
Durante un incidente alguien desactiva el bucle y no vuelve a activarse	El camino de emergencia por confirmación es más lento que tocar a mano	Haz el camino de emergencia rápido —sin aprobación, sincronización inmediata, aviso automático— y mide cuántos bucles quedan desactivados.
El bucle no detecta que se ha desplegado otra versión	Lo declarado usa una etiqueta móvil, que no identifica un artefacto	Declara la huella del artefacto, como exige la inmutabilidad de la clase 099.
Hay secretos en el repositorio de entorno	Lo declarado incluye el valor en vez de una referencia	Declara dónde está el secreto y deja que un operador lo resuelva con la identidad de la carga.

Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

Preguntas de comprobación

1. ¿Qué permiso pierde la canalización al pasar al modelo de tirar y por qué importa?
2. ¿Qué diferencia hay entre el repositorio de código y el de entorno, y cuál despliega?
3. ¿Qué campos no debe gobernar el bucle y qué pasa si no se excluyen?
4. ¿Por qué la poda es a la vez necesaria y peligrosa, y con qué tres protecciones se usa?
5. ¿Por qué la alerta del bucle mira la antigüedad de la reconciliación y no los fallos?

Referencias

- OpenGitOps (2025). GitOps principles — declarativo, versionado, extraído automáticamente y reconciliado de forma continua. <https://opengitops.dev/>
- Argo CD (2025). Diffing customization and resource pruning — propiedad de campo, exclusiones y protecciones de poda. <https://argo-cd.readthedocs.io/en/stable/user-guide/diffing/>
- Flux (2025). Bootstrap and reconciliation — agente dentro del clúster, intervalos y estado de sincronización. <https://fluxcd.io/flux/installation/bootstrap/>
- External Secrets Operator (2025). SecretStore and ExternalSecret — declarar la referencia sin guardar el valor. <https://external-secrets.io/latest/introduction/overview/>
- CNCF (2025). GitOps working group: multi-environment promotion — repositorio de entorno y promoción por huella. <https://github.com/cncf/tag-app-delivery>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Evaluación — 103 GitOps con Argo CD o Flux

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega reconciliacion-gitops con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.

104 — Ambientes efímeros y promoción entre entornos

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 4 Laboratorio: delivery · Estado: EXECUTABLECORE

Propósito

Crear un entorno completo por cada cambio, usarlo mientras el cambio se revisa y destruirlo después. Con lo declarado en un repositorio y un agente que lo materializa, la infraestructura es la parte fácil: la clase se ocupa de las dos partes difíciles —los datos y las dependencias— y de la tercera que nadie planea, destruirlos. Y termina con la lista honesta de lo que un entorno efímero no puede verificar, para no confiar en él más de lo que aguanta.

Resultados de aprendizaje

Al finalizar podrás:

1. Crear un entorno por cambio a partir de lo ya declarado.
2. Elegir una estrategia de datos y asumir su compromiso.
3. Resolver las dependencias sin contaminar entornos compartidos ni multiplicar el coste.
4. Garantizar la destrucción, que es donde se va el presupuesto.
5. Enumerar lo que un entorno efímero no puede verificar.

Conceptos centrales

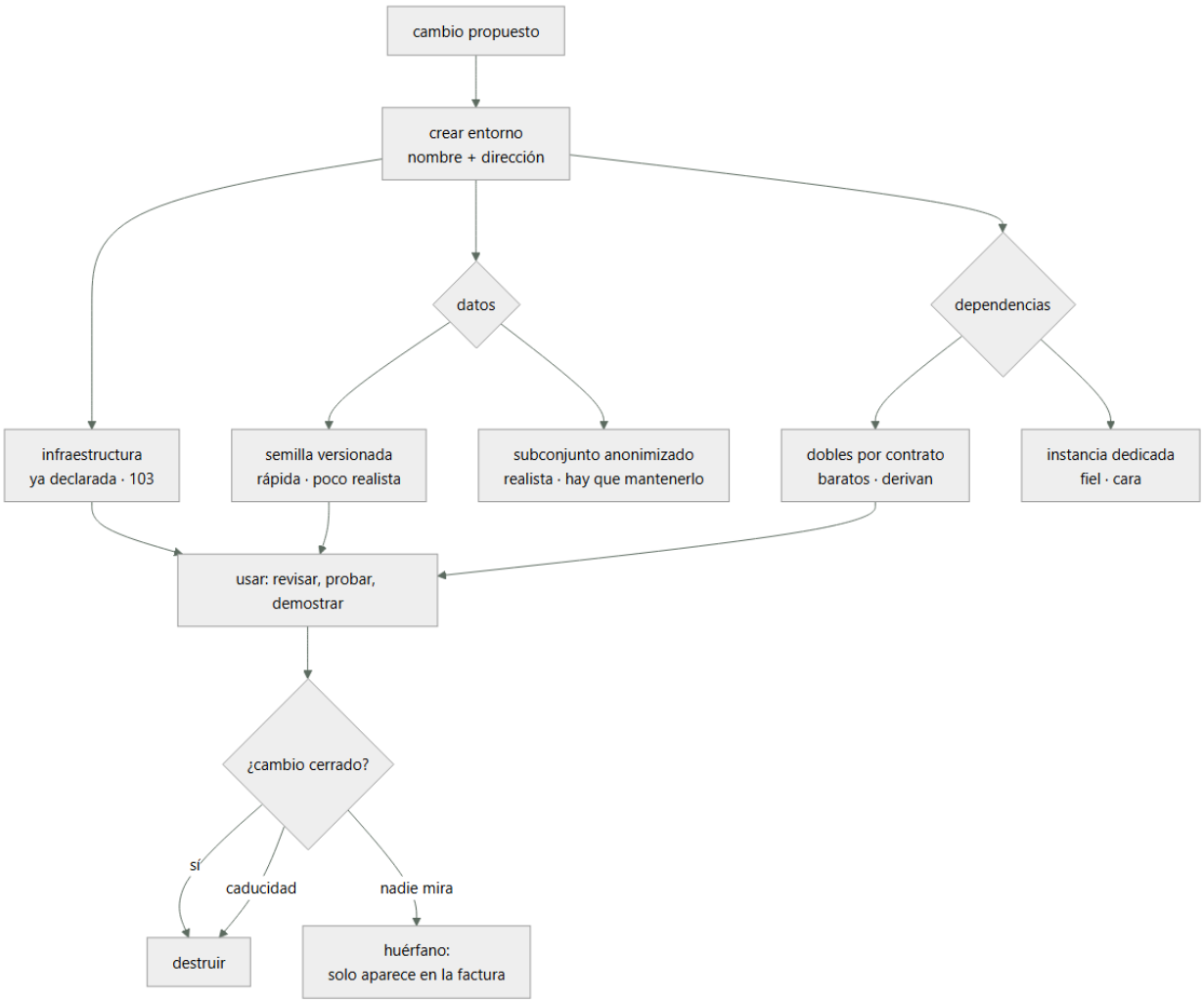
Concepto	Comprensión verificable
entorno efímero	Entorno completo creado para un cambio concreto, con su nombre y su dirección, y destruido cuando el cambio se cierra.
semilla de datos	Conjunto mínimo de datos versionado que hace que el entorno sea utilizable. Es parte del código, no un volcado.
subconjunto anonimizado	Extracto de producción con los datos personales sustituidos, conservando volumen y formas suficientes para que las pruebas signifiquen algo.
doble de dependencia	Sustituto de un servicio externo que responde según su contrato. Barato, y se aleja de la realidad si nadie verifica el contrato.
caducidad	Tiempo de vida máximo tras el cual el entorno se destruye aunque nadie lo pida. Es lo único que impide que el gasto crezca sin techo.
huérfano	Entorno cuyo cambio ya se cerró y que sigue existiendo. Ley 13: nadie da un error por él; solo aparece en la factura.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. Lo fácil y lo difícil

Con lo declarado en un repositorio y un agente que lo materializa —clase 103—, crear la infraestructura de un entorno nuevo es apuntar el agente a otra ruta con otro prefijo. Eso ya está resuelto.

Lo que decide si el entorno sirve para algo es lo otro:

```
infraestructura                 baja        bajo        alta
datos                           ALTA        medio       variable
dependencias externas           media       ALTO        variable
destrucción                     baja        —           —
```

Y conviene fijar antes de nada qué pregunta responde el entorno, porque de eso dependen las tres decisiones siguientes:

```
«¿esta pantalla se ve bien?»          semilla mínima basta
«¿este flujo completo funciona?»      hacen falta datos coherentes
«¿esta consulta aguanta el volumen?»  hace falta volumen real
                                      → y eso rara vez cabe en un efímero
```

La tercera es la del incidente B de la clase 102, y es la que se responde mal más a menudo.

Y lo que un entorno efímero aporta que no aporta ningún otro mecanismo de la parte 08:

```
una dirección que se puede abrir mientras el cambio se revisa
→ la revisión deja de ser leer un diferencial
→ y quien pidió el cambio puede verlo antes de que exista
```

Y un uso menos citado y muy rentable: probar los módulos de infraestructura de verdad. La clase 090 comprobaba los módulos analizando el plan; con entornos efímeros se pueden crear, comprobar y destruir, que es la única forma de

detectar lo que solo falla al aplicar.

Y el nombrado, que parece un detalle y no lo es:

```
prefijo derivado del cambio: pr-1421
  espacio de nombres      pr-1421
  dirección               pr-1421.dev.cloudshop.example
  etiquetas en la nube    cambio=1421, caduca=2026-08-05
```

Las etiquetas de la última línea son las que permiten encontrar los huérfanos, y sin ellas el apartado cuarto no tiene forma de funcionar.

2. Los datos, que es la parte difícil

Un entorno sin datos no permite probar casi nada, y copiarlos de producción no es una opción. Las tres estrategias, con su compromiso real:

```
SEMILLA VERSIONADA
  un conjunto pequeño, en el repositorio, cargado al crear
  + rápido, reproducible, sin datos personales
  + los casos de prueba son explícitos y se revisan como código
  − no detecta nada que dependa del volumen ni de la variedad real

SUBCONJUNTO ANONIMIZADO
  extracto de producción con los datos personales sustituidos
  + formas y distribuciones parecidas a las reales
  − hay que mantener la anonimización cuando cambia el esquema
  − y hay que demostrar que la anonimización es correcta

COPIA DE PRODUCCIÓN
  − datos personales fuera de su entorno, con menos controles
  − tamaño que hace inviable la creación por cambio
  → no
```

Y la elección práctica en casi todos los casos es la primera más un mecanismo que compense su debilidad:

```
semilla para el entorno efímero
+ un entorno persistente con subconjunto anonimizado, compartido,
  para lo que depende del volumen
```

Y una advertencia sobre la anonimización que conviene decir sin rodeos: sustituir el nombre y el correo no es anonimizar. Un conjunto de datos con fechas, importes y códigos postales reales puede reidentificar a personas por combinación. Si se usa un subconjunto, la evaluación de reidentificación es parte del trabajo, no un trámite.

La semilla como código, que es lo que la hace sostenible:

```
datos/semilla/001-catalogo.sql        productos, categorías
datos/semilla/002-clientes.sql        12 clientes de prueba
datos/semilla/003-pedidos.sql         pedidos en cada estado posible
```

El tercer fichero es el que más valor da y el que se olvida: un pedido en cada estado, incluidos los estados raros. Es lo que permite probar el caso que en producción ocurre una vez al mes.

Y el mantenimiento, que es el punto donde estas cosas mueren:

```
la migración de esquema se aplica a la semilla en la misma confirmación
→ si la semilla no carga, la canalización falla
→ y así la semilla no puede quedarse atrás sin que nadie se entere
```

3. Las dependencias, que es la parte cara

Un servicio rara vez está solo. Y las tres formas de resolver lo que hay alrededor tienen costes muy distintos:

```
COMPARTIR LA INSTANCIA DE DESARROLLO
  + coste cero
  − los entornos se contaminan entre sí
  − y una prueba de un cambio rompe la de otro

DESPLEGAR EL SISTEMA ENTERO EN EL EFÍMERO
  + aislamiento completo
  − quince servicios por cada cambio abierto
  − y el tiempo de creación se dispara

DOBLES POR CONTRATO
  + baratos y rápidos
```

```
  – se alejan de la realidad si nadie verifica el contrato
```

Y la combinación que suele funcionar:

```
el servicio que cambia                  real, desplegado en el efímero
sus dependencias directas               reales, si son pocas
el resto                                dobles por contrato
servicios de terceros                   dobles SIEMPRE
```

La última línea no es negociable por dos motivos: cuestan dinero por llamada y tienen efectos reales. Un entorno efímero que envía correos de verdad, cobra de verdad o llama a un proveedor de verdad no es un entorno de pruebas: es producción con otro nombre.

Y el peligro de los dobles es el mismo que la clase 100 nombró: derivan. El doble responde lo que se escribió hace ocho meses y el servicio real ya responde otra cosa. Lo que lo evita es lo que aquella clase llamó nivel de contrato:

```
el doble se genera a partir del contrato publicado del servicio real
y una prueba periódica comprueba que el servicio real sigue cumpliéndolo
→ si el real cambia, la prueba falla y el doble se regenera
```

Y una cuenta que decide el diseño, con los números de una organización mediana:

```
cambios abiertos a la vez, media                      9
servicios desplegados por entorno, si todo es real   15
total                                               135 despliegues vivos

con dobles para lo que no cambia                     2-3 por entorno
total                                                ~25
```

Y el tiempo de creación, que es lo que decide si la gente lo usa —la ley 16 otra vez—:

```
por debajo de 10 min      se usa
10 a 20 min               se usa a regañadientes
por encima de 30 min      no se usa
```

Lo que más baja ese tiempo no es optimizar el despliegue: es no crear lo que no cambia.

4. Destruir, que es donde se va el presupuesto

Este apartado es la ley 13 aplicada al dinero: un entorno que nadie destruye no da ningún error. Solo aparece en la factura, dos meses después.

Los tres mecanismos, y hacen falta los tres:

```
1. al cerrar el cambio      lo destruye el mismo flujo que lo creó
                            → falla si el flujo no se ejecuta

2. por caducidad            se destruye a las N horas de inactividad
                            → sin depender de ningún evento

3. barrido de huérfanos     programado, busca por etiqueta lo que existe
                            y cuyo cambio ya está cerrado
                            → la red que recoge lo que se escapó de las otras dos
```

El segundo es el que de verdad sostiene esto, porque no depende de que ocurra nada. Y el tercero es el que descubre lo que las etiquetas mal puestas dejaron fuera:

```
# entornos vivos cuyo cambio ya está cerrado
$ kubectl get ns -l tipo=efimero -o json \
  | jq -r '.items[] | "\(.metadata.name) \(.metadata.labels.cambio)"' \
  | while read ns pr; do
      estado=$(gh pr view "$pr" --json state -q .state 2>/dev/null || echo DESCONOCIDO)
      [ "$estado" != "OPEN" ] && echo "huérfano: $ns (cambio $pr: $estado)"
    done
```

Y lo que hay que vigilar, con la disciplina de la clase 057:

```
entornos vivos ahora mismo
edad del más antiguo
huérfanos encontrados por el barrido
coste del mes atribuido a efímeros
```

La segunda línea es la más informativa: si el más antiguo tiene tres semanas, la caducidad no está funcionando.

Y una precaución sobre lo que la destrucción deja atrás, que es donde se acumula el gasto invisible:

```
volúmenes persistentes con política de conservar
```

```
registros de imágenes con una etiqueta por cambio
copias de seguridad automáticas de las bases de datos creadas
registros de auditoría, que sí deben conservarse
```

Las tres primeras se limpian; la cuarta no.

Lo que un entorno efímero no puede verificar, que es la parte que conviene decir en voz alta para que nadie confíe de más:

```
rendimiento con el volumen de datos real     ← incidente B de la clase 102
comportamiento bajo carga real
efectos que solo aparecen tras días de ejecución: fugas, acumulaciones
interacciones con el tráfico real y su composición
cualquier cosa que dependa de la escala
```

Y la consecuencia: un entorno efímero no sustituye al canario de la clase 102. Responde antes y responde menos.

Y la lista de comprobación de la clase:

- [ ] está escrito qué pregunta responde el entorno
- [ ] la infraestructura sale de lo ya declarado, sin plantillas aparte
- [ ] la semilla está versionada y las migraciones se le aplican en la misma confirmación
- [ ] hay un pedido en cada estado posible en la semilla
- [ ] los servicios de terceros son SIEMPRE dobles, nunca reales
- [ ] los dobles se generan del contrato y hay prueba de que el real lo cumple
- [ ] la creación tarda menos de diez minutos
- [ ] hay destrucción al cerrar, caducidad y barrido de huérfanos
- [ ] se vigila la edad del entorno más antiguo
- [ ] se limpian volúmenes, imágenes y copias que la destrucción deja atrás
- [ ] está escrito qué NO verifica este entorno

Y el cierre que enlaza con la clase siguiente: hasta aquí, cambiar el comportamiento de producción exige desplegar. Separar las dos cosas —desplegar el código y activar el comportamiento— cambia lo que significa una entrega, y es la materia de la clase 105.

Ejemplo trabajado

CloudShop añade un entorno por cada cambio propuesto. La infraestructura estaba resuelta por la clase 103; el ejercicio son las otras tres partes, y termina con la cuenta de qué detectaron y qué se escapó.

Primer intento: todo real. Dura once días.

```
servicios desplegados por entorno                15
tiempo de creación                          34 min
cambios abiertos a la vez, media                 9
despliegues vivos                              135
coste mensual proyectado                   4.100 €
uso real: entornos abiertos por los equipos    2 de 9
```

Treinta y cuatro minutos y dos de cada nueve usados. Es la ley 16 en su forma más limpia: la herramienta era correcta y nadie la usaba porque era lenta.

Segundo intento: solo lo que cambia.

```
servicios desplegados                    15              2,4 (media)
tiempo de creación                    34 min           6 min 40 s
coste mensual                       4.100 €             520 €
entornos usados por los equipos      2 de 9            9 de 9
```

Y los dobles que sustituyeron a los otros doce servicios se generaron del contrato publicado de cada uno, con una prueba semanal que comprueba que el servicio real sigue cumpliéndolo. En seis meses, esa prueba falló cuatro veces; las cuatro eran contratos que habían cambiado sin avisar, que es exactamente el defecto que la clase 100 clasificó como de nivel de contrato.

Los datos: dos intentos y una decisión honesta.

El primer intento fue un subconjunto anonimizado de producción.

```
tamaño del subconjunto                    2,4 GB
tiempo de carga                            4 min
revisión de reidentificación          no se hizo
```

Al hacer la revisión que faltaba, el resultado paró el enfoque:

```
nombres y correos sustituidos                          sí
fechas de nacimiento, códigos postales e importes   reales
→ 71 clientes reidentificables por combinación de tres campos
```

Se cambió a semilla versionada, y el subconjunto quedó solo en un entorno persistente con los mismos controles que producción.

```
tamaño                                        2,4 GB        18 MB
tiempo de carga                                4 min          11 s
datos personales                               reales      ninguno
casos raros cubiertos                    los que hubiera   explícitos
```

Y la última línea resultó ser una ventaja inesperada. La semilla incluye un pedido en cada estado, incluidos tres que en producción ocurren menos de una vez al mes:

```
defectos detectados en 6 meses por estados raros de la semilla        7
de ellos, imposibles de reproducir con el subconjunto                 5
```

Cinco defectos que el extracto de producción no habría encontrado, porque esos estados no estaban en la ventana extraída.

La destrucción: el mes que costó 1.900 €.

El flujo de destrucción se ejecutaba al cerrar el cambio. Un mes después:

```
entornos vivos                                    31
cambios abiertos                                   9
huérfanos                                         22
edad del más antiguo                           26 días
coste del mes                                   1.900 €
```

Las causas de los veintidós:

```
cambios cerrados sin fusionar (el flujo no se dispara)        14
fallos del flujo de destrucción                                5
entornos creados a mano para depurar                           3
```

Se añadieron la caducidad y el barrido:

```
entornos vivos                                    31          9-11
huérfanos                                         22           0-1
edad del más antiguo                           26 días        31 h
coste mensual                                   1.900 €       520 €
```

Y el barrido encontró además lo que la destrucción dejaba atrás:

```
volúmenes con política de conservar, sin dueño        41   →  liberados
etiquetas de imagen por cambio, ya cerrado           380   →  purgadas
copias de seguridad automáticas de bases efímeras     64   →  desactivadas
```

Qué detectaron, y qué se escapó.

```
defectos detectados en el entorno efímero, 6 meses            34
  de interfaz y de flujo completo                             19
  de configuración específica de entorno                       8
  de contrato entre servicios                                  7

defectos que llegaron a producción igualmente                  6
  dependientes del volumen de datos                            3
  visibles solo tras días de ejecución                         2
  dependientes de la composición del tráfico real              1
```

Los seis de la segunda lista están en la lista de «lo que no verifica» del apartado cuarto, escrita antes de medirlos. Ninguno es una sorpresa, y los seis siguen siendo materia del canario de la clase 102.

A los seis meses.

```
entorno por cambio                               no            sí
tiempo de creación                                —         6 min 40 s
coste mensual de efímeros                         —           520 €
huérfanos                                         —           0-1
datos personales fuera de producción     en 2 entornos          0
defectos detectados antes de fusionar             —         34 / 6 meses
defectos de contrato detectados                   0             7
revisiones con dirección abierta                0 %          88 %
```

La lección que esta clase traslada al resto de la parte 08: de los tres problemas, el que más costó no fue técnico. La creación se arregló no creando lo que no cambia; la destrucción se arregló con una caducidad que no depende de ningún evento; y los datos se arreglaron renunciando al realismo, que era lo contrario de la intuición inicial. La semilla pequeña, con casos explícitos, encontró cinco defectos que el extracto de producción no contenía.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/104-ambientes-efimeros-y-promocion-entre-entornos/lab.py
```

El laboratorio selecciona el motor de práctica delivery y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa flujo-ambientes en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es un pipeline con gates, promoción y rollback. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye flujo-ambientes para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

Errores frecuentes

Síntoma	Causa probable	Corrección
Los entornos existen y los equipos no los usan	Tardan demasiado en crearse; ley 16	Despliega solo el servicio que cambia y sustituye el resto por dobles generados del contrato.
La factura crece sin que nadie sepa por qué	Ley 13: un entorno que nadie destruye no produce ningún error	Caducidad por inactividad, destrucción al cerrar y barrido programado de huérfanos por etiqueta.
Hay datos personales en entornos con menos controles que producción	Se extrajo un subconjunto y se sustituyeron solo nombres y correos	Usa semilla versionada; si hace falta volumen real, evalúa la reidentificación y trata ese entorno con los controles de producción.
Las pruebas pasan contra los dobles y fallan contra el servicio real	Los dobles derivaron respecto del contrato real	Genera los dobles del contrato publicado y comprueba periódicamente que el servicio real lo cumple.
Dos cambios se rompen las pruebas entre sí	Comparten la misma instancia de las dependencias	Aísla con dobles por entorno; comparte solo lo que ningún cambio modifica.
Un cambio pasa el entorno efímero y falla en producción por rendimiento	El entorno no tiene ni el volumen de datos ni la carga real	Escribe qué no verifica el entorno y mantén el canario de la clase 102 para lo que depende de la escala.

Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

Preguntas de comprobación

1. ¿Cuál es la parte fácil de un entorno efímero y cuáles son las dos difíciles?
2. ¿Qué compromiso tiene cada estrategia de datos y por qué la copia de producción no es una opción?
3. ¿Por qué los servicios de terceros deben ser siempre dobles?
4. ¿Qué tres mecanismos de destrucción hacen falta y por qué no basta con el primero?
5. ¿Qué cosas no puede verificar un entorno efímero, y qué mecanismo las cubre?

Referencias

- Fowler, M. (2025). Test doubles and contract tests — dobles por contrato y verificación contra el servicio real. <https://martinfowler.com/bliki/ContractTest.html>
- Argo CD (2025). ApplicationSet: pull request generator — un entorno por cambio propuesto, y su destrucción al cerrarlo. <https://argo-cd.readthedocs.io/en/stable/operator-manual/applicationset/Generators-Pull-Request/>
- ICO (2025). Anonymisation and re-identification risk — por qué sustituir identificadores directos no basta. <https://ico.org.uk/for-organisations/uk-gdpr-guidance-and-resources/anonymisation/>
- FinOps Foundation (2025). Tagging and orphaned resource detection — encontrar lo que quedó vivo sin dueño. <https://www.finops.org/framework/capabilities/resource-utilization-efficiency/>
- Testcontainers (2025). Ephemeral dependencies for tests — dependencias desechables con ciclo de vida acotado. <https://testcontainers.com/guides/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 08 en PDF · Manual integral

Evaluación — 104 Ambientes efímeros y promoción entre entornos

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega flujo-ambientes con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.

105 — Feature flags y separación deploy-release

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 4 Laboratorio: delivery · Estado: EXECUTABLECORE

Propósito

Separar el despliegue de la activación: el código llega a producción apagado y se enciende después, para quien se decida y cuando se decida. Eso resuelve tres problemas que las clases anteriores dejaron abiertos —el cambio grande que no cabe en un incremento pequeño, el defecto que no se puede revertir y la funcionalidad que hay que apagar sin desplegar—. Y trae uno nuevo que hay que mirar de frente: cada interruptor es un cambio de producción sin canalización, sin revisión y sin registro, salvo que se le construyan los tres.

Resultados de aprendizaje

Al finalizar podrás:

1. Separar el despliegue del código de la activación del comportamiento.
2. Clasificar cada interruptor por su vida esperada y tratarlo en consecuencia.
3. Acotar el coste combinatorio antes de que sea inmanejable.
4. Aplicar al cambio de un interruptor los mismos controles que a un despliegue.
5. Retirar los interruptores temporales, que es lo que decide si esto funciona a los dos años.

Conceptos centrales

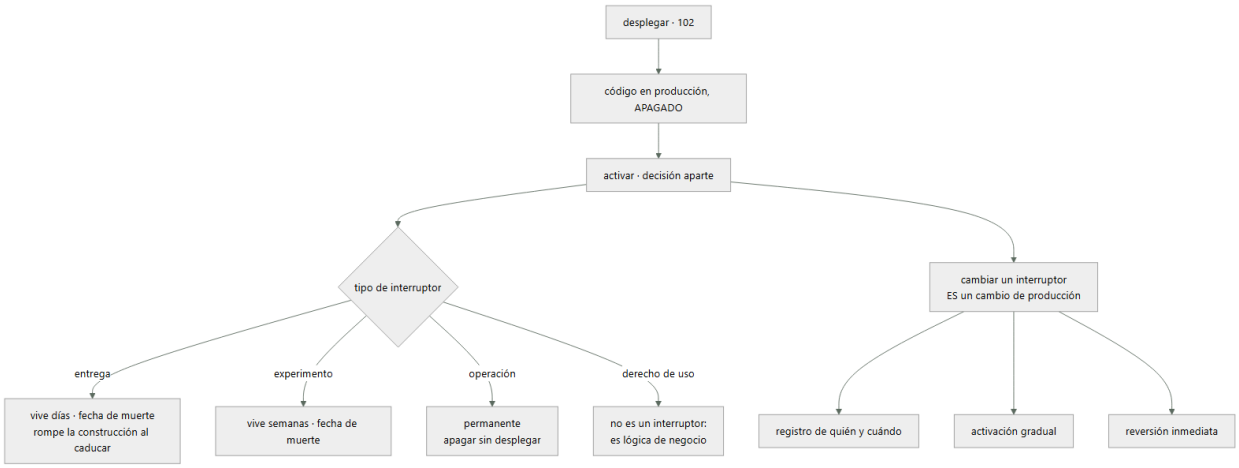
Concepto	Comprensión verificable
desplegar frente a activar	Desplegar es poner el código en producción; activar es hacer que su comportamiento ocurra. Los interruptores separan las dos decisiones.
interruptor de entrega	Oculta un cambio incompleto mientras se construye. Vive días o semanas y debe morir.
interruptor de operación	Apaga una funcionalidad sin desplegar. Es permanente por diseño y es la respuesta al cambio irreversible de la clase 102.
asignación estable	Que un mismo usuario obtenga siempre la misma variante, derivándola de su identificador. Sin ella, el usuario ve el sistema parpadear.
coste combinatorio	Con N interruptores independientes hay 2^N caminos posibles y se prueban dos. Es el motivo por el que pocos interruptores simultáneos importan más que ninguna otra práctica.
fecha de muerte	Plazo tras el cual un interruptor temporal rompe la construcción. Es lo único que impide que el inventario crezca sin techo.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. Qué compra separar el despliegue de la activación

Hasta aquí, poner código en producción y cambiar el comportamiento de producción eran la misma acción. Separarlas resuelve tres problemas concretos que este programa dejó abiertos:

```
el incremento que no cabe                    clase 097
  un cambio de tres semanas no se puede integrar a diario
  → se integra a diario, apagado, y se enciende al final

el cambio que no se puede revertir           clase 102
  el punto de no retorno hace inútil volver a la versión anterior
  → un interruptor apaga el comportamiento sin desplegar nada

la activación por audiencia
  → encender para el 1 %, para un país o para los equipos internos
```

Y una consecuencia que cambia la forma de trabajar más que las tres anteriores: el despliegue deja de ser un evento. Si el código llega apagado, desplegar es rutina y la decisión de riesgo se toma en otro momento, con otra gente y con posibilidad de deshacerse en segundos.

Y conviene ser claro sobre lo que no compra, porque se vende de más:

```
no sustituye a la estrategia de despliegue de la clase 102
  el código encendido sigue teniendo que llegar de forma escalonada
no sustituye a las pruebas
  un camino apagado no está probado por estar apagado
no hace seguro un cambio incompatible
  si el esquema rompe la convivencia, el interruptor no lo arregla
```

La segunda línea merece una precisión, porque es una fuente de sorpresas: el código detrás de un interruptor apagado se despliega igual. Se compila, se carga, se inicializa. Un error de arranque en ese código tumba el servicio aunque la funcionalidad esté apagada.

```
lo que el interruptor apaga:  la ejecución de una rama
lo que NO apaga:              la carga del módulo, sus dependencias,
                              lo que se inicializa al arrancar
```

Y la técnica que aprovecha esto en el mejor sentido —lanzamiento a oscuras— consiste en ejecutar el camino nuevo sin usar su resultado:

```
se ejecuta la ruta nueva en paralelo con la vieja
se sirve el resultado de la vieja
y se compara: latencia, errores, diferencias de resultado
→ se mide con tráfico real sin exponer a nadie
```

Es lo más cercano a probar en producción sin riesgo, y responde justo lo que el entorno efímero de la clase 104 no puede responder: cómo se comporta con el volumen y la composición reales.

2. Cuatro tipos, y solo dos deben morir

Casi todo el desorden de este tema viene de tratar igual cosas que tienen vidas distintas:

```
ENTREGA            oculta un cambio incompleto
                   vida: días o semanas       debe morir
                   quién lo cambia: el equipo que lo puso

EXPERIMENTO        compara dos variantes con usuarios reales
                   vida: la del experimento   debe morir
                   quién lo cambia: quien diseñó el experimento

OPERACIÓN          apaga una funcionalidad o degrada un servicio
                   vida: permanente           no muere
                   quién lo cambia: quien está de guardia

DERECHO DE USO     qué puede hacer cada cliente según su plan
                   vida: permanente
                   → esto NO es un interruptor: es lógica de negocio
```

La cuarta línea es la que más problemas evita. Los derechos de uso son parte del dominio, se prueban, tienen su propia persistencia y no deben vivir en el mismo sistema que los interruptores temporales: si viven ahí, nadie puede limpiar nada porque el inventario mezcla lo desechable con lo permanente.

Y los de operación son los que este programa lleva pidiendo desde la clase 102:

```
apagar la funcionalidad que causó el incidente, sin desplegar
degradar: servir del caché, saltarse el enriquecimiento, desactivar
          las recomendaciones cuando su servicio está caído
cortar el tráfico a un proveedor externo que está fallando
```

Y los de operación tienen un requisito que los demás no tienen, y es el que decide si sirven durante un incidente:

```
tienen que funcionar cuando el sistema está mal
→ evaluación local, con el último valor conocido en memoria
→ un interruptor que necesita consultar un servicio remoto
  para apagarse es inútil justo cuando hace falta
```

Y el valor por defecto se elige con ese criterio: si el sistema de interruptores no responde, cada uno debe caer al valor seguro, que para uno de entrega es apagado y para uno de operación suele ser encendido.

Y la consecuencia organizativa que hace falta escribir: quién puede cambiar cada tipo. Un interruptor de operación que cualquiera puede tocar es una puerta abierta a producción; uno de entrega que solo puede tocar una persona concreta es un cuello de botella.

3. El coste que crece solo

Con N interruptores independientes hay 2^N combinaciones de estado, y se prueban una o dos:

```
 5 interruptores      32 combinaciones
10                 1.024
20             1.048.576
```

Y lo que hace daño no es el número total, sino cuántos están a la vez en un estado distinto entre entornos:

```
en preproducción todo encendido y en producción algunos apagados
→ lo que se probó no es lo que corre
```

Esa es la forma en que un interruptor rompe la garantía de la clase 099: el artefacto es el mismo, pero el comportamiento no.

Las tres reglas que acotan el coste, en orden de eficacia:

```
1. pocos simultáneos     un límite explícito por servicio, y se vigila
2. independientes        si dos interruptores interactúan, es un defecto
                         de diseño, no una combinación que probar
3. vida corta            la mayor parte del inventario debe ser temporal
```

Y la comprobación mínima que sí es viable, en vez de intentar probar 2^N:

```
probar el estado de producción                siempre
probar el estado que se va a activar           siempre
probar cada interruptor encendido y apagado   por separado
no probar combinaciones                       salvo las que interactúan,
                                              que no deberían existir
```

Y una regla de higiene que evita la mitad de los problemas de diagnóstico: el estado de los interruptores forma parte del contexto de cada petición. Si un registro de error no dice qué variantes estaban activas, reproducirlo es adivinar.

```
{"nivel":"error","traza":"a91c…","ruta":"/checkout",
 "interruptores":{"pago-nuevo":true,"envio-express":false}}
```

Y la asignación estable, que es un fallo clásico y muy visible para el usuario: si la variante se decide al azar en cada petición, la misma persona ve el sistema cambiar entre pantallas. Se deriva del identificador:

```
variante = "nueva" if hash_estable(id_usuario, nombre_interruptor) % 100 < porcentaje else "vieja"
```

Y el segundo argumento importa: incluir el nombre del interruptor evita que los mismos usuarios caigan siempre en el grupo de prueba de todos los experimentos.

4. Un cambio de interruptor es un cambio de producción

Aquí está el problema nuevo, y es serio. Toda la parte 08 ha construido controles alrededor del despliegue: revisión, puertas, artefacto firmado, canario, registro. Cambiar un interruptor no pasa por nada de eso.

```
desplegar          revisión, puertas, canario, registro, reversión
cambiar un valor   un clic
```

Y el efecto en producción puede ser mayor. La consecuencia práctica es que hay que reconstruir los cuatro controles imprescindibles sobre el sistema de interruptores:

```
1. REGISTRO      quién, cuándo, valor anterior y nuevo, y por qué
                 → es la primera pregunta de cualquier incidente
2. GRADUAL       1 % → 10 % → 50 % → 100 %, no de 0 a 100
                 → es la clase 102 aplicada a la activación
3. REVERSIÓN     volver al valor anterior en un clic, sin aprobación
4. PERMISOS      quién puede tocar qué, por tipo y por entorno
```

Y una pregunta que conviene responder pronto: si el interruptor está en un repositorio, ¿lo revierte el bucle de la clase 103? Las dos opciones y su compromiso:

```
interruptores en el repositorio    revisados y con historial, y lentos
                                   → sirven para los de entrega
interruptores en un servicio       inmediatos, y fuera del bucle
                                   → imprescindible para los de operación
```

Y si conviven las dos cosas, hay que declarar cuál manda para que el bucle no revierta un apagado de emergencia.

La retirada, que es lo que decide si esto sigue siendo manejable a los dos años. Un interruptor de entrega que nadie quita se convierte en una rama muerta que sigue evaluándose, y en un camino que nadie prueba.

```
al crear un interruptor temporal se fija su fecha de muerte
al pasar la fecha, la construcción falla
y quitarlo es una confirmación que borra la rama muerta, no solo el valor
```

La última línea es la que se olvida: apagar el interruptor no es retirarlo. Mientras el código de las dos ramas siga ahí, el coste combinatorio sigue ahí.

Y lo que se vigila:

```
interruptores vivos, por tipo
temporales por encima de su fecha
edad del más antiguo
interruptores que llevan meses en el mismo valor       ← candidatos a retirar
interruptores que no se evalúan nunca                  ← código muerto
```

Y la lista de comprobación de la clase:

- ■ cada interruptor está clasificado por tipo y vida esperada
- ■ los derechos de uso no viven en el sistema de interruptores
- ■ los de operación se evalúan localmente y funcionan con el sistema caído
- ■ cada uno tiene un valor por defecto seguro si no hay respuesta
- ■ la asignación de variante es estable por usuario y por interruptor
- ■ el estado de los interruptores está en el contexto de los registros
- ■ hay límite explícito de interruptores simultáneos por servicio
- ■ cambiar un valor deja registro de quién, cuándo y por qué
- ■ la activación es gradual y la reversión no requiere aprobación
- ■ los temporales tienen fecha de muerte que rompe la construcción
- ■ retirar significa borrar la rama muerta, no solo apagar el valor

Y el cierre que enlaza con la clase siguiente: las nueve clases de esta parte han añadido canalización, puertas, firma, bucle, entornos e interruptores. Cada equipo no puede construir todo eso por su cuenta, y que cada uno lo construya distinto es peor que no tenerlo. Quién lo mantiene y cómo se ofrece es la materia de la clase 106.

Ejemplo trabajado

CloudShop introduce interruptores para resolver dos problemas concretos: un rediseño del pago que lleva cinco semanas sin poder integrarse y el incidente C de la clase 102, que no se pudo revertir. A los dieciocho meses, el inventario es el que enseña la lección.

Los dos problemas iniciales.

El rediseño del pago se integraba en una rama larga —justo lo que la clase 097 dijo que no—:

```
vida de la rama                        35 días          < 1 día
conflictos al fusionar                 211 líneas          0
integraciones a la principal          1 en 5 semanas   4 al día
```

Y el incidente C —pedidos duplicados, dos horas y cuarenta minutos, irreversible— se reprodujo en un ensayo con un interruptor de operación delante:

```
detección                                2 h 40             4 min
parar el daño                          no se pudo        11 s
pedidos duplicados                        1.847              23
```

Once segundos frente a «no se pudo». Es la respuesta que la clase 102 no tenía para un cambio pasado el punto de no retorno.

El inventario a los dieciocho meses.

```
interruptores vivos                                      147
  de entrega                                              71
  de experimento                                          18
  de operación                                            22
  derechos de uso                                         36
```

Y al mirar los 71 de entrega:

```
por encima de su fecha prevista                           58
en el mismo valor desde hace más de 6 meses               44
que no se evalúan nunca (el código ya no los consulta)    12
el más antiguo                                       29 meses
```

Cuarenta y cuatro interruptores encendidos al 100 % desde hace medio año: no son interruptores, son ramas muertas que siguen evaluándose. Y doce que ni siquiera se consultan.

El incidente que lo puso en evidencia.

```
síntoma      el 3 % de los pedidos con el importe de envío mal calculado
duración     9 días hasta detectarlo
causa        dos interruptores que interactuaban:
             «envío-express» encendido y «tarifas-v2» apagado
             una combinación que no se había probado nunca
```

Y la pregunta del apartado tercero, respondida con datos:

```
interruptores en ese servicio                  19
combinaciones posibles                      524.288
combinaciones probadas                          2
combinaciones vividas en producción            31
```

Y lo que costó reproducirlo: los registros de error no decían qué variantes estaban activas. Tres días de los nueve se fueron en eso. Se añadió el estado al contexto de la petición, y el siguiente caso parecido se diagnosticó en veinte minutos.

La limpieza, y lo que reveló.

Se fijó fecha de muerte para todo lo temporal y se rompió la construcción al pasarla. Retirar significaba borrar la rama muerta, no apagar el valor:

```
interruptores de entrega                  71          38          9
líneas de código eliminadas                —       4.100      7.350
defectos encontrados al retirar            —           6          9
```

Los quince defectos son el hallazgo interesante: aparecieron al borrar la rama vieja y descubrir que la rama nueva nunca había manejado un caso que la vieja sí manejaba. Estaba oculto porque el camino viejo seguía cubriéndolo para una parte del tráfico.

Y los treinta y seis derechos de uso se sacaron del sistema de interruptores y se llevaron al dominio, con sus pruebas:

```
dónde viven                       sistema de interruptores   modelo de datos
quién los puede cambiar           cualquiera con acceso      proceso comercial
cubiertos por pruebas                     no                     sí
inventario a limpiar                     147                     93
```

El cambio de interruptor como cambio de producción.

Dos meses después de empezar hubo un incidente causado no por un despliegue, sino por un valor:

```
alguien activó «recomendaciones-v3» al 100 % directamente
latencia del percentil 99: de 240 ms a 3,1 s
tiempo hasta que alguien supo QUIÉN lo había cambiado: 35 min
```

Los treinta y cinco minutos son el argumento del apartado cuarto. Se añadieron los cuatro controles:

```
registro de quién, cuándo y por qué             no             sí
activación gradual obligatoria                  no        1-10-50-100
reversión en un clic, sin aprobación            sí             sí
permisos por tipo y entorno                     no             sí

tiempo medio hasta identificar un cambio    35 min          inmediato
incidentes causados por un valor          3 / trimestre   0-1 / trimestre
```

A los seis meses de la limpieza.

```
interruptores vivos                             147             31
  temporales por encima de su fecha              58              0
  que no se evalúan nunca                        12              0
edad del más antiguo                          29 meses         6 semanas
límite por servicio                            no había     5, vigilado
defectos por combinación de interruptores     1 / 9 días          0
registros con estado de interruptores            no             sí
cambios de valor con registro y gradualidad      no             sí
ramas largas de más de una semana                 3              0
```

La lección que esta clase traslada al resto de la parte 08: los interruptores hicieron lo prometido —la rama de cinco semanas pasó a integrarse cuatro veces al día, y el incidente irreversible se paró en once segundos—, y a cambio crearon un inventario de ciento cuarenta y siete elementos que nadie gobernaba. Cuarenta y cuatro llevaban medio año en el mismo valor: eso no es un interruptor, es una rama muerta que se evalúa en cada petición. Y el mecanismo que lo arregló fue el mismo que este programa lleva usando desde la clase 046 para las excepciones: una fecha que rompe la construcción.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/105-feature-flags-y-separacion-deploy-release/lab.py
```

El laboratorio selecciona el motor de práctica delivery y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa plan-feature-flags en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es un pipeline con gates, promoción y rollback. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye plan-feature-flags para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
El inventario de interruptores crece y nadie sabe cuáles se pueden quitar	Los temporales no tienen fecha de muerte y conviven con los permanentes	Clasifica por tipo, saca los derechos de uso del sistema y pon fecha de muerte que rompa la construcción a los temporales.
Un defecto aparece solo con cierta combinación de valores	Interruptores que interactúan; con N hay 2^N caminos y se prueban dos	Limita los simultáneos por servicio y trata la interacción entre dos interruptores como defecto de diseño, no como combinación a probar.
Reproducir un error lleva días porque no se sabe qué variantes estaban activas	El estado de los interruptores no forma parte del contexto de la petición	Incluye los interruptores evaluados en los registros y en la traza.
El interruptor de emergencia no responde justo durante el incidente	Su evaluación depende de consultar un servicio remoto que también está afectado	Evaluación local con el último valor conocido y valor por defecto seguro cuando no hay respuesta.
Un mismo usuario ve la interfaz cambiar entre pantallas	La variante se sortea en cada petición en vez de derivarse del usuario	Asignación estable a partir del identificador de usuario y del nombre del interruptor.
Un cambio de valor causa un incidente y nadie sabe quién lo hizo	Cambiar un interruptor es un cambio de producción sin ninguno de los controles del despliegue	Registro de quién, cuándo y por qué; activación gradual; reversión inmediata; y permisos por tipo y entorno.

■ Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

■ Preguntas de comprobación

1. ¿Qué tres problemas de clases anteriores resuelve separar el despliegue de la activación?
2. ¿Qué cuatro tipos de interruptor hay y cuáles deben morir?
3. ¿Por qué un interruptor de operación debe evaluarse localmente?
4. ¿Por qué apagar un interruptor no es lo mismo que retirarlo?
5. ¿Qué cuatro controles del despliegue hay que reconstruir sobre el cambio de un valor?

Referencias

- Hodgson, P. (2025). Feature toggles: types and lifecycle — clasificación por vida esperada y coste de mantenimiento. <https://martinfowler.com/articles/feature-toggles.html>
- Humble, J. y Farley, D. (2010). Continuous Delivery, cap. 13 — separar despliegue de activación y lanzamiento a oscuras. <https://www.oreilly.com/library/view/continuous-delivery-reliable/9780321670250/>
- OpenFeature (2025). Specification: evaluation, context and providers — evaluación local, contexto y valores por defecto. <https://openfeature.dev/specification/>
- Kohavi, R. y otros (2020). Trustworthy Online Controlled Experiments — asignación estable y validez de los experimentos. <https://experimentguide.com/>
- Google SRE (2025). Graceful degradation — apagar y degradar como respuesta operativa durante un incidente. <https://sre.google/workbook/managing-load/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 08 en PDF · Manual integral

Anterior	Índice	Siguiente
← 104 · Ambientes efímeros y promoción entre entornos	Parte 08 · Programa	106 · Platform engineering e Internal Developer Platform →

Evaluación — 105 Feature flags y separación deploy-release

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega plan-feature-flags con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.

106 — Platform engineering e Internal Developer Platform

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 4 Laboratorio: platform · Estado: EXECUTABLECORE

Propósito

Decidir quién mantiene todo lo que las clases 073 a 105 han ido acumulando —módulos, plantillas de canalización, puertas, bucle de reconciliación, entornos, interruptores— y cómo se ofrece a quince equipos sin que cada uno lo construya distinto ni tenga que pedirlo por ventanilla. La clase trata la plataforma interna como un producto con usuarios, y defiende una medida incómoda de su éxito: la proporción que la usa sin estar obligada.

Resultados de aprendizaje

Al finalizar podrás:

1. Distinguir una plataforma de una ventanilla de peticiones y de una colección de herramientas.
2. Inventariar qué ofrece la plataforma y qué sigue siendo del equipo de producto.
3. Diseñar el camino asfaltado con su salida, para que sea camino y no jaula.
4. Medir la adopción voluntaria y usarla como señal de calidad del producto.
5. Versionar la plataforma sin romper a quien la consume.

Conceptos centrales

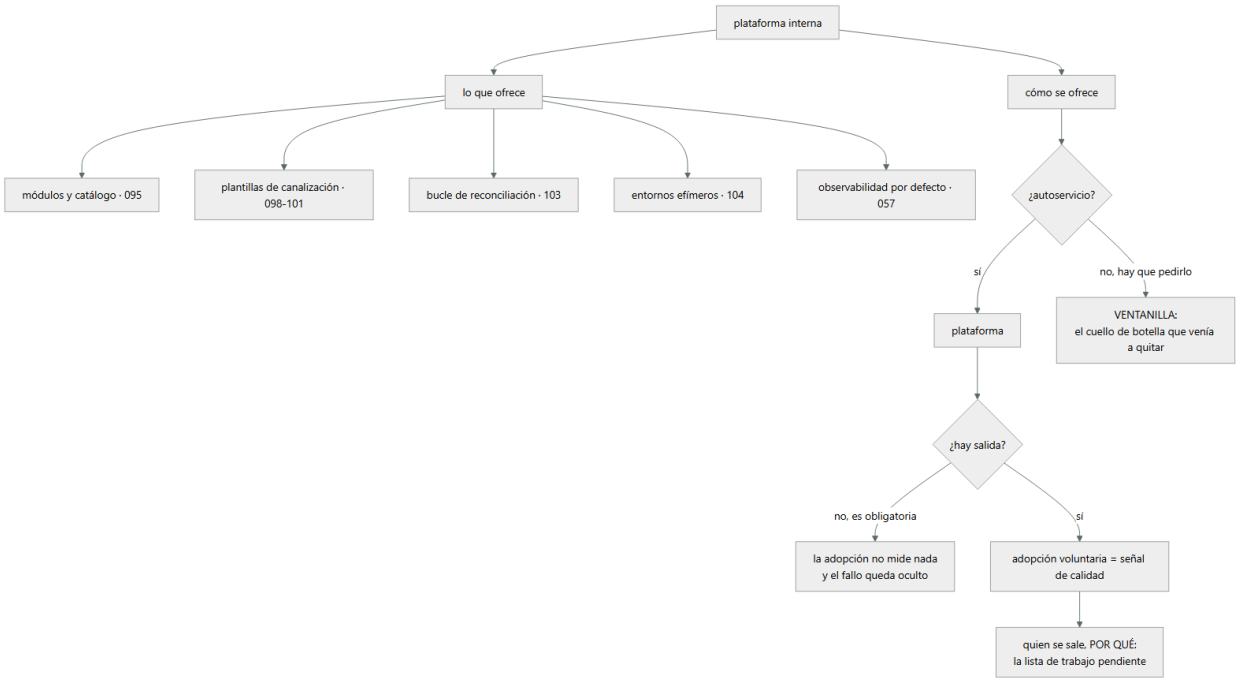
Concepto	Comprensión verificable
plataforma interna	Producto cuyos usuarios son los equipos de la organización. Se mide por adopción y por tiempo ahorrado, no por número de componentes.
camino asfaltado	La forma respaldada de hacer algo: más rápida y con las decisiones difíciles ya tomadas. Su valor está en que ahorra trabajo, no en que sea obligatoria.
salida	Posibilidad explícita de dejar el camino asfaltado sin pedir permiso. Sin ella la adopción deja de significar nada.
ventanilla	Modo de fallo en el que la plataforma es un equipo al que se le piden cosas. Convierte a la plataforma en el cuello de botella que venía a eliminar.
autoservicio	El equipo obtiene lo que necesita sin intervención humana. Es la diferencia entre plataforma y ventanilla.
contrato de la plataforma	Qué versiones se soportan, cuánto duran y con cuánto aviso se retiran. Sin él, cada mejora de la plataforma es una interrupción para quince equipos.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. Qué es y qué no es

Al final de las partes 06, 07 y 08 hay un inventario considerable de cosas que funcionan. El problema no es técnico: es que quince equipos no pueden mantener quince copias de todo eso, y que cada uno lo haga distinto es peor que no tenerlo, porque impide cualquier mejora transversal.

Lo que una plataforma no es, con sus tres modos de fallo:

```
VENTANILLA
  un equipo al que se le piden entornos, permisos y despliegues
  → es el cuello de botella que la nube vino a eliminar
  → y su síntoma es que la métrica que se enseña es «tickets cerrados»

COLECCIÓN DE HERRAMIENTAS
  catorce productos comprados y ninguna integración
  → el equipo sigue teniendo que juntarlos

PLATAFORMA VACÍA
  se construyó lo que el equipo de plataforma quiso construir
  → nadie la usa, y la respuesta es hacerla obligatoria
```

Y lo que sí es, en una frase que se puede comprobar: el equipo consigue lo que necesita sin hablar con nadie.

```
crear un servicio nuevo con canalización, entorno y observabilidad
  ventanilla     ticket, 6 días
  plataforma     un comando, 20 minutos
```

Y el inventario concreto de lo que este programa ya ha construido y que la plataforma ofrece:

```
módulos de infraestructura versionados            clases 085, 086
catálogo de servicios con dueño                   clase 095
caminos asfaltados por tipo de servicio           clase 096
plantillas de canalización con puertas            clases 097-101
firma, inventario y verificación                  clases 067, 101
bucle de reconciliación y repositorio de entorno  clase 103
entornos efímeros por cambio                      clase 104
sistema de interruptores                          clase 105
observabilidad, paneles y alertas por defecto     clases 056, 057
```

Y lo que no es de la plataforma, que hay que escribir para que la frontera no se discuta cada semana:

```
el código del servicio
sus pruebas
su presupuesto de error y su guardia
```

```
sus decisiones de arquitectura
las alertas específicas de su dominio
```

La tercera línea es la que más se intenta empujar hacia la plataforma, y es la que menos debe moverse: quien opera lo que construye es quien puede arreglarlo.

2. Camino asfaltado, no jaula

El camino asfaltado de la clase 096 funcionaba por una razón concreta: era más rápido que hacerlo a mano. No porque estuviera prohibido lo demás.

Y esa distinción decide todo lo que sigue:

```
CAMINO          se elige porque ahorra trabajo
                si nadie lo elige, la plataforma tiene un defecto

JAULA           se impone
                si nadie lo elegiría, no hay forma de saberlo
```

Y la consecuencia incómoda: una plataforma obligatoria oculta su propio fracaso. La adopción es del 100 % por definición, y las quejas se interpretan como resistencia al cambio.

Por eso hace falta la salida, y hace falta que sea explícita:

```
un equipo puede no usar el camino asfaltado
  sin pedir permiso
  asumiendo lo que el camino le daba: puertas, observabilidad, cumplimiento
  y declarándolo en el catálogo, para que se sepa
```

Y lo que se hace con esa información es lo que convierte la salida en una herramienta de producto:

```
quien se salió, y por qué
→ esa lista ES el trabajo pendiente de la plataforma
```

Y hay dos cosas que sí son obligatorias, y conviene separarlas claramente de lo que es camino:

```
OBLIGATORIO      lo que protege a la organización, no lo que la acelera
  identidad federada, sin claves de larga duración (clase 098)
  cifrado y registro de auditoría (clases 046, 054)
  firma verificada en admisión (clases 067, 101)

CAMINO           todo lo demás
```

La regla que ordena la frontera: se impone el control, no la herramienta. Un equipo puede usar otra canalización; lo que no puede es desplegar sin firma verificada.

Y una advertencia sobre la métrica, porque es fácil engañarse: contar equipos que usan la plataforma no dice nada si no se cuenta también el trabajo que les ahorra.

```
adopción voluntaria del camino, por tipo de servicio
tiempo desde «quiero un servicio nuevo» hasta «funcionando en dev»
proporción del trabajo del equipo dedicada a infraestructura
equipos que se salieron, y su motivo
```

La tercera es la que justifica la existencia de la plataforma ante quien paga.

3. La interfaz, y lo que hay debajo

La plataforma se ofrece de tres formas, y conviene que sean tres vistas de lo mismo:

```
línea de comandos     lo que usa quien desarrolla a diario
portal                lo que usa quien busca, descubre o no lo usa a diario
API                   lo que usa la automatización
```

Y la regla que evita el error más caro de este tema: lo que hay debajo tiene que ser usable directamente.

```
mal   el portal es la única forma de crear un servicio
      → cuando el portal falla o falta un campo, nadie puede hacer nada

bien  el portal escribe una confirmación en un repositorio
      → y esa confirmación se puede escribir a mano
```

Es la misma lógica que la clase 103 impuso al bucle: lo declarado es la verdad, y la interfaz es una comodidad encima.

Y el catálogo de la clase 095 es lo que hace que el portal sirva para algo más que crear cosas:

```
qué servicios existen y de quién son
```

```
qué versión de qué módulo usa cada uno
qué está fuera del camino asfaltado, y por qué
qué versiones de plataforma están por retirarse
cómo se pide guardia, cómo se escala, dónde está el manual
```

El contrato de la plataforma, que es lo que decide si las mejoras son bienvenidas o temidas. Los equipos consumen versiones —de módulos, de plantillas, de imágenes base—, y cada cambio incompatible es una interrupción multiplicada por quince.

```
versiones soportadas a la vez                         2
aviso antes de retirar una versión                90 días
cambio incompatible                          versión mayor, nunca en sitio
migración                                    con herramienta, no con documento
```

La última línea es la que más adopción compra: si la plataforma pide migrar, la plataforma escribe la migración. Un documento de doce pasos multiplicado por quince equipos es trabajo que la plataforma está externalizando a sus usuarios.

Y el tamaño, porque es la pregunta que siempre aparece. Como orden de magnitud observado, y no como regla:

```
plataforma pequeña      3-5 personas para 10-20 equipos de producto
por debajo de eso       no puede mantener lo que ofrece, y vuelve a ventanilla
por encima sin demanda  construye lo que nadie pidió
```

Y la señal de que el tamaño está mal en la dirección peligrosa: la proporción del tiempo del equipo de plataforma dedicada a atender peticiones. Si pasa de un tercio, la plataforma se está convirtiendo en ventanilla.

4. Tratarla como producto

Lo que separa una plataforma que se usa de una que se impone es que la primera se gestiona como producto. Y eso significa cuatro cosas concretas:

```
1. TIENE USUARIOS y se les pregunta
   entrevistas, no encuestas de satisfacción
   observar a alguien creando un servicio nuevo enseña más que veinte respuestas

2. TIENE TRABAJO PENDIENTE priorizado por valor
   y el motivo de cada salida entra en esa lista

3. TIENE DOCUMENTACIÓN que es parte del producto
   si el camino asfaltado necesita que alguien lo explique, no está asfaltado

4. TIENE MEDIDAS de uso y de ahorro
   y se publican, incluidas las malas
```

Y los tres antipatrones que aparecen cuando falta lo anterior:

```
la plataforma como impuesto        se cobra y no se elige
la plataforma como museo           componentes que nadie usa y nadie retira
la plataforma como reescritura     cada 18 meses se tira y se empieza de nuevo
```

El segundo se corrige con la misma disciplina que la clase 105 aplicó a los interruptores: medir qué se usa y retirar lo que no.

Y una tensión que conviene nombrar porque no tiene solución limpia: la plataforma quiere estandarizar y los equipos quieren libertad. La forma de resolverla que este programa ha usado sistemáticamente es la de la clase 096:

```
estandarizar lo que nadie quiere decidir
  registro, cifrado, red, identidad, empaquetado
dejar libre lo que diferencia
  lenguaje, marco de trabajo, modelo de datos, arquitectura interna
```

Y la lista de comprobación de la clase:

- [ ] el equipo consigue lo que necesita sin abrir un ticket
- [ ] está escrito qué es de la plataforma y qué es del equipo de producto
- [ ] el camino asfaltado es más rápido que hacerlo a mano, y se mide
- [ ] existe salida explícita, sin pedir permiso, y se declara en el catálogo
- [ ] el motivo de cada salida entra en el trabajo pendiente de la plataforma
- [ ] lo obligatorio son controles, no herramientas
- [ ] lo que hay debajo de la interfaz es usable directamente
- [ ] hay contrato de versiones: cuántas se soportan y con cuánto aviso se retiran
- [ ] cuando la plataforma pide migrar, la plataforma escribe la migración
- [ ] se vigila qué proporción del tiempo se va en atender peticiones

- se retira lo que nadie usa

Y el cierre que enlaza con la clase siguiente: todo lo anterior se justifica con una promesa —que entregar sea más rápido y más seguro—. Comprobar si esa promesa se cumple exige medir, y las medidas que se suelen usar tienen trampas conocidas. Es la materia de la clase 107.

Ejemplo trabajado

CloudShop crea un equipo de plataforma de cuatro personas para quince equipos de producto. El primer año tiene dos fases claramente separadas por una decisión, y la decisión es la que da la lección.

Fase 1, meses 1 a 5: ventanilla.

El equipo se formó con quienes ya hacían la infraestructura, y siguieron haciendo lo mismo con otro nombre:

```
peticiones atendidas al mes                        118
tiempo medio de atención                        4,5 días
proporción del tiempo del equipo en peticiones     84 %
componentes reutilizables construidos               2
tiempo desde «quiero un servicio» hasta dev      6 días
```

Ochenta y cuatro por ciento en peticiones. El indicador que el apartado tercero señala —un tercio— estaba más que duplicado, y la consecuencia era previsible: no quedaba tiempo para construir la plataforma.

Y el catálogo de peticiones enseñaba lo que había que automatizar:

```
crear un servicio nuevo con lo básico              41 %
dar acceso a algo                                  23 %
crear un entorno o una base de datos               19 %
dudas sobre cómo hacer algo                        11 %
otras                                               6 %
```

El 83 % de las peticiones eran las mismas tres cosas, una y otra vez.

La decisión: dejar de atender para poder automatizar.

Se congelaron las peticiones de los tres tipos principales durante seis semanas y se construyó el autoservicio. Fue impopular, y se midió:

```
crear un servicio nuevo                  4,5 días         22 min
dar acceso                               2 días           inmediato
                                                   (grupos del catálogo)
crear entorno de datos                   6 días           14 min
peticiones al mes                          118              19
proporción del tiempo en peticiones        84 %             21 %
```

Fase 2: la plataforma como producto, y la salida.

El camino asfaltado se ofreció sin obligar, con la salida declarada. A los tres meses:

```
servicios en el camino asfaltado            11 de 15
servicios fuera, con salida declarada         4 de 15
```

Y los cuatro motivos, que es lo que valía la información:

```
servicio de análisis    necesita una imagen base con bibliotecas científicas
                        que el camino no ofrecía
motor de precios        necesita despliegue en dos regiones activas y el
                        camino solo hacía una
portal antiguo          no está en contenedor y no lo va a estar
integraciones           usa una canalización distinta por requisito de un cliente
```

Los dos primeros entraron en el trabajo pendiente y se resolvieron; sus equipos volvieron al camino sin que nadie se lo pidiera. El tercero no se va a resolver nunca y está bien que sea así. El cuarto reveló algo distinto: usaba otra canalización y aun así cumplía las tres obligaciones —identidad federada, cifrado, firma verificada—, porque lo obligatorio eran los controles y no la herramienta.

```
en el camino asfaltado                   11 de 15   14 de 15
salidas por carencia de la plataforma          2          0
salidas legítimas y permanentes                2          1
```

El error que costó dos meses de confianza: una versión retirada sin aviso.

```
semana 22: se publica la versión 2 de la plantilla de canalización
           y se retira la 1 en la misma semana
```

```
equipos afectados                                    9
horas de trabajo no planificado                    ~60
equipos que dijeron que volverían a su canalización  3
```

Se escribió el contrato de la plataforma:

```
dos versiones soportadas a la vez
90 días de aviso antes de retirar
y la plataforma escribe la migración
```

La migración siguiente se hizo con una herramienta que abría el cambio propuesto en cada repositorio:

```
horas de trabajo por equipo                  6,5              0,4
equipos migrados en plazo                    5 de 9           9 de 9
quejas                                          9                0
```

Lo que costó y lo que ahorró, al año.

```
coste del equipo de plataforma           4 personas
tiempo ahorrado a los equipos de producto
  creación de servicios (28 al año)      28 × 4,3 días  ≈ 120 días
  accesos y entornos                                    ≈  95 días
  migraciones con herramienta                           ≈  55 días
  incidentes evitados por puertas y bucle               no cuantificado
total estimado                                          ≈ 270 días-persona
```

Y la medida que el apartado segundo señala como la que justifica la existencia:

```
proporción del trabajo de un equipo de producto
dedicada a infraestructura                    31 %            9 %
```

Al año.

```
peticiones al mes                               118             19
tiempo del equipo de plataforma en peticiones    84 %           21 %
crear un servicio nuevo hasta dev             6 días          22 min
adopción voluntaria del camino                   —          14 de 15
salidas por carencia                             —               0
contrato de versiones                         no había        2 y 90 días
migraciones con herramienta                   0 de 2          3 de 3
componentes retirados por falta de uso           —               5
```

La lección que esta clase traslada al resto de la parte 08: la fase 1 y la fase 2 tenían el mismo equipo, el mismo presupuesto y las mismas herramientas. Lo que cambió fue dejar de atender peticiones durante seis semanas, que es lo único que permitía construir lo que las eliminaba. Y la información más valiosa del año no la dio la adopción, sino su contrario: los cuatro equipos que se salieron señalaron dos carencias reales, una decisión permanente correcta y un caso que demostró que lo obligatorio deben ser los controles y no las herramientas.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/106-platform-engineering-e-internal-developer-platform/lab.py
```

El laboratorio selecciona el motor de práctica platform y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa mapa-capacidades-plataforma en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es una capacidad autoservicio con contrato y golden path. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye mapa-capacidades-plataforma para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
El equipo de plataforma no tiene tiempo de construir nada	Está atendiendo peticiones; es una ventanilla con otro nombre	Clasifica las peticiones, congela los tipos más repetidos y automatízalos; vigila que la atención no pase de un tercio del tiempo.
La adopción es del 100 % y las quejas son constantes	La plataforma es obligatoria, así que su fracaso no puede medirse	Ofrece salida explícita sin pedir permiso y trata cada motivo de salida como trabajo pendiente del producto.
Cada mejora de la plataforma interrumpe a quince equipos	No hay contrato de versiones ni aviso de retirada	Soporta dos versiones, avisa con antelación y escribe tú la herramienta de migración.
Cuando el portal falla, nadie puede hacer nada	La interfaz es la única forma de operar y lo que hay debajo no es usable	Que el portal escriba en el repositorio y que esa confirmación se pueda escribir a mano.
Se construyeron componentes que nadie usa	Se priorizó por criterio del equipo de plataforma, no por demanda	Observa a los usuarios trabajando, prioriza por lo que aparece en las peticiones y retira lo que no se usa.
Los equipos discuten cada semana qué es responsabilidad de quién	La frontera entre plataforma y producto no está escrita	Escribe el inventario de lo que ofrece la plataforma y la lista de lo que no; mantén la guardia del servicio en el equipo que lo construye.

■ Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

■ Preguntas de comprobación

1. ¿Qué distingue una plataforma de una ventanilla de peticiones?
2. ¿Por qué una plataforma obligatoria oculta su propio fracaso?
3. ¿Qué debe ser obligatorio y qué debe ser camino?
4. ¿Por qué lo que hay debajo de la interfaz tiene que ser usable directamente?
5. ¿Qué contiene el contrato de versiones de la plataforma y por qué la migración la escribe la plataforma?

Referencias

- Skelton, M. y Pais, M. (2019). Team Topologies, cap. 5 — equipo de plataforma y su relación con los equipos de producto. <https://teamtopologies.com/book>
- CNCF (2025). Platforms white paper — plataforma como producto, capacidades e interfaces. <https://tag-app-delivery.cncf.io/whitepapers/platforms/>
- Thoughtworks (2025). Platform as a product and the paved road — camino asfaltado, salida y medición de adopción. <https://www.thoughtworks.com/insights/blog/platforms>
- Backstage (2025). Software catalog and scaffolder — catálogo, plantillas y autoservicio. <https://backstage.io/docs/features/software-catalog/>
- Google (2025). Internal developer platforms and golden paths — estandarizar lo que no diferencia. <https://cloud.google.com/architecture/devops>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 08 en PDF · Manual integral

Anterior	Índice	Siguiente
← 105 · Feature flags y separación deploy-release	Parte 08 · Programa	107 · Developer experience, DORA y carga cognitiva →

Evaluación — 106 Platform engineering e Internal Developer Platform

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega mapa-capacidades-plataforma con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.

107 — Developer experience, DORA y carga cognitiva

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 4 Laboratorio: metrics · Estado: EXECUTABLECORE

Propósito

Comprobar si todo lo construido en las partes 07 y 08 cumple su promesa. Las cuatro medidas de entrega son la herramienta habitual, y son útiles leídas por parejas y como diagnóstico propio de un equipo a lo largo del tiempo. La clase enseña las cuatro con precisión, demuestra con casos concretos qué pasa cuando se convierten en objetivo, y añade lo que ninguna de las cuatro ve: dónde se va realmente el tiempo de quien desarrolla.

Resultados de aprendizaje

Al finalizar podrás:

1. Definir las cuatro medidas con precisión, incluido desde dónde se miden.
2. Leerlas por parejas, porque cada una sin su contraria se degrada.
3. Reconocer cómo se falsea cada una cuando se convierte en objetivo.
4. Medir la fricción real, que es donde suele estar la mayor mejora.
5. Usarlas como diagnóstico propio y no como comparación entre equipos.

Conceptos centrales

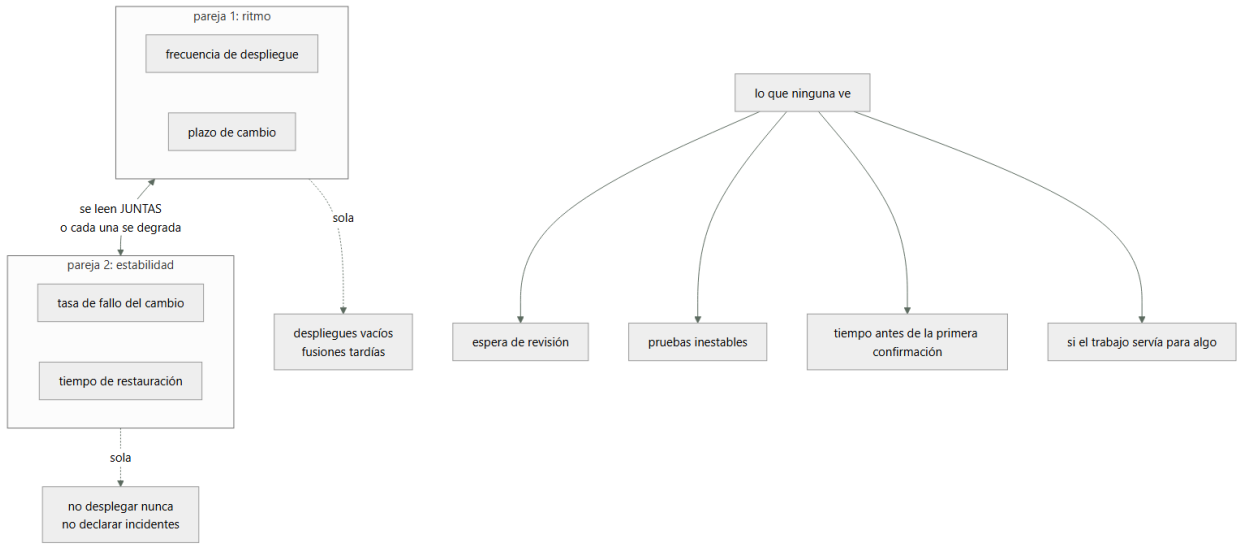
Concepto	Comprensión verificable
frecuencia de despliegue	Cada cuánto llega a producción un cambio. Mide la capacidad de entregar en incrementos pequeños.
plazo de cambio	Tiempo desde que el código se confirma hasta que está en producción. No incluye lo anterior a la confirmación, y por eso oculta la mitad del problema.
tasa de fallo del cambio	Proporción de despliegues que provocan una degradación que exige actuar. Es la contraparte de la frecuencia.
tiempo de restauración	Desde que la degradación empieza hasta que se restablece el servicio. Es la contraparte del plazo.
ley de Goodhart	Cuando una medida se convierte en objetivo, deja de ser una buena medida. Las cuatro son especialmente fáciles de falsear.
fricción	Tiempo que se pierde esperando, repitiendo o rehaciendo. No aparece en ninguna de las cuatro y suele ser la mayor palanca.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. Las cuatro, con precisión

La definición importa más que el nombre, porque casi todas las discusiones vienen de medir cosas distintas con la misma palabra.

```
FRECUENCIA DE DESPLIEGUE
  despliegues a producción por unidad de tiempo, por servicio
  se cuenta el despliegue, no la fusión
  se cuenta por servicio, no por organización
     → sumar quince servicios da un número grande y sin significado

PLAZO DE CAMBIO
  desde la confirmación hasta que ese código sirve tráfico
  mediana y percentil 90, no media
     → la media la domina el caso raro

TASA DE FALLO DEL CAMBIO
  despliegues que provocan degradación que exige actuar
     (revertir, arreglar en caliente, apagar un interruptor)
  dividido entre despliegues totales

TIEMPO DE RESTAURACIÓN
  desde que la degradación empieza —no desde que se detecta—
  hasta que el servicio se restablece
```

Y la línea de «no desde que se detecta» es la que la hace honesta: si se mide desde la detección, mejorar la medida es tardar más en detectar.

Y el mayor punto ciego del plazo, que conviene decir pronto:

```
idea → escritura → PRIMERA CONFIRMACIÓN → revisión → fusión → producción
                   ■■■■■■■■ lo que el plazo mide ■■■■■■■■■■■■■■■■■■■■■■
        ■■■ lo que NO mide, y suele ser mayor ■■■
```

En organizaciones donde la canalización ya funciona, la mayor parte del tiempo total está antes de la primera confirmación o esperando revisión. Optimizar la canalización cuando el problema está ahí es trabajar en el sitio equivocado.

Y una nota sobre los rangos que se publican como referencia: sirven para saber en qué orden de magnitud se está, no como objetivo. Un servicio interno que se usa dos veces al día no gana nada desplegando cincuenta veces.

2. Por qué van por parejas

El valor de estas cuatro medidas no está en cada una: está en que cada pareja restringe a la otra.

```
frecuencia y plazo          cuánto ritmo
tasa de fallo y             a qué precio
tiempo de restauración
```

Y el motivo es que cada una, sola, tiene una forma trivial de mejorarse que empeora el sistema:

```
solo frecuencia          desplegar cambios vacíos
solo plazo               fusionar sin revisar
solo tasa de fallo       no desplegar
solo restauración        no declarar incidentes
```

Y el hallazgo que hace interesantes a las cuatro juntas —y que este programa ha ido confirmando en varias clases— es que no están en conflicto. Lo que mejora el ritmo suele mejorar la estabilidad, por una razón mecánica:

```
desplegar más veces → cada despliegue lleva menos cambios
                    → cuando algo falla, hay menos donde buscar
                    → y revertir cuesta menos
```

Es lo mismo que la clase 097 dijo del tamaño del incremento y lo que la clase 102 midió al quitar la aprobación para revertir.

Y conviene una quinta medida que las cuatro no cubren y que decide si el equipo puede sostener el ritmo:

```
trabajo no planificado como proporción del total
  → si sube, el sistema está devolviendo el ritmo en forma de incidentes
```

Y una advertencia sobre la agregación, que es un error frecuente y difícil de deshacer:

```
mal   una cifra de la organización, comparando equipos
bien  la evolución de cada servicio contra sí mismo
```

Dos servicios con perfiles distintos —uno de cara al cliente y otro por lotes— no son comparables, y el momento en que estas medidas aparecen en una comparación entre equipos es el momento en que dejan de servir para diagnosticar.

3. Qué pasa cuando se convierten en objetivo

Este apartado es la parte práctica de la ley de Goodhart, y conviene tenerla escrita porque los cuatro casos aparecen de verdad.

```
OBJETIVO: subir la frecuencia de despliegue
  se despliega el mismo artefacto varias veces
  se parte un cambio en tres despliegues sin motivo
  se cuentan los despliegues a dev
  → la cifra sube y no llega más valor a nadie

OBJETIVO: bajar el plazo de cambio
  se retrasa la primera confirmación hasta tenerlo casi listo
     → el reloj empieza más tarde y el trabajo total es el mismo
  se abre el cambio propuesto ya aprobado de antemano
  → la cifra baja y la espera se ha movido a donde no se mide

OBJETIVO: bajar la tasa de fallo
  se redefine qué cuenta como fallo
  se arregla en caliente sin declarar nada
  se despliega menos veces, con más cambios juntos
  → la cifra baja y el riesgo por despliegue sube

OBJETIVO: bajar el tiempo de restauración
  se cierra el incidente cuando se aplica la mitigación
  se empieza a contar desde la detección
  → la cifra baja y el usuario sigue afectado
```

La tercera es la más dañina de las cuatro, porque mueve el sistema en dirección contraria a todo lo que las partes 07 y 08 han construido.

Y las tres reglas que evitan casi todo esto:

```
1. las cuatro se publican siempre juntas, nunca una sola
2. no se fija objetivo numérico; se fija una dirección y se revisa la causa
3. quien las mira pregunta «¿qué lo impide?», no «¿por qué no llegas?»
```

Y una cuarta, más concreta, que corta el falseo de raíz: la definición de cada medida se escribe y se versiona, y cambiarla exige el mismo trámite que cambiar cualquier otra cosa. Casi todos los saltos bruscos de estas cifras vienen de un cambio de definición, no de un cambio de comportamiento.

Y una comprobación honesta que conviene hacer cada cierto tiempo:

```
¿la mejora de la cifra se corresponde con algo que la gente note?
  si el plazo bajó a la mitad y nadie del equipo lo percibe,
```

```
probablemente se movió el punto de medida
```

4. Dónde se va el tiempo de verdad

Las cuatro medidas describen el sistema de entrega. No describen el día de quien desarrolla, y ahí suele estar la palanca mayor.

Lo que se mide, y con qué instrumento:

```
espera de revisión          del momento de pedirla al primer comentario
                            → suele ser la mayor partida, y es barata de arreglar
pruebas inestables          proporción de ejecuciones que fallan y pasan al repetir
                            → destruye la confianza en las puertas (clase 100)
tiempo de bucle local       guardar → ver el efecto
                            → si pasa de un minuto, cambia cómo se trabaja
tiempo de la canalización   clase 097
cambios de contexto         interrupciones por día
búsqueda de información     cuánto cuesta averiguar cómo se hace algo
```

Y el instrumento para las dos últimas no son los registros: son las personas. Una pregunta periódica y corta, con la misma redacción cada vez, da una serie comparable:

```
«¿cuánto de tu semana se fue en esperar o en rehacer?»
«¿qué te ha frenado más esta semana?»
```

Y la segunda pregunta, abierta, es la que produce la lista de trabajo pendiente de la plataforma de la clase 106.

Y una advertencia sobre las pruebas inestables, porque tienen un efecto que no se ve en ninguna cifra de tiempo:

```
si el 8 % de las ejecuciones falla sin motivo
→ la gente repite sin mirar
→ y cuando el fallo es real, también lo repite
→ la puerta ha dejado de ser puerta (clase 100)
```

Y lo que ninguna medida de esta clase captura, y hay que decirlo:

```
si lo entregado sirvió para algo
si lo que se construyó era lo que hacía falta
si el equipo puede sostener este ritmo el año que viene
```

Un equipo puede tener las cuatro medidas excelentes entregando muy deprisa algo que nadie quiere. Las cuatro miden la máquina, no la dirección.

Y la lista de comprobación de la clase:

- [ ] las cuatro definiciones están escritas, versionadas y son estables
- [ ] el plazo se mide desde la confirmación, y se sabe qué queda fuera
- [ ] la restauración se cuenta desde que empieza, no desde que se detecta
- [ ] se miden por servicio, no agregadas por organización
- [ ] se publican siempre las cuatro juntas
- [ ] no hay objetivo numérico por equipo ni comparación entre equipos
- [ ] se sigue el trabajo no planificado como quinta medida
- [ ] se mide la espera de revisión y la inestabilidad de las pruebas
- [ ] hay una pregunta abierta y periódica sobre qué frena al equipo
- [ ] se comprueba que la mejora de la cifra la nota alguien

Y el cierre que enlaza con la clase siguiente: con esto termina el material de la parte 08. La clase 108 monta la fábrica completa sobre los tres proveedores y, sobre todo, califica la hipótesis escrita al cerrar la parte 07, incluidas las partes en las que se equivocó.

Ejemplo trabajado

CloudShop mide las cuatro durante catorce meses. El ejercicio tiene tres momentos: lo que enseñó la primera medición, lo que pasó cuando alguien fijó un objetivo, y dónde estaba de verdad la mayor mejora.

Primera medición, mes 1.

```
frecuencia de despliegue      1,2 / semana        —
plazo de cambio                    3,1 días    11 días
tasa de fallo del cambio             14 %        —
tiempo de restauración             2,4 h       9,1 h
trabajo no planificado               38 %        —
```

Y al descomponer el plazo, que es lo que las cifras agregadas ocultaban:

```
confirmación → petición de revisión                4 h
petición de revisión → primer comentario        31 h   ← 42 % del total
primer comentario → aprobación                  18 h
aprobación → fusión                              2 h
fusión → producción                             19 h
                                               ■■■■■■■
                                                74 h ≈ 3,1 días
```

Treinta y una horas esperando a que alguien mirara. La canalización, que era donde todo el mundo miraba, era el 26 % del plazo.

Lo que se hizo con eso, y lo que costó.

```
compromiso de revisión en menos de 4 h laborables
rotación diaria de quién revisa primero
aviso automático a los 2 h sin revisar
límite de tamaño: por encima de 400 líneas se pide partirlo
```

```
espera hasta primer comentario         31 h      3,5 h
plazo de cambio (mediana)            3,1 días   1,1 días
tamaño mediano del cambio            310 lín.   140 lín.
```

Y sin tocar la canalización. La segunda mayor partida fueron las pruebas inestables:

```
ejecuciones que fallan y pasan al repetir  8,1 %    0,9 %
reintentos por semana                       47         6
cambios fusionados con la puerta en rojo    11         0
```

La última línea es la importante: once cambios se habían fusionado saltándose una puerta roja porque «seguro que es la prueba inestable». La inestabilidad no cuesta tiempo: cuesta la puerta.

Mes 7: alguien fija un objetivo, y las cuatro se mueven.

Se anunció un objetivo trimestral: «tasa de fallo del cambio por debajo del 5 %». En dos meses:

```
tasa de fallo del cambio            9 %       4,1 %   ← objetivo cumplido
frecuencia de despliegue        4,1 / sem   2,2 / sem
cambios por despliegue              1,8       4,3
incidentes declarados              12/mes     5/mes
incidentes en el registro de
guardia sin declarar                1/mes     9/mes   ← aquí está la respuesta
tiempo de restauración             48 min     2,1 h
```

La cifra bajó a la mitad y el sistema empeoró en todo lo demás: se declararon menos incidentes y se juntaron más cambios por despliegue. Las dos formas de falsear que el apartado tercero anticipa, las dos a la vez, y ninguna deliberada: la gente responde al incentivo sin proponérselo.

Se retiró el objetivo numérico y se sustituyó por las tres reglas:

```
tasa de fallo del cambio           4,1 %       6,2 %   ← sube, y es más honesta
frecuencia de despliegue       2,2 / sem   5,8 / sem
incidentes sin declarar            9/mes       0/mes
tiempo de restauración             2,1 h      37 min
```

La tasa de fallo subió y todo lo demás mejoró. Es el argumento más claro de por qué las cuatro se publican juntas.

Lo que las cuatro no vieron.

La pregunta abierta semanal —«¿qué te ha frenado más esta semana?»— produjo, en seis meses:

```
esperar revisión                          41 menciones  → resuelto arriba
no encontrar cómo se hace algo            33            → catálogo (106)
pruebas inestables                        28            → resuelto arriba
entorno local lento                       19            → efímeros (104)
esperar a otro equipo                     17            → sin resolver
no saber si lo que hago se usa             9            → sin resolver
```

Las dos últimas no se resolvieron y siguen en la lista. Y la última es la que ninguna medida de esta clase puede capturar: el sistema de entrega no dice si lo entregado servía para algo.

A los catorce meses.

```
frecuencia de despliegue             1,2 / sem     6,4 / sem
plazo de cambio (mediana)             3,1 días       9 h
plazo de cambio (p90)                   11 días      31 h
tasa de fallo del cambio                   14 %       6,2 %
```

```
tiempo de restauración                  2,4 h       37 min
trabajo no planificado                   38 %         19 %
espera hasta primer comentario           31 h        3,5 h
pruebas inestables                      8,1 %        0,9 %
incidentes sin declarar                 1/mes        0/mes
objetivos numéricos por equipo            1            0
```

La lección que esta clase traslada al resto de la parte 08: la mejora mayor del plazo —de 3,1 días a 1,1— no vino de nada construido en las partes 07 y 08. Vino de comprometerse a revisar en cuatro horas y de partir los cambios. La canalización, que era donde estaba toda la atención, era el 26 % del problema. Y el episodio del mes 7 deja la advertencia con datos: un objetivo numérico sobre una sola de las cuatro consiguió su cifra y empeoró el sistema, y lo hizo sin que nadie tuviera intención de falsear nada.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/107-developer-experience-dora-y-carga-cognitiva/lab.py
```

El laboratorio selecciona el motor de práctica metrics y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa tablero-dora en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es métricas definidas, consultables y vinculadas a una decisión. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye tablero-dora para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

Errores frecuentes

Síntoma	Causa probable	Corrección
Una medida mejora mucho y nadie del equipo nota ninguna diferencia	Se movió el punto desde el que se mide, no el comportamiento	Escribe y versiona las definiciones; sospecha de todo salto brusco y comprueba si coincide con un cambio de definición.
La tasa de fallo baja y los incidentes reales no bajan	Se declaran menos incidentes o se despliega menos veces con más cambios juntos	Publica siempre las cuatro juntas y sigue el número de incidentes sin declarar en el registro de guardia.
Se optimiza la canalización durante meses y el plazo apenas mejora	La mayor parte del plazo está en la espera de revisión, no en la canalización	Descompón el plazo por tramos antes de decidir dónde trabajar.
Los equipos se comparan y los que peor salen dejan de reportar	Las medidas se usan como evaluación en vez de como diagnóstico	Mide cada servicio contra su propia evolución y prohíbe la comparación entre equipos y personas.
Se fusionan cambios con una puerta en rojo	Las pruebas inestables han destruido la confianza en la señal	Mide la proporción de ejecuciones que fallan y pasan al repetir, y trátala como defecto prioritario.
Las cuatro medidas son excelentes y el producto no avanza	Miden la máquina de entrega, no si lo entregado servía	Complementa con medidas de resultado y con una pregunta abierta y periódica al equipo.

Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

Preguntas de comprobación

1. ¿Desde qué momento se mide el plazo de cambio y qué queda fuera?
2. ¿Por qué el tiempo de restauración se cuenta desde que empieza la degradación y no desde la detección?
3. ¿Qué forma trivial de mejorar tiene cada medida por separado?
4. ¿Por qué no deben fijarse objetivos numéricos por equipo sobre estas medidas?
5. ¿Qué mide la inestabilidad de las pruebas que no aparece en ninguna de las cuatro?

Referencias

- Forsgren, N., Humble, J. y Kim, G. (2018). Accelerate, caps. 2 y 4 — definición de las cuatro medidas y su relación. <https://itrevolution.com/product/accelerate/>
- DORA (2025). State of DevOps report — evolución de las medidas y advertencias sobre su uso. <https://dora.dev/research/>
- Forsgren, N. y otros (2021). The SPACE of developer productivity — por qué una sola dimensión no basta. <https://queue.acm.org/detail.cfm?id=3454124>
- Google (2025). Engineering productivity research: flaky tests — coste de la inestabilidad sobre la confianza en las puertas. <https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html>
- Strathern, M. (1997). Improving ratings: audit in the British university system — formulación de la ley de Goodhart aplicada a indicadores. <https://doi.org/10.1111/1468-0009.00035>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 08 en PDF · Manual integral

Evaluación — 107 Developer experience, DORA y carga cognitiva

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega tablero-dora con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.

108 — Proyecto: fábrica de software multi-cloud

Parte: 08 — Entrega continua y platform engineering Nivel: avanzado · Horas estimadas: 8 Laboratorio: capstone · Estado: EXECUTABLECORE

Propósito

Montar la fábrica completa —del cambio propuesto a producción en los tres proveedores— con todo lo de las clases 097 a 107 funcionando a la vez, y comprobar dónde encaja mal. La clase cierra la parte 08 con las tres piezas de siempre: calificar la hipótesis escrita al terminar la parte 07, incluidas las partes en las que se equivocó, actualizar el recuento de leyes con la que esta parte ha hecho aparecer cuatro veces, y escribir la predicción que la parte 09 tendrá que corregir.

Resultados de aprendizaje

Al finalizar podrás:

1. Montar la fábrica completa y localizar sus costuras.
2. Adaptar el mismo camino a AWS, Azure y Google Cloud sin duplicarlo tres veces.
3. Calificar con evidencia la hipótesis de la parte 07, incluido lo que falló.
4. Incorporar la ley 17 al cuestionario, con sus cuatro apariciones.
5. Escribir la predicción de la parte 09 en términos que se puedan desmentir.

Conceptos centrales

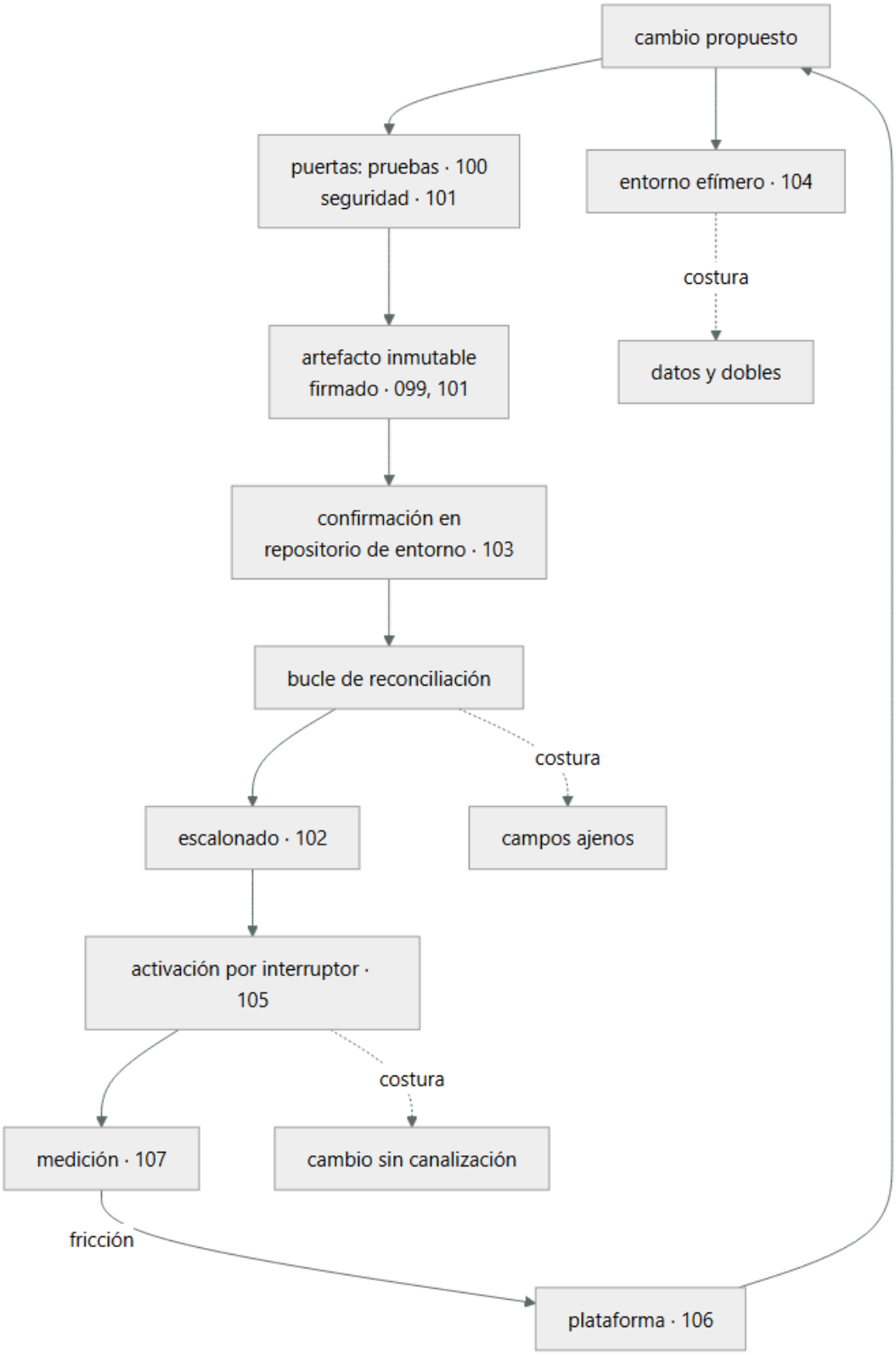
Concepto	Comprensión verificable
fábrica	El recorrido completo del cambio: propuesta, entorno, puertas, artefacto firmado, promoción por huella, reconciliación, activación y medición.
costura	Punto donde dos mecanismos de la fábrica se tocan y ninguno es dueño. Es donde aparecen los fallos que ninguna clase suelta predice.
camino común	La parte del recorrido que no depende del proveedor. Lo que sí depende se aísla en el borde, no se replica tres veces.
calificación de hipótesis	Comparar lo que se predijo con lo que ocurrió, publicando también lo que se predijo mal. Es lo que convierte la parte en conocimiento y no en opinión.
ley 17	Toda medida que se convierte en objetivo se alcanza; el sistema que medía no tiene por qué mejorar.
hipótesis de la parte 09	Predicción escrita ahora, sobre lo que ocurrirá cuando el sujeto pase a ser el estado, para que la clase 120 la corrija con datos.

Modelo mental

Una plataforma interna ofrece capacidades como productos y reduce carga cognitiva mediante caminos dorados, sin quitar autonomía donde importa.

Aplicado a esta clase, separa siempre cuatro planos: intención (qué necesita el usuario), configuración (qué declaramos), estado observado (qué existe de verdad) y evidencia (cómo sabemos que cumple). Confundirlos produce diseños que se ven correctos en un diagrama pero fallan al operar.

■ Flujo de razonamiento



Desarrollo

1. La fábrica, y sus tres costuras

Las once clases anteriores construyeron piezas. Juntas forman un recorrido:

```
cambio propuesto
  → entorno efímero con su nombre y dirección              104
  → puertas: pruebas por nivel, seguridad, contrato        100, 101
```

```
  → artefacto inmutable, firmado, con inventario          099, 101
  → confirmación en el repositorio de entorno             103
  → el bucle lo materializa                               103
  → escalonado con análisis y reversión automática        102
  → activación por interruptor, gradual                   105
  → medición del recorrido y de la fricción               107
  → y lo que la medición descubre entra en la plataforma  106
```

Y lo que no predice ninguna clase suelta son las costuras: los puntos donde dos piezas se tocan y ninguna es dueña.

```
COSTURA 1   entorno efímero ←→ datos y dependencias
  el entorno se crea en minutos y los datos deciden si sirve
  → y ahí no manda ni la plataforma ni el equipo: manda el esquema

COSTURA 2   bucle ←→ otros controladores
  el bucle quiere converger y el autoescalador también
  → 196 de 209 diferencias eran de esta costura (clase 103)

COSTURA 3   interruptor ←→ todo lo demás
  cambiar un valor tiene efecto de producción y no pasa por la fábrica
  → hubo que reconstruirle cuatro controles (clase 105)
```

Y una cuarta que aparece al montarlo todo junto y que conviene anticipar: el artefacto es el mismo en los tres entornos y el comportamiento no, porque los interruptores están en estados distintos. La garantía de la clase 099 sigue siendo cierta y ya no basta; lo que hay que poder responder es:

```
¿qué artefacto corre?      huella                       099
¿con qué configuración?    confirmación del entorno     103
¿con qué variantes?        estado de los interruptores  105
```

Las tres respuestas juntas son lo que identifica un sistema en ejecución. Con dos no basta, y casi todo el mundo se queda en dos.

2. El proyecto

Montar la fábrica para un servicio real en los tres proveedores. Lo que hay que entregar:

```
1. CAMINO COMÚN
   una plantilla de canalización reutilizable con las puertas de las
   clases 100 y 101, y el presupuesto de ruido declarado
   → lo específico de cada proveedor, aislado en el último paso

2. IDENTIDAD SIN CLAVES en los tres
   federación desde el flujo hacia AWS, Azure y Google Cloud   098
   → inventario que demuestre que no queda ninguna clave de larga duración

3. ARTEFACTO ÚNICO
   una imagen, tres registros, la misma huella
   firmada con la identidad del flujo, con inventario y procedencia
   → y verificación en admisión que compruebe flujo y rama

4. REPOSITORIO DE ENTORNO
   tres entornos por proveedor, declarados por huella
   propiedad de campo resuelta, poda con umbral y protecciones   103

5. ESCALONADO
   canario con control simultáneo y primer escalón dimensionado   102
   prueba con una versión rota a propósito, que debe pararla

6. ENTORNO EFÍMERO
   por cambio propuesto, con semilla versionada y dobles por contrato
   caducidad y barrido de huérfanos                            104

7. INTERRUPTORES
   uno de operación por cada camino que cruce un punto de no retorno
   con registro, gradualidad, reversión y fecha de muerte      105

8. MEDICIÓN
   las cuatro medidas por servicio, con sus definiciones escritas
   y el plazo descompuesto por tramos                          107
```

Y las preguntas cuya respuesta hay que escribir, porque son las que separan un montaje de una fábrica:

```
¿cuánto tarda un cambio de una línea desde la confirmación
```

```
 hasta servir tráfico en producción, medido?
¿qué pasa si el bucle deja de sincronizar? ¿cuánto tarda alguien en saberlo?
¿qué pasa si el registro de imágenes no responde durante un despliegue?
¿quién puede revertir, y necesita permiso de alguien?
¿qué parte del camino es distinta en cada proveedor, y por qué?
```

Y sobre la última: lo que de verdad cambia entre proveedores es menos de lo que parece. La construcción, las puertas, la firma, el inventario, el repositorio de entorno y el escalonado son iguales. Lo que cambia es la federación de identidad, el nombre del registro y el servicio de secretos.

```
idéntico entre proveedores      construcción, puertas, artefacto, firma,
                                repositorio de entorno, escalonado, medición
distinto                        federación de identidad
                                registro de imágenes
                                servicio de secretos
                                admisión (verificación de firma)
```

Y el error a evitar es el de siempre: tres canalizaciones parecidas que divergen en seis meses. Una plantilla con un paso final parametrizado, y ninguna copia.

3. Calificación de la hipótesis de la parte 07

Al cerrar la parte 07, la clase 096 escribió dos predicciones. Las dos se califican aquí con lo que las clases 097 a 107 encontraron.

Predicción 1: «cuando el bucle exista, la canalización pasará a ser el objetivo más valioso».

```
veredicto: acertada en la conclusión, EQUIVOCADA en el mecanismo
```

Se predijo que lo sería porque despliega. Y ocurrió lo contrario en esa dimensión concreta: al pasar al modelo de tirar, la canalización perdió las credenciales del clúster (clase 103). Dejó de poder desplegar.

Y siguió siendo el objetivo más valioso por otro camino que no se había previsto: es quien escribe las confirmaciones que el bucle obedece sin preguntar.

```
la canalización puede desplegar          sí                no
la canalización puede hacer que
se despliegue lo que quiera              sí                sí
la revisión humana lo impide             no había          en producción
```

El bucle no redujo el valor del objetivo: cambió qué hay que proteger. Antes, unas credenciales; después, el derecho a confirmar en un repositorio, que es exactamente lo que la clase 098 llamó el permiso de escritura como puerta de entrada.

Predicción 2: la lista de lo que el bucle no debe revertir.

La clase 096 escribió tres elementos. Lo que la clase 103 midió al activarlo:

```
réplicas fijadas por el autoescalador          sí          61
anotaciones inyectadas por operadores          no          88   ← la mayor
campos rellenados por el sistema               no          47
certificados rotados                           sí           0   (no ocurrió aún)
cambios manuales legítimos de incidente        sí           1

veredicto: la lista era correcta y estaba INCOMPLETA en su mayor partida
```

De las 209 diferencias, la categoría más grande —88, el 42 %— era la que no estaba en la lista. Y el motivo del error es instructivo: se predijo pensando en quién cambia el número de réplicas, y no en qué escribe cosas en los recursos sin que nadie lo pida. La malla de servicio y los operadores anotan constantemente, y son invisibles hasta que algo compara.

Y hay un acierto que conviene registrar porque fue el más útil: la predicción de que haría falta un camino de emergencia que no consistiera en desactivar el bucle. Se cumplió en la semana 8, y el único bucle que se desactivó estuvo apagado seis días.

Y lo que la parte 08 encontró y ninguna predicción contemplaba:

```
la mayor mejora del plazo no vino de la fábrica
  vino de comprometerse a revisar en 4 horas: 3,1 días → 1,1 días
  la canalización era el 26 % del plazo                       clase 107

renunciar al realismo de los datos mejoró la detección
  la semilla pequeña encontró 5 defectos que el extracto no contenía
```

```
                                                        clase 104

la información más valiosa de la plataforma la dieron
  los equipos que NO la usaban                          clase 106
```

4. La ley 17, y el recuento

Una regularidad ha aparecido cuatro veces en esta parte, en contextos independientes, y ya cumple el criterio para entrar en el cuestionario.

```
LEY 17
  Toda medida que se convierte en objetivo se alcanza.
  El sistema que esa medida describía no tiene por qué mejorar.
```

Sus cuatro apariciones:

```
clase 100   cobertura del 91 % como objetivo
            → 33 % de mutantes sobrevivían: pruebas que ejecutaban sin comprobar

clase 106   plataforma obligatoria
            → adopción del 100 % por definición, y el fracaso invisible

clase 107   objetivo de tasa de fallo por debajo del 5 %
            → se cumplió (4,1 %) y se declararon 9 incidentes menos al mes

clases 067  «escáner implantado» como medida de cumplimiento
      y 091  → implantado y desactivado durante 8 y 14 meses
```

Y lo que la ley 17 añade al cuestionario de cada tecnología nueva:

```
¿qué medida usaremos para decir que esto funciona?
¿cuál es la forma más barata de mejorar esa medida sin mejorar nada?
¿qué medida contraria la sujeta?
```

La tercera pregunta es la única defensa que ha funcionado en las cuatro apariciones: la cobertura sujeta con mutación, la frecuencia con la tasa de fallo, la adopción con el motivo de las salidas, el «implantado» con los hallazgos por cambio.

Recuento tras la parte 08. Las cinco leyes con más apariciones acumuladas:

```
ley 13  en un sistema declarativo, el bucle que no corre no da error      10
        parte 08: agente parado 9 días (103), entorno huérfano (104),
                  interruptor que nadie retira (105)

ley 15  una señal con demasiados elementos deja de ser señal               9
        parte 08: 812 hallazgos (101), 209 diferencias (103),
                  147 interruptores (105)

ley 16  un control que estorba acaba desactivado o rodeado                 9
        parte 08: escáner desactivado (101), entorno de 34 min sin uso (104),
                  puerta roja rodeada por pruebas inestables (107)

ley 11  lo que entra en un sistema de solo-añadir se queda                 6
        parte 08: secretos en el historial (101)

ley 14  las decisiones de creación son irreversibles                       6
        parte 08: punto de no retorno del despliegue (102),
                  poda que borra lo que ya no se declara (103)

ley 17  la medida que se vuelve objetivo se alcanza sin mejorar el sistema  4
        NUEVA en esta parte
```

Y una observación sobre las tres primeras: aparecen juntas. Un bucle que no corre (13) produce una señal permanente que nadie mira (15) y acaba desactivado (16). En la parte 08 esa cadena se ha visto entera tres veces.

5. La hipótesis de la parte 09

Todo lo construido en las partes 05 a 08 funciona por una razón que casi nunca se enuncia: lo que se despliega no guarda estado. El artefacto inmutable, la reversión, el entorno efímero, la poda, el canario y el interruptor son mecanismos que se apoyan en poder destruir y recrear sin perder nada.

La parte 09 cambia el sujeto: bases relacionales y no relacionales, caché, almacenamiento de objetos, colas, flujos, eventos y orquestación. El estado pasa a ser el tema.

La predicción, escrita para poder desmentirla:

1. De los ocho mecanismos de las partes 07 y 08, TRES O MENOS se aplicarán sin cambios a un componente con estado.
   Los mecanismos: artefacto inmutable, reversión, entorno efímero, reconciliación con poda, canario, interruptor, puerta de canalización, camino asfaltado.
   → la clase 120 dirá cuántos sobrevivieron y cuáles necesitaron una versión distinta

2. La ley 14 —las decisiones de creación son irreversibles— será la ley dominante de la parte 09, con más apariciones que la ley 13.
   Motivo: en los servicios de datos, la clave de partición, el modo de consistencia, el número de particiones y el formato de almacenamiento se eligen al crear y no se cambian.

3. El problema más difícil de la parte 09 no será guardar los datos.
   Será la GARANTÍA EN LA FRONTERA: qué ocurre exactamente una vez, qué llega en orden y qué pasa cuando se reintenta.
   → y predigo que la respuesta recurrente será la misma que ya usa este programa para otra cosa: hacer la operación repetible sin efecto adicional, en vez de intentar que ocurra una sola vez

4. Y una predicción concreta sobre la poda: será PELIGROSA de una forma nueva. Un recurso con estado que se borra por no estar declarado no se recrea igual: se recrea vacío.

Y lo que hay que anotar ahora para poder calificar honestamente:

lo que ya sabemos que se rompe	reversión, cuando se cruza el punto de no retorno (clase 102)
lo que creemos que aguanta	artefacto inmutable, puerta, camino
lo que no tenemos ni idea	canario sobre un cambio de esquema

La tercera línea es la que hace que valga la pena escribir esto: si la clase 120 no corrige nada, la predicción era demasiado cómoda.

Ejemplo trabajado

Se monta la fábrica completa para el servicio de pedidos de CloudShop, en AWS, Azure y Google Cloud a la vez. Lo interesante no es que funcione: es qué parte resultó ser común, qué costó más y qué se rompió al juntarlo todo.

Lo que resultó común, medido en líneas.

plantilla de canalización	410	sí
definición de puertas	180	sí
construcción y firma	140	sí
repositorio de entorno (base)	260	sí
definición del escalonado	95	sí
	■■■■■	
	1.085	

federación de identidad	3 × 40	no
registro de imágenes	3 × 12	no
servicio de secretos	3 × 25	no
admisión y verificación de firma	3 × 30	no
	■■■■■	
	321	

El 77 % del código es común. El primer diseño replicaba la canalización entera por proveedor; la segunda versión aisló los 321 en un paso final parametrizado.

líneas totales	3.255	1.406
cambios que hay que hacer 3 veces	todos	0
divergencia a los 6 meses (proyectada)	alta	nula

Lo que se rompió al juntarlo: cinco costuras.

1. el entorno efímero creaba una base de datos por cambio
   → 9 entornos × 3 proveedores = 27 bases vivas
   → coste proyectado 2.400 €/mes
   corrección: una instancia compartida con un esquema por entorno

2. la huella de la imagen es la misma; los tres registros son distintos

```
   → el repositorio de entorno declaraba el registro, no solo la huella
   → y una promoción a otro proveedor no era una promoción
   corrección: la huella en la base y el registro en la capa del entorno

3. la verificación de firma funcionaba en dos proveedores y en el tercero
   la admisión no tenía el mismo mecanismo
   corrección: verificación previa en la canalización + admisión donde
               exista; documentado como diferencia real, no tapado

4. el canario comparaba con un control, y en un proveedor el reparto de
   tráfico no era estable a bajos porcentajes
   → el 5 % real oscilaba entre el 2 % y el 9 %
   corrección: primer escalón al 10 % en ese proveedor, con la misma
               cuenta de eventos del apartado de la clase 102

5. los interruptores tenían tres inventarios, uno por proveedor
   → el mismo interruptor en estados distintos, y nadie lo veía
   corrección: un único sistema de interruptores, consultado por los tres
```

La quinta es la que más se parece a la costura 3 del primer apartado, y confirma la regla: el estado del sistema en ejecución es artefacto, configuración e interruptores; con dos de los tres no se puede razonar.

La prueba de que la fábrica funciona: un cambio de una línea, cronometrado.

```
confirmación                                          0 s
puertas rápidas (secretos, código, dependencias)   1 min 40 s
pruebas de unidad e integración                    3 min 10 s
construcción, firma, inventario                    2 min 05 s
entorno efímero listo                              6 min 40 s   (en paralelo)
revisión humana                                   media 3,5 h
confirmación en repositorio de entorno               12 s
bucle materializa en dev                             48 s
promoción a pre (automática)                        1 min
promoción a pro (una aprobación)                  media 40 min
canario 10 % · 30 min                               30 min
escalones restantes                                 15 min
                                                  ■■■■■■■■■■■■■■■■■■■■■
   confirmación → producción, sin contar esperas humanas:   54 min
   con esperas humanas, mediana:                             5 h 10
```

Y la lectura que corresponde a la clase 107: de las 5 h 10, cuatro horas son espera humana. La fábrica es el 17 % del tiempo. Es exactamente el hallazgo del mes 1 de aquella clase, reproducido con la fábrica ya montada.

Las pruebas negativas, que es lo que distingue montar de tener.

```
versión rota a propósito → ¿la para el canario?          sí, minuto 11
agente parado → ¿alerta por antigüedad?                  sí, 31 min
confirmación que borra un directorio → ¿umbral de poda?  sí, se detuvo
secreto en el envío → ¿lo rechaza?                       sí
imagen firmada desde otra rama → ¿la rechaza admisión?   en 2 de 3
registro de imágenes caído → ¿qué pasa?                  el bucle no puede
                                                         materializar; los
                                                         pods existentes siguen
interruptor de operación con el sistema de interruptores
caído → ¿se puede apagar?                                sí, valor local
```

La quinta línea —dos de tres— es la única diferencia real entre proveedores que quedó sin resolver, y está documentada como tal en vez de omitida.

Estado final del servicio de pedidos.

```
frecuencia de despliegue                  1,2 / semana     6,4 / semana
plazo de cambio (mediana)                   3,1 días          9 h
tasa de fallo del cambio                      14 %           6,2 %
tiempo de restauración                       2,4 h          37 min
credenciales de larga duración                  7              0
imágenes en producción sin firma verificada     2              0
cambios manuales en el entorno              2 / mes          0 / mes
vulnerabilidades críticas en producción     no se sabía        0
interruptores vivos                             —             31
entornos efímeros huérfanos                     —            0-1
proporción del trabajo en infraestructura     31 %             9 %
```

Y la conclusión que cierra la parte: la fábrica hizo lo que prometía, y el mayor factor de los tiempos que quedan no está dentro de ella. De las 5 h 10 del recorrido, cuatro son espera de personas. Ese es el trabajo que la parte 08 deja abierto, y ninguna de las once clases anteriores lo puede resolver con más automatización.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-08-continuous-delivery-platform-engineering/108-proyecto-fabrica-de-software-multi-cloud/lab.py
```

El laboratorio selecciona el motor de práctica capstone y produce labresult.json. El escenario, sus comprobaciones y el artefacto esperado corresponden a esta clase; no requiere credenciales y deja explícito qué debe revalidarse en un sandbox real.

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa plataforma-entrega en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es un incremento integrado, demostrable y documentado. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye plataforma-entrega para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.
- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
Tres canalizaciones parecidas que divergen a los seis meses	Se replicó el camino por proveedor en vez de parametrizar lo que cambia	Aísla federación, registro, secretos y admisión en un paso final; el 77 % restante es común.
Se sabe qué artefacto corre y aun así no se puede reproducir el comportamiento	Falta el estado de los interruptores, que es la tercera pieza de la identidad del sistema	Registra siempre huella, confirmación del entorno y estado de los interruptores; con dos no basta.
Cada entorno efímero crea su propia base de datos y el coste se dispara	Se aplicó el aislamiento completo a un recurso caro sin evaluarlo	Instancia compartida con un esquema por entorno, y aislamiento completo solo donde haga falta.
Una promoción entre proveedores no es una promoción	El repositorio de entorno declara el registro junto con la huella	Deja la huella en la base común y el registro en la capa específica del entorno.
Se declara la fábrica terminada sin haberla desafiado	Se comprobó que funciona el camino feliz y ninguna prueba negativa	Ejecuta las siete pruebas negativas —versión rota, agente parado, poda excesiva, secreto, firma ajena, registro caído, interruptor con el sistema caído— y documenta las que fallan.
La hipótesis de la parte anterior se da por acertada sin revisarla	Calificar solo lo que salió bien convierte el aprendizaje en opinión	Publica el veredicto de cada predicción con su evidencia, incluida la partida mayor que la lista no contemplaba.

■ Seguridad, ética y costo

Trabaja con cuentas propias o sandboxes autorizados. No publiques secretos, identificadores reales ni datos personales. Antes de crear recursos pagos define presupuesto, etiquetas y comando de destrucción; después verifica que no queden recursos huérfanos. Los ejemplos locales enseñan contratos, pero no certifican cumplimiento ni disponibilidad de producción.

■ Preguntas de comprobación

1. ¿Qué tres datos identifican un sistema en ejecución, y por qué no bastan dos?
2. ¿Qué parte de la fábrica es común a los tres proveedores y cuál no?
3. ¿En qué acertó y en qué se equivocó la predicción de que la canalización sería el objetivo más valioso?
4. ¿Qué dice la ley 17 y cuál es la única defensa que ha funcionado contra ella?
5. ¿Qué predice la hipótesis de la parte 09 sobre los ocho mecanismos y sobre la ley dominante?

Referencias

- Forsgren, N., Humble, J. y Kim, G. (2018). Accelerate, cap. 4 — capacidades de entrega y su efecto conjunto. <https://itrevolution.com/product/accelerate/>
- CNCF (2025). Platforms white paper — la fábrica como producto interno y sus interfaces. <https://tag-app-delivery.cncf.io/whitepapers/platforms/>
- SLSA (2025). Levels and requirements — qué exige cada nivel de la cadena de construcción. <https://slsa.dev/spec/v1.0/levels>
- OpenGitOps (2025). Principles — reconciliación continua como base del recorrido. <https://opengitops.dev/>
- Kim, G. y otros (2016). The DevOps Handbook, parte IV — flujo, retroalimentación y aprendizaje continuo. <https://itrevolution.com/product/the-devops-handbook-second-edition/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 08 en PDF · Manual integral

Evaluación — 108 Proyecto: fábrica de software multi-cloud

Preguntas

1. Explica el problema que resuelve esta capacidad sin mencionar un proveedor.
2. Identifica una frontera de responsabilidad y su propietario.
3. Compara dos alternativas mediante confiabilidad, seguridad, costo y operación.
4. ¿Qué demuestra el laboratorio y qué no puede demostrar?
5. Propón un caso límite o una prueba de fallo.

Reto verificable

Entrega plataforma-entrega con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

- [ ] Resultado reproducible por una segunda persona.
- [ ] Evidencia vinculada a cada afirmación técnica.
- [ ] Prueba negativa o de fallo incluida.
- [ ] Costos y permisos expresados con unidades y alcance.
- [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

Nivel	Evidencia
A — competente	Cumple todo y defiende trade-offs con datos.
B — en desarrollo	Cumple lo esencial; falta profundidad en una dimensión.
C — inicial	Artefacto parcial o conclusión sin evidencia suficiente.
0	Sin entrega reproducible.