Multi-Cloud Engineering Program

Parte 07 — Infraestructura como código y configuración

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	07 de 23
Clases	085–096
Evaluaciones	12
Horas estimadas	52

Índice

Alcance de este cuaderno 2

Índice 3

Parte 07 — Infraestructura como código y configuración 4

085 — Declarativo, imperativo, idempotencia y convergencia . . . 6
086 — Terraform: HCL, providers, resources y grafo . . . 15
087 — Estado remoto, locking, cifrado y recuperación . . . 24
088 — Módulos, contratos, versiones y composición . . . 34
089 — Variables, outputs, locals y data sources . . . 44
090 — Plan, apply, drift, import y refactor con moved . . . 54
091 — Validación, lint, pruebas y policy as code . . . 63
092 — Secretos y datos sensibles en IaC . . . 73
093 — CloudFormation, Bicep, Pulumi y Terraform . . . 83
094 — Ansible e imagen dorada para configuración . . . 92
095 — Plantillas, golden paths y catálogo interno . . . 102
096 — Proyecto: infraestructura multiambiente promovible . . . 112

Parte 07 — Infraestructura como código y configuración

← Parte 06 · Índice completo · Parte 08 →

Descargar: Esta parte en PDF · Manual integral

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

Convertir infraestructura y políticas en cambios revisables, probables y repetibles.

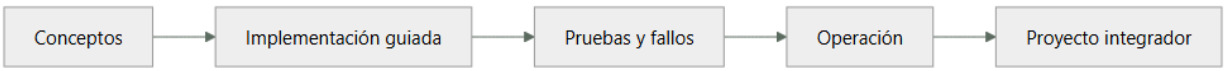
Resultados de la parte

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

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

Secuencia



Clases

ID	Clase	Laboratorio	Horas
085	Declarativo, imperativo, idempotencia y convergencia	iac	4
086	Terraform: HCL, providers, resources y grafo	iac	4
087	Estado remoto, locking, cifrado y recuperación	iac	4
088	Módulos, contratos, versiones y composición	iac	4
089	Variables, outputs, locals y data sources	iac	4
090	Plan, apply, drift, import y refactor con moved	iac	4
091	Validación, lint, pruebas y policy as code	testing	4
092	Secretos y datos sensibles en IaC	security	4
093	CloudFormation, Bicep, Pulumi y Terraform	decision	4
094	Ansible e imagen dorada para configuración	configuration	4
095	Plantillas, golden paths y catálogo interno	platform	4
096	Proyecto: infraestructura multiambiente promovible	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

- Infrastructure as Code — Kief Morris.
- Terraform: Up & Running — Yevgeniy Brikman.
- Ansible for DevOps — Jeff Geerling.

085 — Declarativo, imperativo, idempotencia y convergencia

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: iac · Estado: EXECUTABLECORE

Propósito

Fijar los cuatro conceptos que decidieron todo lo que va a ocurrir en esta parte, y que ya se han pagado varias veces en las anteriores: qué distingue declarar de ordenar, qué significa exactamente que una operación sea idempotente, por qué la convergencia es una propiedad del bucle y no del fichero, y por qué una plantilla que se aplica una vez no garantiza nada dentro de un mes. La clase 084 dejó una pregunta —si el bucle continuo es mejor que ejecutar órdenes, por qué la infraestructura se sigue gestionando con ejecuciones— y esta es la base para responderla.

Resultados de aprendizaje

Al finalizar podrás:

1. Distinguir una descripción del estado deseado de una secuencia de órdenes, con sus consecuencias operativas.
2. Comprobar si una operación es idempotente en vez de suponerlo.
3. Explicar por qué la convergencia exige un bucle y qué se degrada sin él.
4. Reconocer la desviación y clasificarla por su origen antes de corregirla.
5. Elegir entre reconciliar, recrear o corregir a mano según el tipo de recurso.

Conceptos centrales

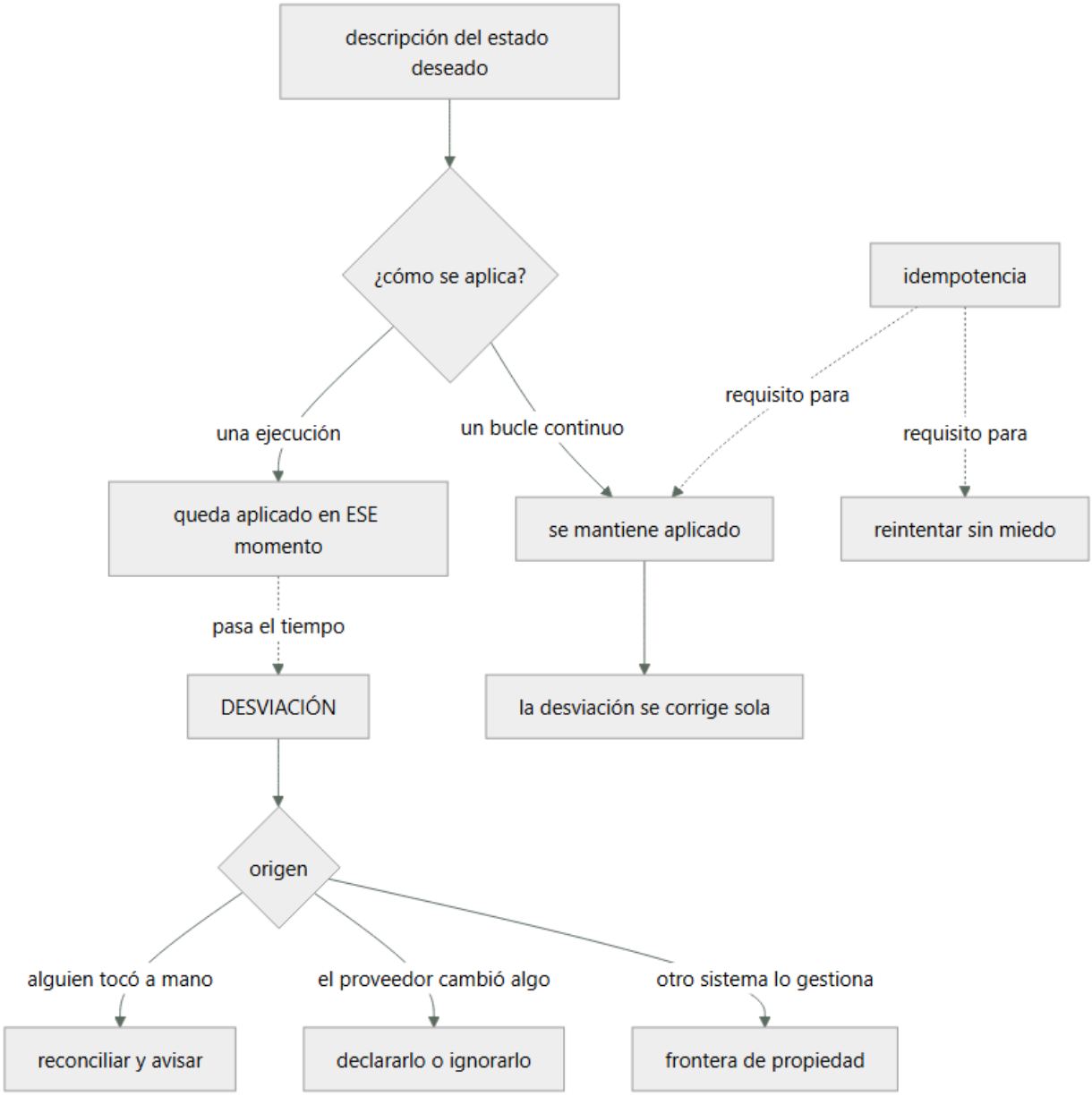
Concepto	Comprensión verificable
declarativo	Se describe qué debe existir; otro decide cómo llegar. La descripción vale igual desde cualquier estado de partida, y por eso se puede volver a aplicar.
imperativo	Se describe qué hacer, en orden. Depende del estado de partida, así que ejecutarlo dos veces puede no dar el mismo resultado.
idempotencia	Aplicar la misma operación dos veces deja el mismo resultado que aplicarla una. No es lo mismo que no fallar la segunda vez.
convergencia	Propiedad de un sistema que se acerca al estado deseado con el tiempo, corrigiendo lo que se desvía. Exige un bucle, no un fichero.
desviación	Diferencia entre lo declarado y lo que existe. Tiene tres orígenes distintos y cada uno pide una respuesta distinta.
ganado frente a mascota	Recurso que se reemplaza sin ceremonia frente a otro que se cuida y se repara. Decide si la respuesta a una desviación es recrear o corregir.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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. Declarar no es ordenar, y la diferencia se nota al repetir

La distinción parece obvia y sus consecuencias no lo son:

```
imperativo   "crea la red, luego la subred, luego la máquina"
             depende del estado de partida
             ejecutarlo dos veces puede fallar o duplicar
             para corregir algo hay que saber qué hay ahora

declarativo  "debe existir esta red con esta subred y esta máquina"
             vale desde cualquier estado de partida
             ejecutarlo dos veces deja lo mismo
             para corregir algo se compara y se actúa
```

La consecuencia práctica no es de elegancia: es que con una descripción declarativa se puede volver a aplicar sin saber qué pasó antes, y eso es lo que permite automatizar sin miedo.

Y conviene separar dos cosas que se confunden constantemente:

```
un FORMATO declarativo    un fichero que describe el estado deseado
un SISTEMA declarativo    algo que compara ese estado con la realidad
                          y actúa para acercarlos, una y otra vez
```

Un fichero de manifiestos aplicado a mano es un formato declarativo dentro de un proceso imperativo: alguien decide cuándo ejecutarlo. Y eso deja exactamente el hueco que la clase 084 señaló — nada garantiza que dentro de un mes la realidad se parezca al fichero.

Y hay un matiz que el programa ya ha pagado y conviene recordar: una descripción declarativa describe el estado, no el camino. Cuando el camino importa —una migración de datos, un orden de corte, un cambio de esquema en dos fases (clases 071 y 079)— la descripción no basta y hay que añadir un procedimiento. Ese es el límite honesto del enfoque, y las clases 047 y 059 ya lo enunciaron: la infraestructura como código describe el destino, no el trayecto.

Y una tercera categoría que aparece siempre y merece nombre, porque es donde vive la mayor parte del trabajo real:

```
lo declarativo cubre     lo que debe existir y cómo debe estar configurado
lo procedimental cubre   migraciones, cortes, rotaciones, restauraciones
el bucle cubre           que lo primero siga siendo cierto con el tiempo
```

Las tres piezas hacen falta, y confundir una con otra produce dos errores simétricos: intentar expresar una migración como estado deseado, o gestionar el estado deseado con guiones.

2. Idempotencia: qué es y qué no es

La definición es corta y se aplica mal con frecuencia:

```
una operación es idempotente si aplicarla N veces
deja el mismo resultado que aplicarla una vez
```

Y lo que no significa:

```
no significa "la segunda vez no falla"
no significa "no hace nada la segunda vez"
no significa "es segura de reintentar" — eso se DEDUCE de ella, no al revés
```

La distinción importa porque el programa ha dependido de esta propiedad seis veces:

```
manejadores de mensajes que pueden recibir duplicados   033, 044, 056
reproducción de mensajes para reparar un daño            056
plantillas que hay que reejecutar tras un fallo parcial   047, 059
reconciliación continua de un bucle                       073
```

Y en las cuatro, la propiedad se construye del mismo modo: la operación consulta el estado antes de actuar, o usa una clave que la identifica.

```
"crear el usuario X"                    no idempotente
"asegurar que existe el usuario X"      idempotente
"añadir 10 al saldo"                    no idempotente
"fijar el saldo a 150"                  idempotente
"aplicar la transacción con id T-991"   idempotente por clave
```

La tercera y la cuarta son la misma operación de negocio expresada de dos formas, y solo una se puede reintentar. Es exactamente la distinción que las clases 044 y 056 exigían al manejador.

Y la comprobación práctica, que se puede automatizar y casi nunca se hace:

```
$ aplicar && aplicar
# la segunda ejecución no debe cambiar nada
$ aplicar --solo-mostrar-cambios | wc -l
0                                                              ✓
```

Una segunda ejecución que sigue proponiendo cambios significa que algo no es idempotente, y las dos causas habituales conviene conocerlas:

```
1. la descripción no declara un campo que el sistema rellena solo
   → cada comparación ve una diferencia y propone corregirla
   → es el ruido del `what-if` de la clase 047 y del plan de la 059

2. algo genera un valor nuevo en cada ejecución
   una marca de tiempo, un identificador aleatorio, un secreto regenerado
   → cada aplicación cambia el recurso de verdad
```

La segunda es peor: no es ruido, es una plantilla que modifica infraestructura cada vez que se ejecuta, lo que convierte cada aplicación en un cambio con riesgo aunque nadie haya tocado el código.

3. La convergencia es del bucle, no del fichero

Un sistema converge si, dejado a su aire, se acerca al estado deseado. Y eso exige tres cosas, no una:

```
1. una descripción del estado deseado
2. la capacidad de OBSERVAR el estado real
3. algo que compare y actúe, REPETIDAMENTE
```

Las dos primeras las tienen todas las herramientas de esta parte. La tercera es la que cambia el resultado, y su ausencia produce una degradación con un patrón conocido:

```
día 0    se aplica: realidad = descripción
día 3    alguien cambia algo a mano para resolver una urgencia
día 12   el proveedor añade un campo por defecto
día 30   otro equipo modifica un recurso que creía suyo
día 60   nadie sabe si la descripción refleja la realidad
día 90   nadie se atreve a aplicarla, por si deshace algo necesario
```

Ese último renglón es el estado final de casi toda infraestructura como código sin bucle, y es peor que no tenerla: existe una descripción en la que nadie confía, así que los cambios se siguen haciendo a mano y la descripción se queda como documentación desactualizada.

Con un bucle, la línea temporal es otra:

```
día 3    alguien cambia algo a mano
día 3    el bucle lo detecta y lo revierte, y deja constancia
         → el cambio manual DEJA DE FUNCIONAR como método
```

Y ahí está el efecto cultural que importa más que el técnico: cuando los cambios manuales se deshacen solos, la única vía que funciona es el repositorio. No hay que convencer a nadie: el sistema hace que la otra vía no sirva.

Con dos matices honestos que hay que aceptar antes de adoptarlo:

```
revertir un cambio manual durante un incidente puede empeorarlo
  → hace falta una forma de suspender el bucle, y usarla deja rastro
el bucle necesita permisos amplios y funciona sin supervisión
  → es un actor privilegiado más, con su propia superficie
```

El primero se resuelve con un mecanismo explícito de suspensión y una alerta cuando se usa. El segundo con lo de siempre: privilegio mínimo, ámbito acotado y auditoría — el mismo tratamiento que cualquier identidad de carga desde la clase 026.

Y la ley 13 de la clase 084 aparece aquí antes de tiempo, y conviene anticiparla: si el bucle se detiene, no ocurre nada y nadie se entera. Un sistema convergente sin una señal de última reconciliación con éxito es un sistema que puede llevar semanas sin converger.

4. La desviación tiene tres orígenes y tres respuestas

Tratar toda diferencia entre lo declarado y lo real como «alguien tocó algo» lleva a corregir cosas que no había que corregir. Los tres orígenes:

```
1. INTERVENCIÓN MANUAL
   alguien cambió algo, casi siempre por una urgencia
   respuesta: reconciliar, y averiguar qué urgencia lo motivó
              — si el cambio era necesario, va al repositorio

2. EL PROVEEDOR
   campos que el servicio rellena, valores por defecto que cambian,
   normalizaciones
   respuesta: declararlos explícitamente, o ignorarlos de forma declarada

3. OTRO SISTEMA QUE GESTIONA EL MISMO RECURSO
   un escalado automático que cambia réplicas, un operador que ajusta
   una configuración, un controlador que añade anotaciones
   respuesta: FRONTERA DE PROPIEDAD, no reconciliación
```

El tercero es el que la clase 084 anticipó como el problema nuevo de esta parte, y ya apareció en la 081 con las réplicas oscilando entre el manifiesto y el escalado automático. Su corrección no es técnica sino de acuerdo:

```
para cada campo de cada recurso, UN solo dueño
lo que gestiona otro sistema, NO se declara
y si la herramienta lo permite, se declara que se ignora
```

Y hay un caso especial que merece decisión propia: los recursos que se crean solos. Un volumen que un controlador aprovisiona, un certificado que se emite, una regla que un servicio añade. Declararlos produce conflicto permanente; no declararlos deja infraestructura sin registro. La salida habitual es declarar quién los crea en vez de los recursos, que es

lo que hacen las clases de almacenamiento y los emisores de certificados.

Y la clasificación ganado o mascota decide la respuesta cuando algo está mal:

```
ganado    se reemplaza sin ceremonia: nodos, contenedores, instancias sin estado
          respuesta a una desviación: DESTRUIR y recrear
          más simple, más fiable, y exige que nada importante viva dentro

mascota   se cuida y se repara: una base de datos con años de datos,
          un recurso con una identidad que otros referencian
          respuesta a una desviación: corregir en su sitio, con cuidado
```

La mayor parte del valor de esta parte viene de mover cosas de la segunda categoría a la primera, y la pregunta que lo consigue es siempre la misma: qué hay dentro de este recurso que no se pueda reconstruir. Si la respuesta es «nada», es ganado, y su gestión se vuelve trivial.

Y una advertencia sobre recrear que el programa ya pagó en las clases 047 y 077: recrear no siempre es reversible. Un recurso con datos, con una dirección que otros usan o con una identidad referenciada no se puede sustituir sin consecuencias. Antes de convertir algo en ganado hay que comprobar que de verdad lo es.

5. Lo que esta parte tiene que responder

Con los cuatro conceptos fijados, las once clases siguientes tienen preguntas concretas que contestar, y conviene enunciarlas ahora para poder evaluarlas después:

```
086-090   cómo se describe, dónde vive lo que la herramienta cree saber,
          y qué hacer cuando la realidad y esa creencia divergen
091-092   cómo se comprueba antes de aplicar, y cómo se manejan los secretos
          sin repetir los tres incidentes de las clases 047, 059 y 061
093       qué cambia entre herramientas y qué es común
094-095   qué queda fuera de lo declarativo, y cómo se ofrece a otros equipos
096       si todo lo anterior sostiene cuatro entornos promovibles
```

Y tres criterios que van a decidir si una solución de esta parte es buena, sacados de lo que las partes anteriores ya midieron:

```
1. ¿se puede ver el cambio antes de aplicarlo, y es legible?
   el ruido convierte la revisión en un ritual: clases 047 y 059

2. ¿lo que se revisó es exactamente lo que se aplica?
   el plan guardado de la clase 059

3. ¿alguien sabría que el sistema dejó de converger?
   la ley 13 de la clase 084
```

Los tres son comprobaciones, no funciones de una herramienta. Y esa es la tesis de la parte, que conviene dejar dicha desde el principio:

> La diferencia entre una infraestructura como código que funciona y una que se abandona no está en la herramienta. Está en si el cambio se puede leer, si lo leído es lo que se ejecuta, y si alguien se entera cuando el mecanismo se detiene.

Y una nota sobre lo que no es infraestructura como código, porque la confusión produce proyectos interminables:

```
no es     reescribir toda la infraestructura existente antes de empezar
no es     describir hasta el último detalle de cada recurso
no es     prohibir toda intervención manual desde el primer día

sí es     que los recursos nuevos nazcan declarados
          que los existentes se adopten cuando haya un motivo para tocarlos
          que la intervención manual sea visible y temporal
```

El adoptar progresivamente es lo que hace viable el enfoque en una organización con años de infraestructura, y la clase 090 da el mecanismo concreto para incorporar lo que ya existe sin recrearlo.

Ejemplo trabajado

CloudShop tiene tres años de infraestructura y dos intentos fallidos de infraestructura como código. El equipo hace un inventario antes del tercer intento, y el resultado explica por qué fracasaron los dos anteriores.

El estado de partida.

```
repositorio de plantillas          existe, 41 ficheros
última aplicación real             hace 7 meses
recursos que describe              ~60 %
recursos que describe correctamente  se desconoce
nadie se atreve a aplicarlo        confirmado por tres personas distintas
```

La última fila es el diagnóstico completo: existía una descripción en la que nadie confiaba, así que todos los cambios seguían haciéndose a mano y la descripción envejecía.

Por qué fracasaron los dos intentos anteriores.

Al revisar el historial:

```
intento 1 (2024)   objetivo: describir TODA la infraestructura existente
                   abandonado a los 5 meses, al 34 %
                   causa: no había fecha de entrega ni valor intermedio

intento 2 (2025)   objetivo: aplicar la plantilla en producción
                   abandonado tras la primera aplicación
                   causa: el plan proponía 140 cambios y nadie sabía
                          cuáles eran reales
```

Los 140 cambios se analizaron por primera vez en este inventario, y su clasificación es el hallazgo del ejercicio:

```
campos que el proveedor rellena solo            96      proveedor
valores que otro sistema gestiona               22      otro sistema
cambios manuales reales                         14      intervención
diferencias legítimas por entorno                8      la plantilla estaba mal
```

El 84 % del ruido no era desviación. Eran campos no declarados y recursos gestionados por otro sistema, exactamente los orígenes 2 y 3 de esta clase. Y ese ruido fue lo que hizo la revisión ilegible y el intento inviable.

Las correcciones, por origen.

```
campos rellenados por el proveedor      no declarados     declarados: 96 campos
                                                          en 11 recursos
recursos gestionados por otro sistema    declarados      declarados como ignorados
                                                          o retirados de la plantilla
cambios manuales reales                      14          11 incorporados al
                                                        repositorio, 3 revertidos
diferencias por entorno                       8          expresadas como variables
```

Y el resultado sobre el plan:

```
cambios propuestos en una ejecución limpia    140                0
legibilidad de la revisión                  ilegible          una página
confianza para aplicar                      ninguna          aplicado el mismo día
```

Cero cambios propuestos en una ejecución sin modificaciones es la comprobación de idempotencia de esta clase, y fue la primera vez en tres años que se pudo hacer.

Los tres cambios manuales revertidos, y lo que enseñaron.

```
1. una regla de cortafuegos abierta "temporalmente" hacía 14 meses
   → se revirtió; nadie la echó de menos
2. un tamaño de instancia subido durante un incidente
   → era necesario: se incorporó al repositorio
3. una etiqueta añadida a mano para una auditoría
   → se incorporó, y se añadió a la plantilla de todos los recursos
```

La proporción —dos de tres cambios manuales eran necesarios— es la razón por la que revertir sin preguntar es un error. Un cambio manual es información sobre algo que la descripción no contemplaba.

Y la clasificación de recursos, que cambió el alcance del proyecto.

```
nodos y grupos de nodos                    sí
contenedores y despliegues                 sí
redes, subredes y reglas                   sí
balanceadores                              sí
bases de datos gestionadas                          sí
buckets con datos                                   sí
registros de DNS y certificados                     sí
identidades y roles                                 sí
```

Ocho categorías, cinco de ellas ganado. Y la decisión que salió de ahí:

```
lo que es ganado    se puede destruir y recrear en cualquier entorno
                    → la plantilla se prueba de verdad, creando y destruyendo
lo que es mascota   se declara, se protege contra el borrado (clases 042, 077)
                    y NUNCA se recrea desde la plantilla
```

Con esa separación, el entorno de pruebas pasó a crearse y destruirse en cada cambio importante, lo que convirtió la plantilla en algo probado en vez de algo aplicado y esperado.

Resumen del inventario previo:

```
cambios propuestos en ejecución limpia         140              0
recursos descritos                             ~60 %          88 %
recursos descritos y verificados            se desconoce      88 %
meses desde la última aplicación               7              0
recursos clasificados como ganado o mascota    no        8 categorías
entorno de pruebas creado desde cero          nunca      en cada cambio mayor
```

La lección que esta clase traslada al resto de la parte 07: los dos intentos anteriores no fracasaron por la herramienta ni por falta de disciplina. Fracasaron porque el 84 % del ruido de la revisión no era desviación, y con la revisión ilegible no hay forma de aplicar nada con confianza. La corrección —declarar lo que el proveedor rellena y retirar lo que gestiona otro sistema— es aburrida, cuesta unos días y es la que hace posible todo lo demás.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/085-declarativo-imperativo-idempotencia-y-convergencia/lab.py
```

El laboratorio selecciona el motor de práctica iac 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 comparativa-iac 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 plan reproducible sin secretos ni cambios inesperados. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye comparativa-iac 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
Existe una plantilla que describe la infraestructura y nadie se atreve a aplicarla	La revisión propone decenas de cambios y no se distingue el ruido de la desviación real	Clasifica cada diferencia por origen: campos del proveedor, otro sistema o intervención manual; los dos primeros se declaran o se ignoran.
Una segunda ejecución seguida sigue proponiendo cambios	La descripción no es idempotente: faltan campos por declarar o algo genera un valor nuevo cada vez	Ejecuta dos veces y exige cero cambios en la segunda; declara los campos calculados y elimina los valores que se regeneran.
La descripción envejece hasta ser documentación desactualizada	Se aplica en ejecuciones puntuales; no hay bucle que la mantenga cierta	Un mecanismo que reconcilie continuamente, con señal de última reconciliación con éxito y alerta de envejecimiento.
Un campo cambia constantemente entre la plantilla y otro sistema	Dos sistemas se creen dueños del mismo campo	Un solo dueño por campo: lo que gestiona otro sistema no se declara, o se declara explícitamente como ignorado.
Revertir un cambio manual causa un incidente	El cambio era necesario y respondía a algo que la descripción no contemplaba	Trata cada intervención manual como información: averigua qué la motivó antes de revertirla, e incorpora lo que sea legítimo.
Recrear un recurso desde la plantilla destruye datos o rompe referencias	Se trató como ganado algo que era mascota	Clasifica cada categoría de recurso antes de automatizar su recreación, y protege contra el borrado lo que no se puede reconstruir.

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 diferencia hay entre un formato declarativo y un sistema declarativo, y qué hueco deja el primero?
2. Da dos ejemplos de operación no idempotente y su equivalente idempotente, y di cómo se comprueba.
3. ¿Qué tres cosas exige la convergencia, y qué pasa con el tiempo cuando falta la tercera?
4. Clasifica tres desviaciones por origen y di qué respuesta corresponde a cada una.
5. ¿Qué pregunta decide si un recurso es ganado o mascota, y qué implica la respuesta?

Referencias

- Kief Morris (2020). Infrastructure as Code, 2.ª ed., cap. 2 — principios, idempotencia y convergencia. <https://infrastructure-as-code.com/book/>
- Mark Burgess (2004). Promise Theory and convergent operators — origen del concepto de convergencia en configuración. <http://markburgess.org/promises.html>
- HashiCorp (2025). Drift detection and reconciliation — orígenes de la desviación y respuestas. <https://developer.hashicorp.com/terraform/tutorials/state/resource-drift>
- Kubernetes (2025). Controllers and the reconciliation loop — el bucle como mecanismo de convergencia. <https://kubernetes.io/docs/concepts/architecture/controller/>
- Bill Baker (2012). Scaling SQL Server: pets vs cattle — origen de la distinción y su uso operativo. <https://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Evaluación — 085 Declarativo, imperativo, idempotencia y convergencia

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 comparativa-iac 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.

086 — Terraform: HCL, providers, resources y grafo

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: iac · Estado: EXECUTABLECORE

Propósito

Escribir Terraform entendiendo la pieza que decide su comportamiento y que casi nadie mira: el grafo. El orden de ejecución no lo da el fichero ni el orden de las declaraciones, lo dan las referencias entre recursos — y de ahí salen los tres problemas característicos: dependencias que existen y no se pueden expresar con una referencia, ciclos que aparecen al refactorizar, y recursos que no se pueden sustituir en el sitio. La clase 085 dejó además una deuda concreta —qué hacer con los campos que gestiona otro sistema— y aquí está el mecanismo.

Resultados de aprendizaje

Al finalizar podrás:

1. Leer un conjunto de ficheros como un grafo y predecir en qué orden se aplicará.
2. Distinguir una dependencia expresable por referencia de una que exige declararla.
3. Elegir el comportamiento de sustitución adecuado para un recurso que no se puede reemplazar en el sitio.
4. Ignorar los campos que gestiona otro sistema, cerrando la deuda de la clase 085.
5. Configurar proveedores con versiones fijadas y con alias para varias regiones o cuentas.

Conceptos centrales

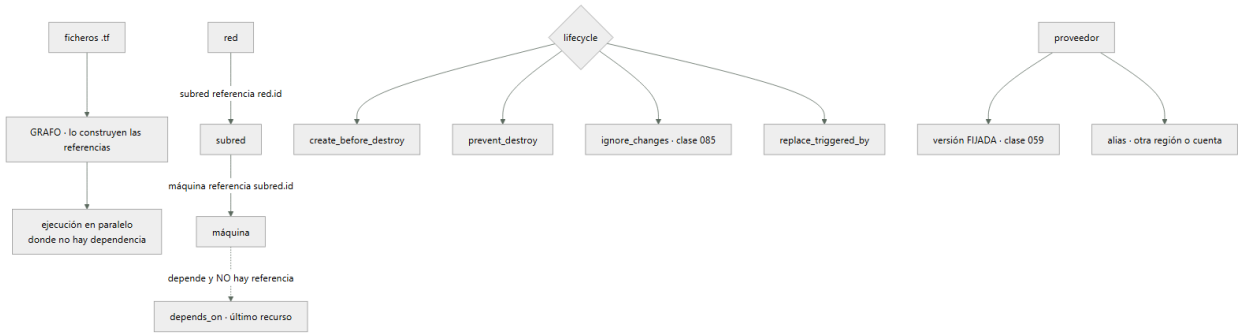
Concepto	Comprensión verificable
grafo de dependencias	Estructura que Terraform construye a partir de las referencias entre recursos. Decide el orden y el paralelismo; el orden de los ficheros es irrelevante.
dependencia implícita	La que nace de referenciar un atributo de otro recurso. Es la forma correcta: se declara sola y no hay que mantenerla.
dependencia explícita	La que se declara a mano porque no hay ninguna referencia que la exprese. Es un último recurso y casi siempre señala un efecto colateral.
crear antes de destruir	Comportamiento que crea el sustituto antes de retirar el original. Imprescindible cuando el recurso no se puede reemplazar en el sitio y algo depende de él.
ignorar cambios	Declaración de que un campo lo gestiona otro sistema. Es la respuesta al origen 3 de la desviación de la clase 085.
alias de proveedor	Segunda configuración del mismo proveedor, para otra región o cuenta. Se pasa explícitamente a los recursos que la usan.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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 orden lo da el grafo, no el fichero

Terraform lee todos los ficheros de un directorio como si fueran uno, ignora el orden en que aparecen las declaraciones y construye un grafo a partir de las referencias:

```
resource "aws_subnet" "app" {
  vpc_id     = aws_vpc.principal.id      # ← esto crea la dependencia
  cidr_block = "10.20.4.0/24"
}

resource "aws_vpc" "principal" {        # declarado DESPUÉS, se crea ANTES
  cidr_block = "10.20.0.0/16"
}
```

De ahí salen tres consecuencias que conviene interiorizar:

El paralelismo es automático. Lo que no depende de nada se crea a la vez. Por defecto se ejecutan hasta diez operaciones simultáneas, y eso puede chocar con los límites de tasa del proveedor:

```
Error: Throttling: Rate exceeded
```

La corrección no es reintentar más: es bajar el paralelismo en las ejecuciones que crean muchos recursos del mismo tipo.

```
$ terraform apply -parallelism=4 tfplan
```

Organizar en ficheros es solo para las personas. Separar en red.tf, datos.tf y computo.tf no cambia nada del comportamiento; es legibilidad. Y por eso una convención sencilla vale más que una elaborada:

```
main.tf        los recursos
variables.tf   las entradas
outputs.tf     las salidas
versions.tf    proveedores y sus versiones
```

Ver el grafo ayuda cuando algo no cuadra:

```
$ terraform graph -type=plan | dot -Tsvg > grafo.svg
```

Y hay un caso en que hace falta mirarlo: los ciclos. Aparecen al refactorizar, cuando dos recursos acaban referenciándose mutuamente:

```
Error: Cycle: aws_security_group.app, aws_security_group.bd
```

Ocurre con reglas de cortafuegos que se referencian entre sí, y la solución es sacar las reglas a recursos independientes:

```
# en vez de reglas dentro de cada grupo, que se referencian mutuamente
resource "aws_vpc_security_group_ingress_rule" "bd_desde_app" {
  security_group_id            = aws_security_group.bd.id
  referenced_security_group_id = aws_security_group.app.id
  from_port                    = 5432
  to_port                      = 5432
  ip_protocol                  = "tcp"
}
```

Es el mismo patrón que la clase 047 exigía para los recursos hijo: una sola forma de declararlos, y separada del padre cuando la relación es cruzada.

2. Cuando la dependencia existe y no se puede referenciar

La dependencia implícita es la buena: nace sola de una referencia y no hay que mantenerla. La explícita es un último recurso, y conviene saber cuándo hace falta de verdad.

El caso canónico es el efecto colateral: el recurso A necesita que B exista, pero no usa ningún atributo suyo.

```
resource "aws_iam_role_policy_attachment" "lectura" {
  role       = aws_iam_role.app.name
  policy_arn = aws_iam_policy.lectura.arn
}

resource "aws_lambda_function" "procesar" {
  role = aws_iam_role.app.arn        # depende del ROL, no del permiso
  # …
  depends_on = [aws_iam_role_policy_attachment.lectura]
}
```

Sin esa declaración, la función se crea antes de que el permiso esté asociado. Y el síntoma es característico y desespera: funciona unas veces y otras no, porque depende de qué operación termine antes.

Y hay una segunda causa que produce el mismo síntoma y no se arregla con dependencias: la propagación. Un permiso creado hace un segundo puede no estar visible todavía para el servicio que lo usa. Ahí no hay orden que valga, y las salidas son otras:

```
reintento en el propio proveedor      muchos lo hacen; conviene comprobar si lo hace
una espera declarada                  fea y a veces la única
dos ejecuciones                       la segunda funciona: es la señal de
                                      que el problema es de propagación
```

La última línea es un diagnóstico útil: si volver a aplicar sin cambiar nada arregla el error, el problema es de propagación y no de grafo.

Y el mal uso más común de las dependencias explícitas es ponerlas «por si acaso»:

```
depends_on = [aws_vpc.principal, aws_subnet.app, aws_security_group.bd]
```

Eso añade aristas que el grafo ya tenía, reduce el paralelismo y oculta las dependencias reales. La regla es escribirla solo cuando se puede explicar por qué no hay referencia posible, y dejar un comentario que lo diga.

Y un caso especial que conviene conocer porque produce ejecuciones lentísimas: usar una dependencia explícita sobre un módulo entero. Eso hace que todo lo del módulo espere a todo lo del otro, y en módulos grandes convierte una aplicación paralela en una secuencial:

```
module "aplicacion" {
  source     = "./modulos/aplicacion"
  depends_on = [module.red]        # ← todo espera a todo
}
```

La alternativa es pasar los valores concretos que la aplicación necesita de la red, con lo que la dependencia se vuelve implícita y solo afecta a los recursos que de verdad la tienen.

3. Los cuatro comportamientos de ciclo de vida

El bloque de ciclo de vida cambia cómo se sustituye o se protege un recurso, y sus cuatro opciones responden a problemas distintos.

Crear antes de destruir. Por defecto, un recurso que hay que sustituir se destruye y luego se crea. Para lo que sirve tráfico o lo que otros referencian, eso es un corte:

```
resource "aws_launch_template" "app" {
  # …
  lifecycle {
    create_before_destroy = true
  }
}
```

Y tiene una condición que produce errores al activarlo: durante la sustitución existen los dos a la vez, así que cualquier restricción de unicidad falla. Nombres, direcciones fijas, puertos reservados:

```
Error: name already exists
```

La corrección es que el nombre lo genere el proveedor o que lleve un sufijo derivado del contenido:

```
name_prefix = "app-"          # en vez de name = "app"
```

Impedir la destrucción. Hace fallar la planificación si algo propondría borrar ese recurso:

```
lifecycle {
  prevent_destroy = true
}
```

Es la protección de la clase 059 y tiene una particularidad: el error aparece en el plan, antes de ejecutar nada, lo que la hace mejor que una protección del proveedor para detectar el problema pronto. Las dos son complementarias, y la clase 059 ya pedía ambas.

Ignorar cambios. Es la respuesta a la deuda que dejó la clase 085 — el origen 3 de la desviación, otro sistema gestionando el mismo campo:

```
resource "aws_autoscaling_group" "app" {
  desired_capacity = 3
  # …
  lifecycle {
    ignore_changes = [desired_capacity]     # lo gestiona el escalado automático
  }
}
```

Con eso, el valor inicial lo pone la plantilla y las modificaciones posteriores del otro sistema no producen ruido. Es exactamente la frontera de propiedad que la clase 085 pedía declarar, y conviene acompañarla de un comentario que diga quién es el dueño, porque dentro de un año nadie se acordará.

Y su versión total, para recursos cuyo contenido gestiona otro por completo:

```
ignore_changes = all
```

Que hay que usar con cuidado: a partir de ahí, la plantilla solo controla la existencia del recurso, no su configuración.

Sustituir cuando cambie otra cosa. Fuerza la recreación al cambiar un recurso del que no se depende por atributo:

```
resource "aws_instance" "app" {
  # …
  lifecycle {
    replace_triggered_by = [terraform_data.version_config]
  }
}
```

Es el equivalente de la anotación con huella de la clase 076: un cambio en algo externo debe producir un reemplazo, y sin esto no lo produce.

Y una advertencia sobre los aprovisionadores —ejecutar órdenes tras crear un recurso—: son código imperativo dentro de un sistema declarativo, no son idempotentes, no se reintentan bien y si fallan dejan el recurso marcado como contaminado. La documentación oficial los describe como último recurso y conviene tomárselo literalmente:

```
en vez de un aprovisionador que instala software
  → una imagen construida antes (clases 062, 094)
en vez de un aprovisionador que registra algo
  → un recurso del proveedor que lo haga, o datos de arranque
en vez de un aprovisionador que espera
  → comprobar por qué hace falta esperar
```

4. Proveedores: versiones, alias y lo que aportan

Un proveedor es un complemento que traduce las declaraciones a llamadas de una API. Y su configuración tiene dos partes que conviene no mezclar:

```
terraform {
  required_version = "~> 1.9"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.70"          # QUÉ versión
    }
  }
}

provider "aws" {
  region = var.region                # CÓMO se configura
}
```

La primera va siempre en el módulo raíz y se fija, por lo mismo que la clase 059 exigía: dos ejecuciones del mismo código en semanas distintas deben usar el mismo proveedor. Y el fichero de bloqueo se versiona en el repositorio:

```
$ terraform providers lock -platform=linux_amd64 -platform=darwin_arm64
```

Esa orden merece conocerse porque evita un fallo típico: el fichero de bloqueo generado en un portátil no incluye las huellas para la plataforma del agente de la canalización, y la ejecución falla ahí.

Los alias permiten varias configuraciones del mismo proveedor:

```
provider "aws" {
  region = "eu-west-1"
}

provider "aws" {
  alias  = "us"
  region = "us-east-1"
}

resource "aws_s3_bucket" "replica" {
  provider = aws.us
  bucket   = "cls-replica"
}
```

Y su uso más frecuente no es multirregión sino multicuenta: un proveedor por cuenta, cada uno asumiendo un rol distinto, que es el patrón de la clase 025 aplicado aquí. Con la advertencia de la clase 059: la identidad se obtiene por federación, sin claves.

Y los módulos reciben proveedores explícitamente cuando usan alias:

```
module "replica" {
  source    = "./modulos/bucket"
  providers = { aws = aws.us }
}
```

Una nota sobre lo que conviene no hacer: configurar proveedores dentro de un módulo reutilizable. Impide usar alias desde fuera y complica su composición; la clase 088 lo desarrolla.

Y las fuentes de datos, que son la otra mitad del modelo:

```
data "aws_vpc" "principal" {
  tags = { Name = "vpc-cloudshop" }
}
```

Leen algo que existe y que gestiona otro. Su ventaja sobre pasar identificadores como variables es que no se rompen cuando algo cambia de nombre, y su riesgo es que una consulta que no encuentra nada —o que encuentra varias cosas— falla la planificación entera:

```
Error: multiple VPCs matched; use additional constraints
```

Por eso conviene que los filtros sean exactos, y para lo que de verdad es un contrato entre equipos, la clase 088 propone una alternativa mejor que buscar por etiquetas.

5. Leer una plantilla como se lee código

Con el grafo y el ciclo de vida claros, hay una lista corta que detecta la mayoría de los problemas al revisar un cambio:

- ¿las dependencias son implícitas? toda declaración explícita, justificada
- ¿hay recursos con nombre fijo que también tienen crear antes de destruir?
- ¿los campos que gestiona otro sistema están declarados como ignorados, con un comentario que diga quién es el dueño?
- ¿los recursos que no se pueden perder tienen protección contra destrucción?
- ¿las versiones de proveedor están fijadas y el fichero de bloqueo versionado?
- ¿hay aprovisionadores? cada uno, justificado
- ¿las fuentes de datos filtran de forma que solo puedan devolver una cosa?

Y dos comprobaciones automáticas que valen más que la lista:

```
# 1. el plan de una ejecución sin cambios debe estar vacío (clase 085)
$ terraform plan -detailed-exitcode
# 0 = sin cambios · 2 = hay cambios · 1 = error

# 2. ningún recurso se destruye sin que alguien lo haya leído
$ terraform show -json tfplan | jq -r '.resource_changes[]
  | select(.change.actions | index("delete")) | .address'
```

La primera es la comprobación de idempotencia de la clase 085 en su forma ejecutable, y debería estar en la canalización: si aplicar dos veces seguidas produce cambios, algo está mal declarado.

Y una lectura del plan que ahorra sustos: los cuatro símbolos y lo que implican.

```
+  crear                 sin riesgo salvo por el coste
~  modificar en el sitio  sin corte, normalmente
-/+ destruir y crear      HAY CORTE, salvo con crear antes de destruir
+/- crear y destruir      con crear antes de destruir activado
-  destruir              lo que hay que leer siempre
```

La tercera línea es la que hay que buscar primero en cualquier plan, porque es donde están las sorpresas: un cambio de un campo que obliga a recrear. Y el plan dice cuál:

```
~ resource "aws_db_instance" "pedidos" {
    ~ availability_zone = "eu-west-1a" -> "eu-west-1b" # forces replacement
```

Ese comentario al final de la línea es la información más importante de un plan, y es la que la revisión tiene que buscar. Un recurso con datos marcado para recreación es un incidente que todavía no ha ocurrido, y la protección contra destrucción de la clase 059 lo convierte en un error de planificación en vez de en una pérdida.

Ejemplo trabajado

CloudShop reescribe sus plantillas después del inventario de la clase 085. Los cuatro problemas del primer mes son todos del grafo o del ciclo de vida, y ninguno se resuelve leyendo la documentación del recurso.

Problema 1 — el despliegue funcionaba tres de cada cinco veces.

```
Error: AccessDenied: User is not authorized to perform: s3:GetObject
```

El error aparecía al crear la función y desaparecía al volver a aplicar sin cambiar nada, lo que es el diagnóstico de propagación de esta clase. Pero había además un problema de grafo:

```
$ terraform graph -type=plan | grep -c 'aws_iam_role_policy_attachment.*aws_lambda_function'
0        ← no hay ninguna arista entre el permiso y la función
```

La función referenciaba el rol, no el permiso, así que el grafo permitía crearlos en paralelo.

```
dependencia permiso → función             inexistente     declarada explícitamente
ejecuciones que fallaban                    2 de 5              0
comentario que la justifica                 no había      "no hay atributo del
                                                          permiso que la función use"
```

Problema 2 — un ciclo al separar las reglas de cortafuegos.

```
Error: Cycle: aws_security_group.app, aws_security_group.bd
```

Cada grupo declaraba dentro sus reglas, y una de ellas referenciaba al otro grupo. Es el mismo patrón que la clase 047 encontró con las reglas en línea frente a los recursos hijo, ahora con otra herramienta.

```
reglas                          dentro de cada grupo   recursos independientes
ciclos en el grafo                      1                    0
convención documentada                 no                  sí, una sola forma
```

Problema 3 — la sustitución de una plantilla de arranque cortaba el servicio.

```
-/+ resource "aws_launch_template" "app"  # forces replacement
```

Al activar crear antes de destruir, el error cambió:

```
Error: InvalidLaunchTemplateName.AlreadyExistsException
```

Los dos existían a la vez durante la sustitución, y el nombre era fijo.

```
nombre                                  fijo ("app")     prefijo generado
crear antes de destruir                   no                 sí
corte durante la sustitución              40 s                0
```

Problema 4 — la capacidad deseada volvía a tres cada noche.

El escalado automático subía las instancias durante el día y la aplicación nocturna de la plantilla las devolvía a tres.

```
$ terraform plan | grep desired_capacity
  ~ desired_capacity = 9 -> 3
```

Es el origen 3 de la desviación de la clase 085 —otro sistema gestiona ese campo— y es el mismo incidente que la clase 081 encontró con los manifiestos y el escalado del clúster. Tercera aparición del mismo conflicto de propiedad.

```
campo desired_capacity                  declarado         ignorado, con comentario
                                                          que nombra al dueño
capacidad al día siguiente de aplicar        3            la que decidió el escalado
cambios propuestos en ejecución limpia       1                  0
```

Y la comprobación que se añadió a la canalización.

```
$ terraform apply tfplan
$ terraform plan -detailed-exitcode
$ [ $? -eq 0 ] || { echo "la plantilla no es idempotente"; exit 1; }
```

Esa comprobación, ejecutada por primera vez sobre las cuatro carpetas del repositorio, encontró tres plantillas más con campos no declarados:

```
carpeta      cambios en la segunda ejecución   causa
red                    0                       —
datos                  2                       etiquetas que el proveedor añade
computo                1                       campo calculado no declarado
plataforma             4                       recurso gestionado por un operador
```

Las siete diferencias eran ruido y no desviación, exactamente en la proporción que la clase 085 había medido.

Resumen:

```
ejecuciones que fallaban de forma intermitente   2 de 5          0
ciclos en el grafo                                 1             0
corte al sustituir una plantilla de arranque     40 s            0
cambios propuestos en ejecución limpia            7              0
dependencias explícitas                           9        2, ambas justificadas
recursos con protección contra destrucción        0             6
```

La lección que esta clase traslada al resto de la parte 07: los cuatro problemas se ven en el grafo o en el plan, y ninguno en la documentación del recurso. La comprobación que los resume es la de la clase 085 en forma ejecutable —aplicar dos veces seguidas y exigir cero cambios— y encontró siete diferencias en cuatro carpetas la primera vez que se ejecutó. Es una línea en la canalización y es el único indicador que dice si una plantilla describe la realidad o solo se le parece.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/086-terraform-hcl-providers-resources-y-grafo/lab.py
```

El laboratorio selecciona el motor de práctica iac 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 stack-terraform 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 plan reproducible sin secretos ni cambios inesperados. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye stack-terraform 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 aplicación funciona unas veces y otras no, con errores de permisos	El grafo permite crear en paralelo dos recursos con dependencia real que ninguna referencia expresa	Declara la dependencia explícitamente con un comentario que explique por qué no hay referencia posible.
Volver a aplicar sin cambiar nada arregla el error	Es propagación del proveedor, no un problema de orden	Comprueba si el proveedor reintenta; si no, una espera declarada acotada, y documenta el motivo.
Aparece un ciclo al refactorizar reglas de red	Dos recursos se referencian mutuamente porque las reglas están dentro de cada uno	Saca las reglas a recursos independientes; una sola forma de declararlas, como en la clase 047.
Activar crear antes de destruir produce un error de nombre duplicado	Durante la sustitución existen los dos y el nombre es fijo	Usa prefijo generado o deja que el proveedor asigne el nombre.
Un campo vuelve a su valor de la plantilla en cada aplicación	Otro sistema gestiona ese campo: es el origen 3 de la desviación	Decláralo como ignorado y añade un comentario que nombre al dueño.
Un plan propone recrear un recurso con datos	Un campo que obliga a reemplazo ha cambiado	Busca el comentario que lo señala en el plan y protege esos recursos contra la destrucción para que el error salte al planificar.

■ 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 determina el orden de creación, y por qué el orden de los ficheros es irrelevante?
2. ¿Cuándo hace falta una dependencia explícita, y cómo se distingue de un problema de propagación?
3. ¿Qué condición hay que cumplir para activar crear antes de destruir sin que falle?
4. ¿Cómo se cierra la deuda que la clase 085 dejó sobre los campos que gestiona otro sistema?
5. ¿Qué comprobación de una línea dice si una plantilla describe la realidad o solo se le parece?

Referencias

- HashiCorp (2025). Resource behavior and the dependency graph — dependencias implícitas y explícitas. <https://developer.hashicorp.com/terraform/language/resources/behavior>
- HashiCorp (2025). The lifecycle meta-argument — crear antes de destruir, ignorar cambios y protecciones. <https://developer.hashicorp.com/terraform/language/meta-arguments/lifecycle>
- HashiCorp (2025). Provider configuration and aliases — varias configuraciones del mismo proveedor. <https://developer.hashicorp.com/terraform/language/providers/configuration>
- HashiCorp (2025). Provider requirements and dependency lock file — fijar versiones y plataformas. <https://developer.hashicorp.com/terraform/language/files/dependency-lock>

- HashiCorp (2025). Provisioners are a last resort — por qué evitarlos y qué usar en su lugar. <https://developer.hashicorp.com/terraform/language/resources/provisioners/syntax>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 07 en PDF · Manual integral

Anterior	Índice	Siguiente
← 085 · Declarativo, imperativo, idempotencia y convergencia	Parte 07 · Programa	087 · Estado remoto, locking, cifrado y recuperación →

Evaluación — 086 Terraform: HCL, providers, resources y grafo

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 stack-terraform 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.

087 — Estado remoto, locking, cifrado y recuperación

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: iac · Estado: EXECUTABLECORE

Propósito

Operar el archivo de estado, que la clase 059 presentó como la diferencia estructural de Terraform y que aquí se trata como lo que es: el activo más sensible y más frágil del sistema. Contiene los secretos en claro, su pérdida convierte infraestructura viva en recursos huérfanos, y dos ejecuciones simultáneas lo corrompen. La clase cubre cómo se guarda, cómo se bloquea, cómo se recupera cuando algo va mal y —lo que casi nunca se ensaya— cómo se sale de un estado que ya no corresponde con la realidad.

Resultados de aprendizaje

Al finalizar podrás:

1. Configurar un almacén remoto con bloqueo, versionado y cifrado, y justificar cada pieza.
2. Diagnosticar un bloqueo atascado y liberarlo sin corromper nada.
3. Recuperar un estado perdido o corrupto a partir de sus versiones.
4. Reparar una discrepancia entre el estado y la realidad sin destruir recursos.
5. Repartir el estado por radio de impacto y conectar las piezas sin acoplarlas.

Conceptos centrales

Concepto	Comprensión verificable
archivo de estado	Correspondencia entre lo declarado y los recursos reales, con todos sus atributos. Guarda valores sensibles en claro y su pérdida no destruye nada, pero deja todo huérfano.
bloqueo	Exclusión mutua durante una operación. Sin él, dos ejecuciones simultáneas escriben encima y el estado deja de corresponder con la realidad.
bloqueo atascado	Bloqueo que quedó tomado porque el proceso murió. Se libera con una orden, y liberarlo mientras otra ejecución sigue viva es la forma de corromper el estado.
recurso huérfano	Recurso real que ya no aparece en ningún estado. Nadie lo gestiona, sigue costando y una plantilla que intente recrearlo chocará con él.
estado por unidad de despliegue	Un estado por conjunto de recursos con el mismo radio de impacto. Reduce el bloqueo entre equipos y el tiempo de planificación.
acoplamiento por estado	Leer las salidas del estado de otro equipo. Funciona y crea una dependencia dura; una fuente de datos consulta la realidad y no se rompe al recrear.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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

```
flowchart TB
    T["terraform"] --> S["estado remoto"]
    S --> P1["bloqueo · una ejecución cada vez"]
    S --> P2["versionado · recuperación"]
    S --> P3["cifrado · contiene secretos"]
    S --> P4["acceso mínimo · leerlo es leer los secretos"]
    S -.->|"se pierde"| H["recursos HUÉRFANOS\nsiguen vivos y sin dueño"]
    S -.->|"discrepa de la realidad"| R{"reparar"}
    R --> R1["importar lo que existe y no está"]
    R --> R2["retirar del estado lo que ya no existe"]
    R --> R3["reemplazar lo que existe y está mal"]
    D["repartir el estado"] --> U1["red"]
    D --> U2["datos"]
    D --> U3["aplicación"]
    U1 -.->|"mejor que salidas"| F["fuente de datos: consulta la realidad"]
```

Desarrollo

1. Dónde vive el estado y qué hay que ponerle

El estado local sirve para probar y no sirve para nada más: no se comparte, no se bloquea, no se recupera y acaba en el repositorio con los secretos dentro. El remoto es la única opción operable, y necesita cuatro cosas:

```
terraform {
  backend "s3" {
    bucket       = "cls-tfstate-prod"
    key          = "red/terraform.tfstate"
    region       = "eu-west-1"
    encrypt      = true
    use_lockfile = true
  }
}
```

```
1. BLOQUEO      dos ejecuciones a la vez corrompen el estado
2. VERSIONADO   un estado corrupto o borrado se recupera de una versión anterior
3. CIFRADO      contiene secretos en claro (clase 059)
4. ACCESO MÍNIMO  quien lee el estado lee los secretos
```

Y el punto 4 merece insistencia porque es el que se configura peor. El bucket del estado hereda con frecuencia los permisos del proyecto donde vive, y ahí es donde la clase 059 encontró catorce personas con acceso a siete contraseñas de producción.

```
el estado va en un proyecto o cuenta propios
lectura solo para las identidades que planifican
escritura solo para las que aplican (clase 059)
sin acceso público, con versionado, y con clave del cliente si el requisito existe
```

Y una consecuencia que sorprende y hay que aceptar: no existe una forma de que el estado no contenga los valores sensibles. Marcar una variable como sensible oculta el valor en la salida por pantalla y no lo cifra en el estado. La única protección real es el control de acceso al almacén.

```
$ terraform state pull | jq -r '.resources[].instances[].attributes
  | to_entries[] | select(.key | test("password|secret|private_key"))
  | .key' | sort -u
```

Esa orden lista los campos sensibles que hay ahí dentro, y conviene ejecutarla una vez para dimensionar el problema antes de decidir permisos.

Y sobre el bloqueo, un detalle operativo que aparece pronto:

```
Error: Error acquiring the state lock
  Lock Info:
    ID:        7f4c9b2d-…
    Operation: OperationTypeApply
    Who:       agente@ci-runner-14
    Created:   2026-08-03 09:41:22
```

La información dice quién lo tiene y desde cuándo. Y hay dos situaciones distintas:

```
otra ejecución está en marcha        esperar; es lo que debe pasar
el proceso murió y el bloqueo quedó   liberarlo, con cuidado
```

```
$ terraform force-unlock 7f4c9b2d-…
```

Liberar un bloqueo cuya ejecución sigue viva es la forma más directa de corromper el estado, porque las dos escribirán. La comprobación previa es mirar la marca de tiempo y confirmar que el agente que lo tomó ya no existe. En una canalización, un tiempo de espera de bloqueo evita la mayoría de los casos:

```
$ terraform apply -lock-timeout=10m tfplan
```

2. Repartir el estado, y conectar sin acoplar

Un solo estado para toda la plataforma tiene cuatro problemas concretos:

```
cualquier cambio bloquea a todos los equipos
cada planificación consulta cientos de recursos: lenta
el radio de impacto de un `destroy` es todo
y un estado corrupto se lleva la gestión de todo a la vez
```

La unidad correcta es la misma que en las clases 047 y 059: el radio de impacto.

```
cls-tfstate-prod/
  red/           subredes, cortafuegos, salida
  datos/         bases de datos, buckets, claves
  plataforma/    clúster y sus complementos
  tienda/        la aplicación
```

Y la pregunta que decide dónde va cada recurso es: ¿qué querría poder destruir junto? Lo que se destruye junto va junto.

Para conectarlos hay dos mecanismos y no son equivalentes:

```
# A · leer el estado de otro: acoplamiento duro
data "terraform_remote_state" "red" {
  backend = "s3"
  config  = { bucket = "cls-tfstate-prod", key = "red/terraform.tfstate" }
}
resource "aws_instance" "app" {
  subnet_id = data.terraform_remote_state.red.outputs.subnet_app_id
}

# B · consultar la realidad: acoplamiento blando
data "aws_subnet" "app" {
  filter { name = "tag:Name"  values = ["snet-tienda-euw1"] }
}
```

La opción A exige permiso de lectura sobre el estado ajeno — y con él, sobre sus secretos. Ese solo hecho ya la descarta entre equipos distintos.

Y la opción B tiene la ventaja que la clase 086 señaló: no se rompe si el otro estado se recrea. Su riesgo es que el filtro devuelva algo inesperado, y por eso el contrato entre equipos conviene que sea un nombre estable acordado, no una etiqueta que alguien pueda cambiar:

```
mal    filtrar por una etiqueta descriptiva que puede editarse
bien   un nombre acordado y documentado, tratado como interfaz
mejor  un parámetro publicado en un almacén de configuración,
       que el otro equipo escribe y este lee
```

La tercera es la que escala en organizaciones grandes: el equipo de red publica los identificadores en un almacén de parámetros y quien los consume los lee de ahí. El acoplamiento sigue existiendo y es un contrato explícito con un dueño, en vez de una consulta que adivina.

Y los espacios de trabajo merecen la advertencia que la clase 059 anticipó: sirven para varias instancias efímeras del mismo código —una por rama, una por prueba— y no sirven para separar entornos. Comparten configuración de almacén, invitan a condicionales por entorno y hacen fácil aplicar en producción creyendo estar en pruebas. Un directorio por entorno, con su propio estado y su propia configuración, es más ficheros y menos incidentes.

3. Cuando el estado y la realidad discrepan

Antes o después ocurre: el estado dice una cosa y la realidad otra. Hay tres situaciones y tres operaciones, y usar la equivocada destruye recursos.

Existe de verdad y no está en el estado. Un recurso creado a mano, o uno que quedó huérfano al perder un estado.

```
import {
```

```
  to = aws_s3_bucket.facturas
  id = "cls-facturas"
}
```

```
$ terraform plan -generate-config-out=generado.tf
```

La forma declarativa es mejor que la orden equivalente por dos motivos: queda en el repositorio, revisable, y permite generar la configuración a partir del recurso real, que ahorra escribir a mano decenas de campos. Y el flujo correcto es:

```
1. declarar la importación
2. generar la configuración y revisarla
3. planificar: debe salir SIN CAMBIOS
4. si propone cambios, la configuración no coincide: corregirla
5. aplicar, y retirar el bloque de importación
```

El paso 3 es el importante. Una importación que deja el plan proponiendo modificaciones significa que la configuración escrita no describe el recurso que existe, y aplicar así lo modificaría.

Está en el estado y ya no existe. Alguien lo borró por fuera:

```
$ terraform state rm aws_instance.vieja
```

Eso no destruye nada: solo deja de gestionarlo. Y es exactamente lo que hay que hacer cuando se quiere entregar un recurso a otro equipo o a otro estado.

Existe, está en el estado y está mal. Se cambió a mano y la reconciliación no basta:

```
$ terraform apply -replace=aws_instance.app
```

Marca ese recurso para destruir y recrear en el próximo plan, con lo que la decisión se revisa antes de ejecutarse.

Y la operación que más cuesta y más se necesita: mover recursos al refactorizar. La clase 059 avisó de que renombrar un recurso lo destruye y lo recrea, porque su dirección es su identidad. El mecanismo declarativo lo evita:

```
moved {
  from = aws_instance.web
  to   = module.tienda.aws_instance.web
}
```

Con eso, la operación es solo un cambio de nombre en el estado: el plan no propone crear ni destruir nada. Sirve para renombrar, para mover a un módulo, para pasar de índice numérico a clave estable —el arreglo de la reindexación de la clase 059— y queda en el repositorio como registro de por qué se movió.

```
$ terraform plan
Plan: 0 to add, 0 to change, 0 to destroy.
# y en la salida:
#  aws_instance.web has moved to module.tienda.aws_instance.web
```

Esa línea con cero de todo es la comprobación de que el refactor es seguro. Si aparece cualquier otra cosa, la declaración de movimiento no está completa.

4. Perder el estado, y salir de ello

Conviene tratar esto como un simulacro y no como una hipótesis, porque ocurre.

Qué pasa exactamente al perderlo:

```
la infraestructura sigue funcionando: no se destruye nada
Terraform deja de saber que existe
un `apply` intentará CREARLO ENTERO otra vez
  → y fallará en lo que tenga nombres únicos
  → o duplicará lo que no los tenga
```

La segunda consecuencia es la peligrosa: una plantilla aplicada sobre un estado vacío con recursos que ya existen puede crear duplicados en silencio.

La recuperación, por orden de preferencia:

```
1. una versión anterior del almacén
   → es la razón del versionado; recuperación en minutos
2. la copia local que Terraform deja tras una operación
   → `terraform.tfstate.backup`, en la máquina que la ejecutó
3. reconstruirlo importando recurso a recurso
   → días de trabajo, y hay que conocer cada identificador
```

El salto entre la 1 y la 3 es enorme, y es todo el valor del versionado. Y como cualquier copia de seguridad de este programa, hay que probar la restauración:

```
$ aws s3api list-object-versions --bucket cls-tfstate-prod \
    --prefix red/terraform.tfstate --query 'Versions[0:3].[VersionId,LastModified]'
$ aws s3api get-object --bucket cls-tfstate-prod --key red/terraform.tfstate \
    --version-id <id> estado-recuperado.json
$ terraform state push estado-recuperado.json      # con MUCHO cuidado
$ terraform plan -detailed-exitcode                # debe dar 0
```

El último paso es la verificación: un plan sin cambios significa que el estado recuperado corresponde con la realidad. Con cambios, la versión elegida es demasiado antigua y hay que probar otra.

Y dos precauciones que evitan llegar aquí:

```
protección contra el borrado del bucket del estado (clases 042, 059)
y una copia periódica del estado a otro sitio, con su restauración probada
```

La segunda parece redundante con el versionado y no lo es: el versionado protege del borrado de un objeto, no del borrado del bucket ni de la pérdida de acceso a la cuenta.

Y una situación que se confunde con la pérdida y no lo es: el estado bloqueado por una versión de Terraform más nueva.

```
Error: state snapshot was created by Terraform v1.10.2,
  which is newer than current v1.9.8
```

Alguien aplicó con una versión posterior y el estado ya no se puede leer con la anterior. No se ha perdido nada; hay que subir de versión. La corrección de fondo es fijar la versión de la herramienta en la canalización y en los entornos locales, igual que se fijan las de los proveedores.

Y el simulacro que conviene ensayar una vez al año, con la misma disciplina que las restauraciones de las clases 042, 064 y 077:

```
1. copiar el estado de un entorno no productivo
2. borrarlo del almacén
3. recuperarlo de una versión anterior
4. verificar con un plan sin cambios
5. registrar cuánto se tardó
```

Ese último número es el tiempo de recuperación real, y en las cuatro veces que este programa lo ha medido siempre ha sido mayor que el del plan.

5. Higiene del estado

Tres prácticas que mantienen el estado manejable y una lista de comprobación.

No editarlo a mano. Existe la posibilidad y casi nunca es la respuesta correcta:

```
$ terraform state pull > estado.json
# … editar …
$ terraform state push estado.json
```

Cada operación de reparación de esta clase tiene una orden específica, y esas órdenes validan lo que hacen. Editar el fichero salta esa validación, y un error de formato deja el estado ilegible. Si aun así hay que hacerlo —ocurre—, se copia antes y se verifica después con un plan.

Inspeccionarlo cuando algo no cuadra:

```
$ terraform state list | wc -l
$ terraform state list | grep aws_instance
$ terraform state show aws_instance.app
```

La primera cifra es útil como indicador: un estado con miles de recursos hace lenta cada planificación y es una señal de que hay que repartirlo.

Buscar huérfanos periódicamente. Recursos reales que ningún estado gestiona:

```
# lo que existe, según el proveedor
$ aws resourcegroupstaggingapi get-resources --tag-filters Key=gestionado-por,Values=terraform \
    --query 'ResourceTagMappingList[].ResourceARN' --output text | tr '\t' '\n' | sort > reales.txt
# lo que gestiona cada estado
$ for d in red datos plataforma tienda; do (cd $d && terraform state pull \
    | jq -r '.resources[].instances[].attributes.arn // empty'); done | sort > gestionados.txt
```

```
$ comm -23 reales.txt gestionados.txt
```

Esa comparación es el equivalente de la que la clase 049 hacía entre proyectos facturados y gobernados, y produce el mismo tipo de hallazgo: recursos que nadie sabe que existen y que siguen costando.

Y la lista de comprobación de la clase:

- estado remoto, con bloqueo, versionado y cifrado
- en un proyecto o cuenta propios, con acceso mínimo separado de lectura y escritura
- un estado por radio de impacto, no uno para todo
- conexión entre estados por fuente de datos o parámetro publicado, no leyendo el estado ajeno
- un directorio por entorno; espacios de trabajo solo para instancias efímeras
- tiempo de espera de bloqueo en la canalización
- versión de la herramienta fijada, como las de los proveedores
- importaciones declaradas, verificadas con un plan sin cambios
- refactores con declaración de movimiento, verificados con cero de todo
- recuperación del estado ensayada, con su duración registrada
- búsqueda periódica de recursos huérfanos

Once puntos, de los cuales tres son comprobaciones. Y la que más veces salva es la novena: un refactor que propone crear o destruir algo no es un refactor.

Ejemplo trabajado

CloudShop reparte su estado y descubre, al hacerlo, que llevaba catorce meses gestionando menos infraestructura de la que creía. Los cuatro hallazgos son de estado y ninguno se ve desde la plantilla.

Hallazgo 1 — un estado, cuatro equipos y una cola.

```
recursos en el estado                      1.847
duración de una planificación              3 min 40 s
ejecuciones bloqueadas al día                ~12
equipos que comparten el estado                4
radio de un `destroy` accidental             todo
```

El reparto por radio de impacto:

```
estados                                        1                4
recursos por estado                        1.847        612 / 388 / 501 / 346
duración de una planificación          3 min 40 s           38 s
esperas por bloqueo al día                   ~12              0
```

Y la separación se hizo sin destruir nada, con movimientos declarados entre estados:

```
$ terraform state mv -state-out=../datos/terraform.tfstate \
    aws_db_instance.pedidos aws_db_instance.pedidos
```

Hallazgo 2 — 34 recursos huérfanos.

Al comparar lo que existe con lo que gestiona algún estado:

```
recursos etiquetados como gestionados por Terraform     412
recursos que aparecen en algún estado                   378
huérfanos                                                34
```

De los 34: dieciocho venían de un estado que se perdió catorce meses atrás y se rehízo desde cero, once eran de un entorno retirado que nadie limpió, y cinco los había creado alguien a mano con la etiqueta puesta.

```
huérfanos                                    34                0
  importados al estado correspondiente        21
  eliminados por estar sin uso                13
costo mensual de los eliminados            118 USD             —
búsqueda de huérfanos                       nunca          mensual
```

Y las veintiuna importaciones se hicieron con bloques declarados y generación de configuración, con la comprobación del plan sin cambios en cada una. Tres de ellas la fallaron la primera vez, porque la configuración generada no coincidía con lo que el equipo pensaba que había.

Hallazgo 3 — el bloqueo que nadie sabía liberar.

```
Error: Error acquiring the state lock
  Who: agente@ci-runner-7   Created: hace 3 días
```

Un agente efímero se había destruido a mitad de una aplicación. Durante tres días, nadie pudo aplicar nada en ese estado y dos personas intentaron liberarlo sin comprobar si la ejecución seguía viva.

```
procedimiento de liberación              no había      documentado, con la
                                                       comprobación previa
tiempo de espera de bloqueo en la
  canalización                           no había          10 min
días bloqueado                                3              —
apagado ordenado del agente              no había     libera el bloqueo al salir
```

Hallazgo 4 — el simulacro de pérdida, y lo que costó de verdad.

En preproducción se borró el estado a propósito:

```
recuperación desde una versión anterior         4 min
verificación con plan sin cambios               correcta
```

Y después se hizo el ejercicio completo, suponiendo que el versionado también se hubiera perdido:

```
reconstrucción por importación
  recursos a importar                           388
  identificadores localizables automáticamente  312
  identificadores que hubo que buscar a mano     76
  tiempo estimado                            3-4 días
```

Cuatro minutos frente a cuatro días. Esa comparación es la que justificó dos medidas que llevaban un año propuestas y sin hacer:

```
protección contra el borrado del bucket        no              sí
copia del estado a otra cuenta                 no          diaria, con
                                                           restauración probada
simulacro de pérdida                         nunca          anual
tiempo de recuperación documentado           "rápido"       4 min medidos
```

Y una comprobación que se añadió y encontró algo el primer día:

```
$ terraform state pull | jq -r '.resources[].instances[].attributes
  | to_entries[] | select(.key | test("password|secret|private_key|token"))
  | .key' | sort -u | wc -l
9
```

Nueve campos sensibles en el estado de datos, con doce identidades con permiso de lectura sobre el bucket. La corrección fue la de la clase 059 —proyecto propio, acceso mínimo, rotación de lo expuesto— y se hizo el mismo día.

Resumen:

```
estados                                        1              4
duración de una planificación                3 min 40 s       38 s
esperas por bloqueo al día                     ~12             0
recursos huérfanos                             34              0
identidades con acceso al estado               12              2
recuperación del estado ensayada               no         sí, 4 min
copia del estado fuera de la cuenta            no           diaria
```

La lección que esta clase traslada al resto de la parte 07: el estado es el único componente cuya pérdida no destruye nada y aun así deja la plataforma ingobernable. Los cuatro hallazgos comparten una propiedad —ninguno se ve mirando las plantillas— y tres de ellos existían desde hacía más de un año. La comparación que los destapó es la misma que la clase 049 hacía con los proyectos: lo que existe frente a lo que alguien gestiona, y la diferencia siempre tiene sorpresas.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/087-estado-remoto-locking-cifrado-y-
recuperacion/lab.py
```

El laboratorio selecciona el motor de práctica iac 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 backend-terraform 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 plan reproducible sin secretos ni cambios inesperados. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye backend-terraform 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
Dos ejecuciones simultáneas dejan el estado sin corresponder con la realidad	No hay bloqueo, o se liberó uno cuya ejecución seguía viva	Activa el bloqueo, pon tiempo de espera en la canalización y comprueba que el agente que lo tomó ya no existe antes de liberarlo.
Una importación deja el plan proponiendo modificaciones	La configuración escrita no describe el recurso que existe	Genera la configuración a partir del recurso real y exige un plan sin cambios antes de aplicar.
Renombrar o mover un recurso propone destruirlo y crearlo	La dirección del recurso es su identidad en el estado	Declara el movimiento y verifica que el plan da cero de todo; si propone algo, la declaración está incompleta.
Se pierde el estado y un apply intenta crearlo todo otra vez	Terraform deja de saber que la infraestructura existe	Recupera de una versión anterior y verifica con un plan sin cambios; protege el bucket y copia el estado fuera de la cuenta.
Existen recursos reales que ningún estado gestiona	Estados perdidos, entornos retirados a medias o creación manual	Compara periódicamente lo que existe con lo que gestiona algún estado, e importa o elimina cada diferencia.
Un estado no se puede leer tras una ejecución de otra persona	Se aplicó con una versión más nueva de la herramienta	Fija la versión de Terraform en la canalización y en los entornos locales, igual que las de los proveedores.

■ 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 cuatro propiedades necesita un almacén de estado y qué protege cada una?

2. ¿Por qué leer el estado de otro equipo es peor que consultar la realidad con una fuente de datos?
3. Describe las tres discrepancias posibles entre estado y realidad y la operación que corresponde a cada una.
4. ¿Qué comprobación demuestra que un refactor con movimiento declarado es seguro?
5. ¿Qué se pierde exactamente al perder el estado, y cuál es la diferencia de tiempo entre recuperarlo y reconstruirlo?

Referencias

• HashiCorp (2025). State: purpose and remote backends — por qué existe y cómo se guarda. <https://developer.hashicorp.com/terraform/language/state>

• HashiCorp (2025). State locking — bloqueo, tiempos de espera y liberación forzada. <https://developer.hashicorp.com/terraform/language/state/locking>

• HashiCorp (2025). Import blocks and configuration generation — importar declarativamente y generar la configuración. <https://developer.hashicorp.com/terraform/language/import>

• HashiCorp (2025). The moved block — refactorizar sin destruir ni recrear. <https://developer.hashicorp.com/terraform/language/moved>

• HashiCorp (2025). Manipulating state with the CLI — listar, mostrar, mover y retirar recursos. <https://developer.hashicorp.com/terraform/cli/state>

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

Descargar: Parte 07 en PDF · Manual integral

Anterior	Índice	Siguiente
← 086 · Terraform: HCL, providers, resources y grafo	Parte 07 · Programa	088 · Módulos, contratos, versiones y composición →

Evaluación — 087 Estado remoto, locking, cifrado y recuperación

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 backend-terraform 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.

088 — Módulos, contratos, versiones y composición

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: iac · Estado: EXECUTABLECORE

Propósito

Escribir módulos que otros equipos usen de verdad, que es un problema de diseño de interfaces y no de sintaxis. Los dos fracasos habituales son simétricos —una envoltura que no aporta nada y un módulo con cuarenta variables que intenta cubrirlo todo— y ambos vienen del mismo error: tratar el módulo como una plantilla de recursos en vez de como la codificación de una decisión. La clase fija ese criterio, y añade la disciplina de versionado que las clases 047, 059 y 081 ya exigieron a las dependencias.

Resultados de aprendizaje

Al finalizar podrás:

1. Decidir si algo merece ser un módulo, con un criterio que descarta las envolturas vacías.
2. Diseñar la interfaz de un módulo por la decisión que codifica, no por los campos del recurso.
3. Versionar y consumir módulos de forma que un cambio no llegue a nadie sin que lo acepte.
4. Componer módulos sin crear jerarquías que nadie pueda depurar.
5. Probar un módulo creando y destruyendo de verdad, no leyéndolo.

Conceptos centrales

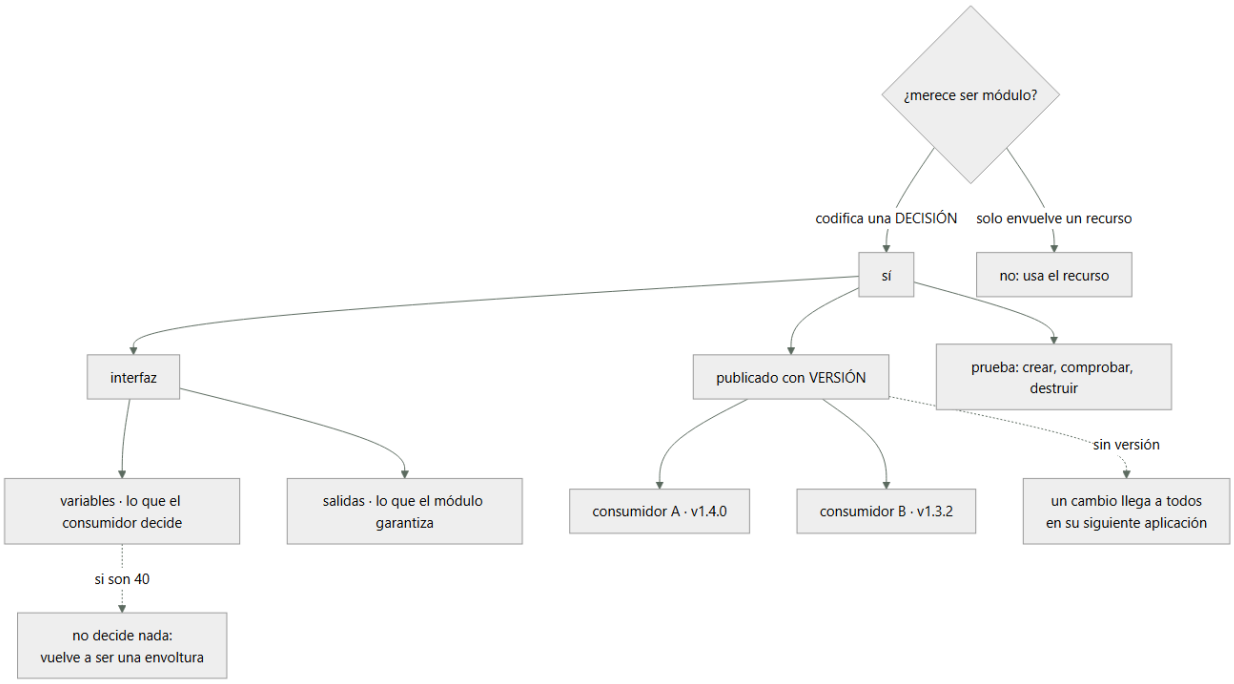
Concepto	Comprensión verificable
módulo	Directorio con entradas y salidas. Su valor no está en agrupar recursos sino en codificar una decisión que no haya que volver a tomar.
interfaz	Las variables y las salidas. Es lo único que los consumidores ven, así que cambiarla es un cambio incompatible aunque el interior no cambie.
envoltura vacía	Módulo que expone tantas variables como campos tiene el recurso. No decide nada y añade una capa que hay que mantener.
módulo que lo cubre todo	Módulo con decenas de variables opcionales y condicionales para cada caso. Nadie sabe qué combinaciones funcionan porque no se pueden probar todas.
origen versionado	Referencia a un módulo por etiqueta o revisión concreta. Sin ella, un cambio publicado llega a todos los consumidores en su siguiente aplicación.
prueba de módulo	Ejecución que crea recursos de verdad, comprueba y destruye. Es lo único que demuestra que las combinaciones declaradas funcionan.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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. Un módulo codifica una decisión, no agrupa recursos

La pregunta «¿esto debería ser un módulo?» se responde mal casi siempre, porque se responde por tamaño. El criterio útil es otro:

```
¿hay una DECISIÓN que no quiero que nadie vuelva a tomar?
  → sí: es un módulo
  → no: es un recurso, y usarlo directamente es más claro
```

Y la diferencia se ve en el nombre. Estos no son módulos:

```
modulo-bucket           envuelve un recurso y expone sus mismos campos
modulo-instancia        igual
modulo-red              agrupa tres recursos sin decidir nada
```

Estos sí:

```
bucket-conforme         un bucket con la línea base de la organización:
                        acceso uniforme, sin acceso público, versionado,
                        ciclo de vida, cifrado y etiquetas obligatorias
                        → codifica las decisiones de las clases 041, 049, 053

servicio-web            todo lo que hace falta para exponer un servicio:
                        despliegue, servicio, entrada, presupuesto,
                        política de red y objeto de escalado, con los
                        valores por defecto que la organización ya decidió

entorno-efimero         una copia completa de un entorno, con las
                        decisiones de nombre, tamaño y caducidad ya tomadas
```

La diferencia práctica es que un consumidor del primer grupo tiene que saber tanto como si escribiera el recurso, y uno del segundo no:

```
module "facturas" {
  source  = "git::ssh://git@interno/plataforma/modulos.git//bucket-conforme?ref=v2.3.0"
  nombre  = "cls-facturas"
  retencion_dias = 2555        # 7 años, requisito legal
  equipo  = "pedidos"
}
```

Cuatro entradas. Todo lo demás lo decidió la organización una vez, y quien use el módulo no puede equivocarse en ello. Ese es el valor real: no ahorrar líneas sino hacer imposible el error.

Y de ahí sale el indicador que detecta un módulo mal diseñado:

```
si el número de variables se acerca al número de campos del recurso,
el módulo no está decidiendo nada
```

Es una envoltura, y una envoltura es peor que nada: añade una capa que mantener, una versión que seguir y un sitio más donde buscar cuando algo falla.

Y el fracaso simétrico, el módulo que intenta cubrir todos los casos:

```
module "servicio" {
  source = "…"
  tipo_de_almacenamiento = var.tipo   # 4 opciones
  con_cache              = true       # cambia 6 recursos
  modo_red               = "privado"  # 3 opciones
  # … y 41 variables más
}
```

Con veinte variables opcionales hay más de un millón de combinaciones, de las que se probarán tres. El resto no se sabe si funcionan, y el consumidor lo descubre aplicando.

La salida es dividir por decisión, no por opción:

```
en vez de un módulo con una variable que elige entre cuatro almacenamientos
  → cuatro módulos, o tres, o los que de verdad se usen
  y cada uno probado
```

Y conviene aceptar la consecuencia: un módulo que solo sirve para un caso es un buen módulo si ese caso es el que la organización tiene. La generalidad se añade cuando aparece el segundo consumidor real, no antes.

2. La interfaz es el contrato, y cambiarla es incompatible

Los consumidores solo ven las variables y las salidas. Todo lo demás es implementación y se puede cambiar; la interfaz, no.

Las variables merecen tipo, descripción y validación, porque son documentación ejecutable:

```
variable "retencion_dias" {
  type        = number
  description = "Días que se conservan los objetos antes de eliminarse."
  validation {
    condition     = var.retencion_dias >= 30 && var.retencion_dias <= 3650
    error_message = "La retención debe estar entre 30 días y 10 años."
  }
}

variable "equipo" {
  type        = string
  description = "Equipo responsable. Se usa como etiqueta de atribución de costo."
  validation {
    condition     = can(regex("^[a-z][a-z0-9-]{2,20}$", var.equipo))
    error_message = "Nombre de equipo en minúsculas, guiones, de 3 a 21 caracteres."
  }
}
```

La validación convierte un error de aplicación en un error de planificación, que es donde cuesta menos. Y el mensaje debe decir qué se esperaba, no que el valor es inválido.

Y un tipo compuesto vale más que seis variables sueltas cuando los valores van juntos:

```
variable "escalado" {
  type = object({
    minimo   = number
    maximo   = number
    objetivo = number
  })
  default = { minimo = 2, maximo = 10, objetivo = 70 }
  validation {
    condition     = var.escalado.minimo >= 2
    error_message = "El mínimo no puede ser menor que 2: una réplica no es alta disponibilidad (clase
074)."
  }
}
```

Ese mensaje de error hace algo más que validar: transmite la decisión y la clase donde se justificó.

Las salidas son la garantía, y conviene que sean pocas y estables:

```
output "nombre"  { value = aws_s3_bucket.este.bucket }
output "arn"     { value = aws_s3_bucket.este.arn }
```

Exponer el objeto entero de un recurso es cómodo y ata a los consumidores a la estructura interna: cambiar el recurso rompe a todos. Y una salida marcada como sensible no se cifra en el estado —la advertencia de la clase 087— así que marcarla protege la pantalla y no el fichero.

Y la regla de compatibilidad, que es la de cualquier interfaz:

```
compatible          añadir una variable con valor por defecto
                    añadir una salida
                    cambiar la implementación sin cambiar el resultado

INCOMPATIBLE        quitar o renombrar una variable o una salida
                    hacer obligatoria una variable que era opcional
                    cambiar el valor por defecto de algo que crea recursos
                    cualquier cambio que produzca destrucción y recreación
```

La última merece atención porque no es evidente: un cambio interno que altere un campo que fuerza reemplazo destruye recursos de los consumidores en su siguiente aplicación. Eso es un cambio incompatible aunque la interfaz no se haya tocado, y hay que publicarlo como tal y con aviso.

Y para retirar algo sin romper, el mismo patrón de expandir y contraer de las clases 071 y 079:

```
1. añadir lo nuevo, marcar lo viejo como obsoleto en la descripción
2. publicar una versión menor; los consumidores migran cuando quieran
3. retirar lo viejo en una versión mayor, con aviso previo
```

3. Versionar, o el cambio llega solo

Este es el error operativo de la clase y su consecuencia es inmediata:

```
# sin versión: la siguiente aplicación de CUALQUIER consumidor
# se lleva lo último que haya en la rama
module "bucket" {
  source = "git::ssh://git@interno/plataforma/modulos.git//bucket-conforme"
}
```

Un cambio publicado el martes llega a producción de otro equipo el miércoles, sin que nadie lo haya decidido. Es exactamente el argumento de las clases 047 —módulos de Bicep por versión— y 062 —base fijada por huella—, con un tercer mecanismo.

```
module "bucket" {
  source = "git::ssh://git@interno/plataforma/modulos.git//bucket-conforme?ref=v2.3.0"
}
```

Y con un registro de módulos, la restricción se expresa como en cualquier gestor de dependencias:

```
module "bucket" {
  source  = "interno.example/plataforma/bucket-conforme/aws"
  version = "~> 2.3"
}
```

Los tres orígenes y cuándo usa cada uno:

```
ruta local        módulos de este mismo repositorio, que se versionan con él
git con revisión  lo habitual en una organización sin registro propio
registro          cuando hay varios consumidores y conviene descubrirlos
```

Y una advertencia sobre la ruta local que produce sorpresas: un módulo local no se versiona por separado, así que un cambio afecta a todo lo que hay en ese repositorio a la vez. Está bien para lo que se despliega junto y mal para lo que comparten equipos distintos.

Y la disciplina de publicación, que es la que hace utilizable un catálogo de módulos:

```
cada versión con notas: qué cambia, si es incompatible y qué hay que hacer
las incompatibles solo en versiones mayores
un ejemplo mínimo funcionando dentro del módulo
y la fecha de retirada de las versiones antiguas
```

La última se olvida y produce el problema contrario al de no versionar: veinte consumidores en catorce versiones distintas, ninguna de las cuales se puede retirar porque alguien la usa. Una política de soporte —las dos últimas versiones mayores, por ejemplo— hay que decidirla antes de tener el problema.

Y una nota sobre la actualización, que conecta con la clase 067: igual que las imágenes base fijadas necesitan un proceso que proponga actualizarlas, los módulos fijados lo necesitan. Un robot que abra un cambio cuando hay versión nueva convierte la fijación en algo sostenible en vez de en deuda.

4. Componer sin construir una torre

Los módulos pueden llamar a otros módulos, y ahí empieza el problema de mantenibilidad.

```
profundidad 1   el módulo raíz llama a módulos                  bien
profundidad 2   esos módulos llaman a módulos comunes           aceptable
profundidad 3+  cada nivel oculta el anterior                   difícil de depurar
```

El coste de la profundidad es concreto y se nota al diagnosticar:

```
cada nivel añade un prefijo a la dirección de los recursos
  module.plataforma.module.red.module.subred.aws_subnet.este[0]
cada nivel puede transformar los valores que pasa
  → el valor que llega al recurso no se parece al que se escribió
cada nivel tiene su propia versión
  → averiguar qué versión efectiva se está usando exige recorrerlos
```

Y dos reglas que mantienen esto manejable:

```
1. un módulo no configura proveedores; los recibe
   configurar un proveedor dentro impide usar alias desde fuera
   y complica retirarlo del estado

2. un módulo no lee el estado de nadie ni consulta datos globales
   recibe lo que necesita como variable
   → así se puede probar de forma aislada
```

La segunda parece restrictiva y es lo que hace probable un módulo. Uno que consulta la realidad para encontrar su red solo funciona donde esa red existe con ese nombre; uno que la recibe funciona en cualquier sitio, incluida una prueba.

Y el antipatrón que aparece en organizaciones grandes: el módulo que despliega un entorno completo.

```
module "entorno" {
  source = "…/entorno-completo?ref=v4.1.0"
  nombre = "produccion"
}
```

Parece la culminación y trae tres problemas: el radio de impacto vuelve a ser todo (clases 047, 087), cualquier cambio del módulo afecta al entorno entero, y la planificación tarda lo que tardaba antes de repartir el estado. La composición correcta ocurre en el módulo raíz de cada unidad de despliegue, no dentro de un módulo:

```
# tienda/main.tf — un estado, una unidad de despliegue
module "servicio"      { source = "…/servicio-web?ref=v3.1.0"    /* … */ }
module "cola"          { source = "…/cola-conforme?ref=v1.7.0"   /* … */ }
module "bucket"        { source = "…/bucket-conforme?ref=v2.3.0" /* … */ }
```

Y para lo que de verdad hay que repetir muchas veces, la iteración sobre módulos, con la advertencia de la clase 059 sobre la identificación por clave estable:

```
module "colas" {
  source   = "…/cola-conforme?ref=v1.7.0"
  for_each = { pedidos = 5, facturas = 3, avisos = 10 }
  nombre   = each.key
  reintentos = each.value
}
```

5. Probar un módulo es crear y destruir

Un módulo que nadie ha ejecutado no está probado, por bien que se lea. Y hay tres niveles de comprobación con coste creciente:

```
estático        formato, sintaxis y reglas de estilo: segundos
plan            que planifica sin error, con valores de ejemplo: segundos
aplicación real  crea, comprueba y destruye: minutos y dinero
```

Los dos primeros detectan errores de escritura; solo el tercero demuestra que la combinación funciona.

```
# pruebas/basico.tftest.hcl
variables {
  nombre         = "cls-prueba-modulo"
  retencion_dias = 30
```

```
  equipo         = "plataforma"
}

run "crea_conforme" {
  command = apply

  assert {
    condition     = aws_s3_bucket_public_access_block.este.block_public_acls
    error_message = "el bucket debe bloquear el acceso público (clase 053)"
  }
  assert {
    condition     = aws_s3_bucket_versioning.este.versioning_configuration[0].status == "Enabled"
    error_message = "el versionado debe estar activo"
  }
}

run "rechaza_retencion_invalida" {
  command = plan
  variables { retencion_dias = 5 }
  expect_failures = [var.retencion_dias]
}
```

El segundo bloque es una prueba negativa en el sentido exacto que este programa ha usado desde la clase 046: comprueba que algo que no debe funcionar, falla. Y es la mitad que casi nunca se escribe.

Y las condiciones que hacen sostenible probar de verdad:

```
una cuenta o proyecto propio para pruebas, con presupuesto y alerta
nombres únicos por ejecución, para que dos pruebas no choquen
destrucción garantizada, incluso si la prueba falla
y una limpieza periódica de lo que se quedó por el camino
```

La última no es opcional: una prueba interrumpida deja recursos, y en un año eso es una factura. La comparación de la clase 087 —lo que existe frente a lo que gestiona algún estado— aplicada a la cuenta de pruebas es la forma de encontrarlos.

Y la lista de comprobación de la clase:

- el módulo codifica una decisión; su número de variables lo demuestra
- toda variable con tipo, descripción y validación cuando aplique
- salidas pocas y estables; nunca el objeto entero de un recurso
- no configura proveedores ni consulta datos globales
- publicado con versión, y consumido por versión
- notas de cada versión, con las incompatibilidades señaladas
- un ejemplo mínimo funcionando dentro del propio módulo
- pruebas que crean y destruyen, con al menos una prueba negativa
- política de soporte de versiones antiguas, decidida
- proceso que proponga actualizar a los consumidores

Diez puntos, de los cuales cuatro son sobre la interfaz. Y esa proporción es la tesis de la clase: un módulo es una interfaz con implementación detrás, y casi todo su valor —y casi todos sus problemas— están en la interfaz.

Ejemplo trabajado

CloudShop tiene un catálogo de módulos que nadie usa. Tres equipos han escrito los suyos por su cuenta y el equipo de plataforma no entiende por qué. El análisis da tres razones y una de ellas explica las otras dos.

El diagnóstico.

```
módulos publicados                    14
módulos usados por más de un equipo    2
variables del módulo más usado        47
módulos con pruebas                     0
módulos con versión fijada por los consumidores   3 de 9 usos
```

La entrevista con los tres equipos dio la misma respuesta con tres formulaciones:

```
"tardo más en entender el módulo que en escribir el recurso"
"no sé qué combinación de opciones funciona"
"la última vez que lo usé me rompió producción al actualizarse solo"
```

Razón 1 — el módulo de 47 variables no decidía nada.

```
$ grep -c '^variable' modulos/servicio/variables.tf
```

```
47
$ grep -c 'resource ' modulos/servicio/main.tf
11
```

Cuarenta y siete entradas para once recursos: el módulo exponía prácticamente todos sus campos. Un consumidor tenía que saber tanto como para escribirlos.

La reescritura por decisión, no por opción:

```
variables                                    47              9
valores por defecto decididos por la
  organización                                3             31
condicionales internos                       19              2
combinaciones posibles                   >1 millón           12
combinaciones probadas                       0              12
líneas que escribe un consumidor            ~60              9
```

Las nueve variables restantes son las que de verdad varían entre servicios: nombre, equipo, imagen, puerto, tamaño, escalado, dominio, dependencias y ventana de mantenimiento. Todo lo demás lo decidió la organización una vez.

Razón 2 — no había forma de saber qué funcionaba.

Se escribieron pruebas que crean y destruyen de verdad:

```
casos probados                               12
de ellos, negativos                           5
tiempo de la suite                       14 min
costo mensual de la cuenta de pruebas    ~18 USD
fallos encontrados la primera ejecución       4
```

Los cuatro fallos son el dato interesante: cuatro combinaciones que el módulo declaraba admitir y que nunca habían funcionado. Una de ellas era la que un equipo había intentado usar seis meses antes, y por la que había decidido escribir el suyo.

Razón 3 — el módulo cambiaba bajo los pies.

```
$ grep -rn 'source.*modulos.git' entornos/ | grep -c 'ref='
3
$ grep -rn 'source.*modulos.git' entornos/ | grep -vc 'ref='
6
```

Seis de nueve usos sin versión fijada. Y el historial mostraba el incidente que había roto la confianza:

```
un cambio en el módulo alteró un campo que fuerza reemplazo
seis equipos aplicaron esa semana por otros motivos
tres bases de datos marcadas para recreación
dos detenidas a tiempo en la revisión del plan
una aplicada: 40 minutos de corte
```

El cambio del módulo era compatible en su interfaz y destructivo en su efecto — exactamente el caso que esta clase señala como incompatible aunque no lo parezca.

```
usos con versión fijada                    3 de 9            9 de 9
notas por versión                         no había      obligatorias, con
                                                         incompatibilidades
cambios que fuerzan reemplazo            sin marcar     versión mayor + aviso
robot que propone actualizar               no                sí
protección contra destrucción en los
  recursos con datos                   0 de 6            6 de 6
```

La última fila es la que habría evitado el corte: con protección contra destrucción, el plan habría fallado en vez de recrear.

Y el resultado a los tres meses:

```
módulos publicados                           14              6
módulos usados por más de un equipo           2              6
equipos con módulos propios duplicados        3              0
variables del módulo principal               47              9
módulos con pruebas                           0              6
usos con versión fijada                    3 de 9        22 de 22
tiempo de un servicio nuevo               2 días          40 minutos
```

Ocho módulos retirados: cinco eran envolturas que no decidían nada y tres cubrían casos que ya no existían.

Y la fila de abajo es la que convenció a los tres equipos: cuarenta minutos para desplegar un servicio nuevo completo, con la línea base de seguridad, red, observabilidad y presupuesto de interrupción ya aplicadas — porque el módulo codifica las decisiones de las clases 041 a 084 y no hay que volver a tomarlas.

La lección que esta clase traslada al resto de la parte 07: los tres equipos no rechazaban los módulos por preferencia. Los rechazaban porque no ahorraban trabajo, no se sabía qué funcionaba y cambiaban solos. Las tres razones son de diseño de interfaz y de disciplina de publicación, no de la herramienta. Y la reescritura de 47 variables a 9 no quitó capacidad: quitó decisiones que nadie debería estar tomando dos veces.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/088-modulos-contratos-versiones-y-composicion/lab.py
```

El laboratorio selecciona el motor de práctica iac 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 modulo-terraform 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 plan reproducible sin secretos ni cambios inesperados. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye modulo-terraform 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 equipos escriben sus propios recursos en vez de usar el módulo	El módulo expone casi tantas variables como campos tiene el recurso: no decide nada	Rediseña por decisión: cuenta cuántas variables varían de verdad entre consumidores y fija el resto.
Nadie sabe qué combinaciones de opciones funcionan	El módulo intenta cubrir todos los casos y las combinaciones no se pueden probar	Divide en módulos por decisión y prueba cada uno creando y destruyendo de verdad.
Un cambio del módulo llega a producción de otro equipo sin que nadie lo decida	El origen no fija versión	Consume siempre por etiqueta o restricción de versión, y ten un robot que proponga la actualización.
Un cambio compatible del módulo destruye recursos de los consumidores	Alteró un campo que fuerza reemplazo, aunque la interfaz no cambiara	Trátalo como incompatible: versión mayor y aviso; y protege contra destrucción los recursos con datos.
Un módulo no se puede probar de forma aislada	Consulta datos globales o configura sus propios proveedores	Que reciba lo que necesita como variable y los proveedores desde fuera.
Hay veinte consumidores repartidos en catorce versiones y ninguna se puede retirar	No hay política de soporte de versiones antiguas	Decide cuántas versiones se soportan y anuncia las retiradas con antelación.

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 pregunta decide si algo merece ser un módulo, y qué indicador delata una envoltura vacía?
2. ¿Qué cambios de un módulo son incompatibles aunque la interfaz no cambie?
3. ¿Por qué consumir un módulo sin fijar versión equivale a aceptar cambios que nadie decidió?
4. ¿Por qué un módulo no debe configurar proveedores ni consultar datos globales?
5. ¿Qué demuestra una prueba que crea y destruye que no demuestra un plan correcto?

Referencias

- HashiCorp (2025). Module composition — profundidad, proveedores y patrones de composición. <https://developer.hashicorp.com/terraform/language/modules/develop/composition>
- HashiCorp (2025). Module sources and versions — orígenes, revisiones y restricciones de versión. <https://developer.hashicorp.com/terraform/language/modules/sources>
- HashiCorp (2025). Variable validation and custom conditions — validación en la planificación. <https://developer.hashicorp.com/terraform/language/expressions/custom-conditions>
- HashiCorp (2025). Terraform test framework — pruebas que aplican y destruyen, y fallos esperados. <https://developer.hashicorp.com/terraform/language/tests>
- HashiCorp (2025). Standard module structure — ejemplos, documentación y organización esperada. <https://developer.hashicorp.com/terraform/language/modules/develop/structure>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 07 en PDF · Manual integral

Evaluación — 088 Módulos, contratos, versiones y composición

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 modulo-terraform 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.

089 — Variables, outputs, locals y data sources

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: iac · Estado: EXECUTABLECORE

Propósito

Decidir de dónde salen los valores, que es el problema real detrás de la promoción entre entornos. El orden de precedencia de las variables tiene una trampa concreta —hay ficheros que se cargan solos y otros que no— y de ahí salen las aplicaciones con los valores del entorno equivocado. La clase fija además la regla que las clases 081 y 085 midieron sin enunciarla: la diferencia entre dos entornos debe caber en un fichero que alguien pueda leer entero, y todo lo que no esté ahí es erosión, no decisión.

Resultados de aprendizaje

Al finalizar podrás:

1. Ordenar las fuentes de valores por precedencia y detectar cuáles se cargan sin pedirlo.
2. Tipar las entradas con estructuras y campos opcionales en vez de decenas de variables sueltas.
3. Centralizar convenciones de nombres y etiquetas en un solo sitio calculado.
4. Distinguir cuándo una fuente de datos es correcta y cuándo esconde un acoplamiento.
5. Expresar la diferencia entre entornos en un fichero corto y revisable.

Conceptos centrales

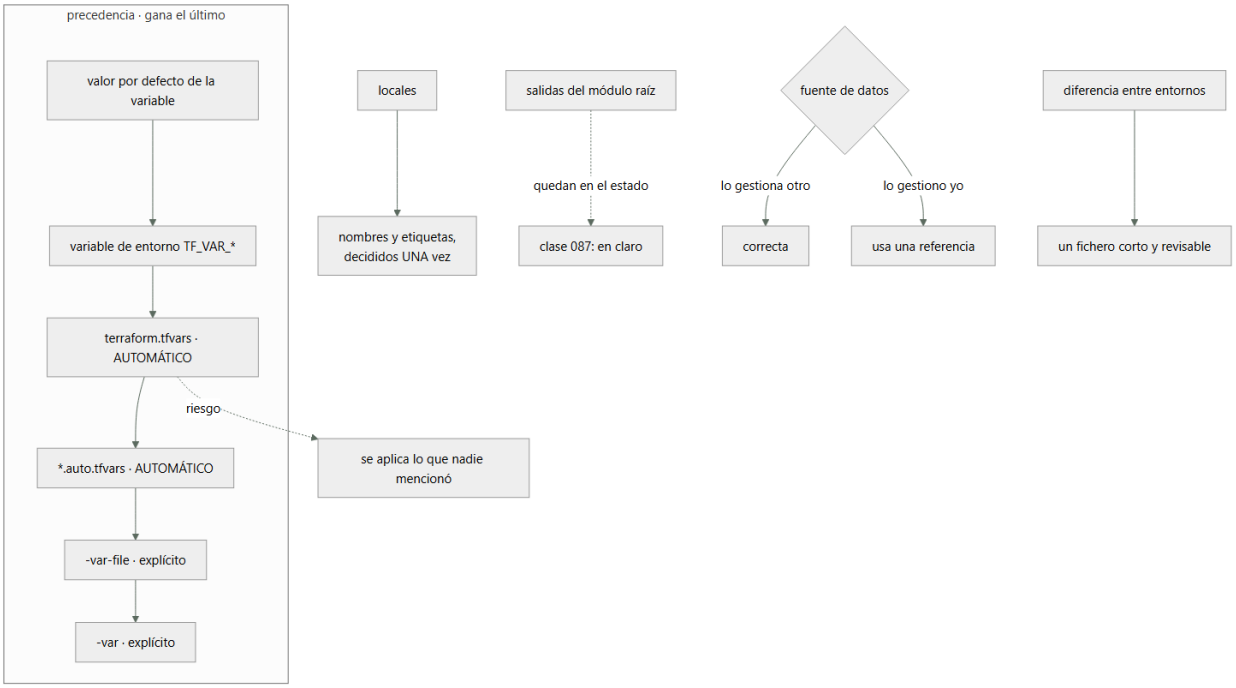
Concepto	Comprensión verificable
precedencia de variables	Orden en que se resuelven las fuentes de un valor. Lo declarado más tarde gana, y algunos ficheros se cargan automáticamente sin mencionarlos.
carga automática	Ficheros que se leen sin indicarlos. Es cómodo y es el mecanismo por el que se aplica una configuración distinta de la que se creía.
atributo opcional	Campo de un objeto con valor por defecto. Permite una sola variable estructurada en vez de una decena de variables sueltas.
valor local	Cálculo con nombre dentro de un módulo. Su mejor uso es centralizar convenciones —nombres y etiquetas— para que se decidan una vez.
fuente de datos	Lectura de algo que gestiona otro. Correcta para lo ajeno; para lo propio, una referencia directa es siempre mejor.
diferencia declarada frente a erosión	Lo que cambia entre entornos por decisión, frente a lo que cambia porque alguien editó una copia. Solo lo primero cabe en un fichero corto.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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. De dónde sale un valor, y cuál gana

Un valor puede venir de seis sitios, y el orden importa porque el último gana:

```
1. el valor por defecto de la variable
2. una variable de entorno con el prefijo convenido
3. terraform.tfvars           ← se carga SOLO
4. cualquier fichero *.auto.tfvars   ← se cargan SOLOS, por orden alfabético
5. -var-file, en el orden en que se indiquen
6. -var
```

Las líneas 3 y 4 son la trampa de esta clase. Un fichero con esos nombres se lee sin que nadie lo mencione, así que basta con que exista en el directorio para que sus valores se apliquen. Y el escenario que produce el incidente es prosaico:

```
alguien copia produccion.tfvars a la carpeta para consultar un valor
lo renombra a terraform.tfvars por costumbre, o lo deja como algo.auto.tfvars
la siguiente ejecución en esa carpeta usa esos valores
y el plan parece razonable, porque los nombres de recurso coinciden
```

La protección es doble y conviene aplicar las dos:

```
1. ningún fichero con nombre de carga automática en el repositorio
   → los valores por entorno se pasan siempre con -var-file explícito
2. que el propio código verifique en qué entorno cree estar
```

La segunda se consigue con una comprobación en la planificación:

```
variable "entorno" {
  type = string
  validation {
    condition     = contains(["dev", "pre", "pro"], var.entorno)
    error_message = "Entorno no reconocido."
  }
}

check "entorno_coincide_con_el_estado" {
  assert {
    condition     = can(regex(var.entorno, data.terraform_remote_state.este.workspace))
    error_message = "El entorno declarado no coincide con el estado en uso."
  }
}
```

Y una comprobación más simple y muy efectiva: que el nombre del almacén de estado incluya el entorno y que una validación lo compare con la variable. Aplicar con los valores de producción sobre el estado de preproducción deja de ser posible en silencio.

Y sobre las variables de entorno, dos avisos:

```
son invisibles en la revisión: nadie las ve en el pull request
y persisten en la sesión: una exportación olvidada afecta a la siguiente carpeta
```

Por eso conviene reservarlas para lo que de verdad no debe estar en un fichero —credenciales que la canalización inyecta— y no para configuración.

2. Tipar bien reduce el número de variables

La clase 088 midió el problema: cuarenta y siete variables sueltas. Una parte de esa inflación es de diseño y otra es de tipado.

Una variable estructurada con campos opcionales sustituye a media docena:

```
variable "escalado" {
  description = "Política de escalado del servicio."
  type = object({
    minimo          = optional(number, 2)
    maximo          = optional(number, 10)
    objetivo_cpu    = optional(number, 70)
    ventana_bajada  = optional(number, 300)
  })
  default = {}
}
```

Con eso, un consumidor que no quiera decidir nada escribe nada, y quien quiera cambiar un campo escribe solo ese:

```
escalado = { maximo = 40 }
```

Y los tres beneficios frente a variables sueltas:

```
los valores que van juntos se ven juntos
añadir un campo opcional NO es un cambio incompatible (clase 088)
la validación puede comprobar la coherencia ENTRE campos
```

La tercera es la que más rinde:

```
  validation {
    condition     = var.escalado.maximo > var.escalado.minimo
    error_message = "El máximo debe ser mayor que el mínimo."
  }
```

Eso es imposible de expresar con variables sueltas, y es el tipo de error que sin ello se descubre aplicando.

Y los valores locales tienen un uso que justifica su existencia por sí solo: centralizar las convenciones.

```
locals {
  prefijo = "cls-${var.sistema}-${var.entorno}-${var.region_corta}"

  etiquetas = {
    sistema          = var.sistema
    entorno          = var.entorno
    equipo           = var.equipo
    gestionado-por   = "terraform"
    repositorio      = var.repositorio
    revision         = var.revision
  }
}

resource "aws_s3_bucket" "facturas" {
  bucket = "${local.prefijo}-facturas"
  tags   = local.etiquetas
}
```

Y con la mayoría de proveedores hay algo mejor: etiquetas por defecto en el proveedor, que se aplican a todo sin repetirlas:

```
provider "aws" {
  default_tags { tags = local.etiquetas }
}
```

Eso cierra de un golpe la obligación de atribución de costo que las clases 025, 037 y 049 pedían, y elimina el recurso que se etiqueta mal por olvido. Con una advertencia: si un recurso define sus propias etiquetas, en algunos proveedores sustituyen a las por defecto en vez de combinarse, así que conviene comprobarlo una vez.

Y un uso de los locales que conviene evitar: la lógica compleja. Un local con tres funciones anidadas y un condicional es un cálculo que nadie va a poder leer en seis meses. Si hace falta, va con un nombre que lo explique y un comentario que diga por qué.

3. Salidas y fuentes de datos: qué exponer y qué leer

Las salidas tienen dos usos distintos que conviene separar:

```
salidas de un MÓDULO      el contrato con quien lo usa (clase 088)
salidas del módulo RAÍZ   información para personas y para otros sistemas
```

Y las del raíz tienen la propiedad que la clase 087 señaló: quedan escritas en el estado. Así que exponer un secreto como salida es publicarlo donde tenga acceso quien pueda leer el estado, exactamente igual que el historial de despliegues de la clase 047. Cuarta aparición de la ley del sistema de solo añadir.

```
output "endpoint" {
  value       = aws_db_instance.pedidos.endpoint
  description = "Punto de conexión de la base de datos."
}

output "password" {                       # ← no
  value     = random_password.bd.result
  sensitive = true
}
```

Marcarla como sensible oculta el valor en la salida por pantalla y no lo cifra en el estado. La forma correcta es que el secreto vaya al gestor de la clase 092 y que la salida sea, como mucho, su referencia.

Y una recomendación práctica: las salidas del raíz que otros sistemas consumen conviene publicarlas también en un almacén de parámetros, por lo que la clase 087 explicó — leer el estado ajeno concede acceso a sus secretos, y leer un parámetro publicado no.

Las fuentes de datos tienen una regla que evita casi todos sus problemas:

```
lo que gestiona OTRO       fuente de datos: correcta
lo que gestiono YO         una referencia directa: siempre mejor
```

El segundo caso aparece con más frecuencia de la esperada:

```
# mal: leer con una consulta algo que este mismo código crea
resource "aws_vpc" "principal" { cidr_block = "10.20.0.0/16" }

data "aws_subnet" "app" {
  vpc_id = aws_vpc.principal.id
  filter { name = "tag:Name" values = ["snet-app"] }
}

# bien: referenciar el recurso
resource "aws_subnet" "app" { /* … */ }
# … y usar aws_subnet.app.id
```

La versión con consulta funciona y tiene dos defectos: añade una dependencia que el grafo no puede resolver bien (clase 086) y falla en la primera aplicación, cuando la subred todavía no existe.

Y tres precauciones con las fuentes de datos que gestionan otros:

```
un filtro que devuelve varios resultados falla la planificación entera
un filtro que no devuelve nada, también
y lo que consultan puede cambiar sin avisar, porque es de otro equipo
```

De ahí que el contrato entre equipos convenga que sea explícito, como la clase 087 propuso: un nombre acordado o un parámetro publicado, con un dueño, en vez de una consulta por etiqueta que alguien puede editar sin saber que rompe algo.

Y una fuente de datos que conviene conocer porque resuelve un caso frecuente: la que consulta la identidad con la que se está ejecutando.

```
data "aws_caller_identity" "actual" {}
```

```
check "cuenta_correcta" {
  assert {
    condition     = data.aws_caller_identity.actual.account_id == var.cuenta_esperada
    error_message = "Se está aplicando sobre la cuenta equivocada."
  }
}
```

Esa comprobación es la versión definitiva de la protección del primer apartado: impide aplicar sobre la cuenta equivocada, con independencia de qué fichero de valores se haya cargado.

4. La diferencia entre entornos cabe en un fichero

Aquí está la regla que las clases 081 y 085 midieron sin enunciarla. Al unificar cuatro copias de manifiestos, la clase 081 encontró doce diferencias reales y treinta y una accidentales. La clase 085 encontró la misma proporción en las plantillas.

La regla que lo evita:

```
toda la diferencia entre dos entornos debe caber en un fichero de valores
que alguien pueda leer entero de una vez
```

Si no cabe, hay dos posibilidades y ninguna buena: o hay diferencias que nadie decidió, o hay diferencias que deberían ser una decisión de arquitectura y se están expresando como configuración.

```
# entornos/pro.tfvars — todo lo que distingue producción
entorno       = "pro"
cuenta_esperada = "418293047512"
region        = "eu-west-1"

escalado      = { minimo = 6, maximo = 40 }
tamano_bd     = "db.r6g.xlarge"
alta_disponibilidad = true
retencion_copias_dias = 35
dominio       = "tienda.example"
```

Ocho valores. Y lo que no debe aparecer ahí:

```
condicionales por entorno dentro del código
  count = var.entorno == "pro" ? 1 : 0
  → un recurso que solo existe en producción es un recurso que
    NUNCA se prueba antes de llegar allí

módulos distintos por entorno
  → entonces no se está promoviendo el mismo código

valores derivados del nombre del entorno dentro de la lógica
  → hace imposible crear un entorno nuevo sin tocar el código
```

El primero es el más común y el más caro. Un recurso condicionado al entorno significa que la ruta de código de producción se ejecuta por primera vez en producción. La alternativa es que el recurso exista en todos los entornos con un tamaño distinto, que es lo que expresa el fichero de valores.

Y hay una excepción legítima que conviene reconocer: algo que no puede existir en un entorno pequeño por su coste —una configuración multirregional, un servicio con tarifa mínima alta—. Ahí, la decisión se documenta y se acepta el riesgo:

```
riesgo declarado: la configuración multirregional solo existe en producción,
  así que su primera ejecución real es en producción.
  Mitigación: ensayo trimestral en un entorno efímero creado para ello.
```

Esa mitigación es la que convierte una excepción en una decisión, y es la misma disciplina de riesgos residuales que este programa pide desde la clase 036.

Y la estructura que sostiene todo lo anterior:

```
modulos/               módulos versionados (clase 088)
infra/
  red/
    main.tf            el MISMO código para los cuatro entornos
    variables.tf
    entornos/
      dev.tfvars
      pre.tfvars
```

```
      pro.tfvars
  datos/
  tienda/
```

Y la aplicación siempre explícita, sin depender de ninguna carga automática:

```
$ terraform -chdir=infra/red init -backend-config=entornos/pro.backend
$ terraform -chdir=infra/red plan -var-file=entornos/pro.tfvars -out=tfplan
$ terraform -chdir=infra/red apply tfplan
```

5. Comprobar que los entornos no divergen

La erosión entre entornos no se evita con disciplina: se evita con una comprobación, porque ocurre despacio y por buenas razones.

La comprobación es comparar los planes:

```
$ for e in dev pre pro; do
    terraform -chdir=infra/tienda plan -var-file=entornos/$e.tfvars \
      -out=/tmp/$e.plan >/dev/null
    terraform -chdir=infra/tienda show -json /tmp/$e.plan \
      | jq -S '[.planned_values.root_module | .. | .type? // empty] | sort' > /tmp/$e.tipos
  done
$ diff /tmp/pre.tipos /tmp/pro.tipos
```

Si los tipos de recurso no coinciden entre preproducción y producción, hay algo que solo existe en uno de los dos. Y esa lista debería ser corta y conocida.

Y una comprobación más fina que detecta la erosión de valores:

```
$ diff <(sort entornos/pre.tfvars) <(sort entornos/pro.tfvars) | grep -c '^[<>]'
```

Ese número es el que hay que vigilar. Si crece con el tiempo sin que nadie tome decisiones, es erosión.

Y la lista de comprobación de la clase:

- ningún fichero con nombre de carga automática en el repositorio
- toda aplicación con fichero de valores explícito
- comprobación de que la cuenta o proyecto es el esperado
- variables estructuradas con campos opcionales, no decenas sueltas
- validación entre campos donde tenga sentido
- convenciones de nombre y etiquetas en un solo sitio, o por defecto en el proveedor
- ningún secreto como salida del módulo raíz
- fuentes de datos solo para lo que gestiona otro, con filtro exacto
- ningún condicional por entorno en el código
- la diferencia entre entornos cabe en un fichero legible de una vez
- comparación periódica de los planes entre entornos

Once puntos, y el noveno es el que más discusión genera. Merece un cierre explícito, porque es la tesis de la clase:

> Un recurso que solo existe en producción es un recurso que nunca se ha probado. Si de verdad no puede existir en otros entornos, eso es un riesgo declarado con su propia mitigación, no una decisión de configuración.

Ejemplo trabajado

CloudShop promueve infraestructura entre cuatro entornos y cada promoción trae sorpresas. Los cuatro hallazgos son de dónde salen los valores, y el último obligó a cambiar cómo se prueban los cambios.

Hallazgo 1 — se aplicó preproducción con los valores de producción.

El plan proponía subir el tamaño de la base de datos de preproducción de db.t4g.medium a db.r6g.xlarge. Alguien lo revisó, no le pareció raro y lo aplicó.

```
$ ls infra/datos/
main.tf  variables.tf  outputs.tf  terraform.tfvars  entornos/
$ head -2 infra/datos/terraform.tfvars
entorno   = "pro"
tamano_bd = "db.r6g.xlarge"
```

Un fichero copiado tres semanas antes para consultar un valor, con nombre de carga automática. Se aplicaba en todas las ejecuciones de esa carpeta, y los valores explícitos de preproducción llegaban antes en la precedencia, así que perdían.

```
ficheros de carga automática en el repositorio    2               0
comprobación de cuenta o proyecto              no había     en las 4 carpetas
comprobación de coincidencia entorno/estado    no había         activa
costo del error                            412 USD en 3 semanas    —
```

La comprobación de identidad es la que de verdad lo cierra: con ella, un fichero equivocado hace fallar la planificación en vez de aplicar en el sitio equivocado.

Hallazgo 2 — 31 recursos que solo existían en producción.

La comparación de planes entre entornos, hecha por primera vez:

```
tipos de recurso en preproducción      41
tipos de recurso en producción         52
diferencia                             11 tipos, 31 recursos
```

Y el origen de los once:

```
condicionados a var.entorno == "pro"      7 tipos   decisión consciente… en 2024
añadidos a mano en producción             3 tipos   nunca llegaron al código
módulo distinto por entorno               1 tipo    nadie recordaba por qué
```

Los siete condicionales incluían la replicación entre regiones, las alertas de guardia y el registro de auditoría. Ninguno se había ejecutado nunca fuera de producción.

```
condicionales por entorno en el código          7               1
recursos solo en producción                    31               4
el recurso restante                             —      configuración multirregional,
                                                       con riesgo declarado y ensayo
                                                       trimestral en entorno efímero
tipos de recurso en preproducción              41              51
costo mensual de preproducción             340 USD         520 USD
```

Ciento ochenta dólares más al mes para que preproducción se parezca a producción. La cifra se comparó con el coste de los dos últimos incidentes atribuibles a esa diferencia —una configuración de alertas que nunca funcionó y una réplica mal configurada— y la decisión fue inmediata.

Hallazgo 3 — el 14 % de los recursos sin etiquetar.

```
$ aws resourcegroupstaggingapi get-resources --query \
  'ResourceTagMappingList[?!(Tags[?Key==`equipo`])].ResourceARN' --output text | wc -l
89
```

Ochenta y nueve recursos sin la etiqueta de atribución de costo que las clases 025 y 049 exigían, casi todos creados por módulos que no la propagaban.

```
mecanismo de etiquetado          repetido en cada recurso   por defecto
                                                            en el proveedor
recursos sin etiqueta de equipo              89                 0
costo no atribuible                      2.180 USD/mes          0
recursos que sustituyen las etiquetas
  por defecto en vez de combinarlas          3, corregidos      —
```

La última fila fue la sorpresa: tres recursos definían sus propias etiquetas y en ese proveedor eso sustituye las por defecto en vez de combinarlas. Se detectó comprobando la lista después del cambio, no antes.

Hallazgo 4 — una contraseña en las salidas durante catorce meses.

```
$ terraform output -json | jq -r 'keys[]'
bd_endpoint
bd_password
$ terraform state pull | jq -r '.outputs.bd_password.value' | head -c 8
Pr0d-202
```

Marcada como sensible, oculta en la salida por pantalla, en claro en el estado — con las doce identidades de lectura que la clase 087 había encontrado.

```
secretos como salida del raíz                  1               0
destino de la contraseña            generada y expuesta   gestor de secretos,
                                                           referencia como salida
credencial rotada                              —          el mismo día
comprobación en la canalización             ninguna     falla si una salida
                                                         coincide con patrones
                                                         de credencial
```

Quinta aparición de la ley del sistema de solo añadir de la clase 072, y la tercera con un mecanismo de infraestructura como código.

Resumen:

```
ficheros de carga automática                      2            0
aplicaciones sobre el entorno equivocado          1            0
condicionales por entorno                         7            1
recursos solo en producción                      31            4
recursos sin etiqueta de atribución              89            0
costo no atribuible                       2.180 USD/mes        0
secretos en salidas                               1            0
comparación de planes entre entornos          nunca          semanal
```

La lección que esta clase traslada al resto de la parte 07: los cuatro hallazgos existían desde hacía meses y ninguno producía un error. El que más caro salió no fue el de la aplicación equivocada sino el segundo: treinta y un recursos cuya primera ejecución real ocurría en producción, porque estaban condicionados al entorno. La comprobación que lo destapó —comparar los tipos de recurso de dos planes— es una línea, y la decisión que salió de ella costó 180 dólares al mes.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/089-variables-outputs-locals-y-data-sources/lab.py
```

El laboratorio selecciona el motor de práctica iac 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 interfaz-infraestructura 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 plan reproducible sin secretos ni cambios inesperados. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye interfaz-infraestructura 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 aplica con los valores de otro entorno y el plan parece razonable	Un fichero con nombre de carga automática está en el directorio y se lee sin mencionarlo	Prohíbe esos nombres en el repositorio, pasa siempre el fichero explícitamente y añade una comprobación de la cuenta o proyecto esperado.
Una configuración falla la primera vez que se usa, en producción	El recurso estaba condicionado al entorno, así que esa ruta nunca se había ejecutado	Que exista en todos los entornos con tamaño distinto; si de verdad no puede, decláralo como riesgo con un ensayo periódico.
Un módulo tiene decenas de variables sueltas relacionadas entre sí	No se usan tipos estructurados con campos opcionales	Agrupa en un objeto con valores por defecto y valida la coherencia entre campos.
Parte de los recursos no lleva las etiquetas obligatorias	El etiquetado se repite en cada recurso y alguno se olvida	Usa etiquetas por defecto del proveedor, y comprueba después que ningún recurso las sustituye en vez de combinarlas.
Un secreto es legible por quien tenga acceso al estado	Se expuso como salida del módulo raíz; marcarla como sensible no la cifra	Que el secreto viva en el gestor y la salida sea, como mucho, su referencia; rota lo ya expuesto.
Los entornos divergen despacio sin que nadie lo decida	No hay ninguna comprobación que compare lo que existe en cada uno	Compara periódicamente los tipos de recurso de los planes y el número de diferencias entre ficheros de valores.

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. Ordena las seis fuentes de valores por precedencia y di cuáles se cargan sin mencionarlas.
2. ¿Qué comprobación impide aplicar sobre la cuenta equivocada, con independencia del fichero de valores?
3. ¿Qué tres ventajas tiene una variable estructurada con campos opcionales frente a variables sueltas?
4. ¿Cuándo es correcta una fuente de datos y cuándo esconde un acoplamiento?
5. ¿Por qué un recurso condicionado al entorno es un problema, y cuál es la excepción legítima?

Referencias

- HashiCorp (2025). Input variables and variable definition precedence — orden de las fuentes y carga automática. <https://developer.hashicorp.com/terraform/language/values/variables>
- HashiCorp (2025). Optional object type attributes — campos opcionales con valores por defecto. <https://developer.hashicorp.com/terraform/language/expressions/type-constraints>
- HashiCorp (2025). Checks and custom conditions — comprobaciones que no bloquean y aserciones. <https://developer.hashicorp.com/terraform/language/checks>
- HashiCorp (2025). Output values and sensitive data — salidas, sensibilidad y su presencia en el estado. <https://developer.hashicorp.com/terraform/language/values/outputs>
- HashiCorp (2025). Default tags in the AWS provider — etiquetado uniforme y su interacción con etiquetas locales. <https://registry.terraform.io/providers/hashicorp/aws/latest/docs#defaulttags>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 07 en PDF · Manual integral

Evaluación — 089 Variables, outputs, locals y data sources

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 interfaz-infraestructura 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.

090 — Plan, apply, drift, import y refactor con moved

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: iac · Estado: EXECUTABLECORE

Propósito

Dominar el ciclo de ejecución —refrescar, planificar, aplicar— y convertir la detección de desviación en una operación programada en lugar de en un descubrimiento accidental. Es la respuesta parcial a la pregunta que dejó la clase 084: sin un bucle continuo, una planificación periódica es lo más parecido a la reconciliación que se puede tener, y sirve para casi todo salvo para corregir sola. La clase cubre además las operaciones quirúrgicas y el procedimiento para refactorizar sin destruir nada.

Resultados de aprendizaje

Al finalizar podrás:

1. Separar las tres fases de una ejecución y saber qué hace cada una con el estado.
2. Programar la detección de desviación y clasificar lo que encuentre.
3. Leer un plan localizando primero lo que destruye o reemplaza.
4. Usar las operaciones quirúrgicas sabiendo qué riesgo introduce cada una.
5. Refactorizar con la comprobación que demuestra que el cambio es seguro.

Conceptos centrales

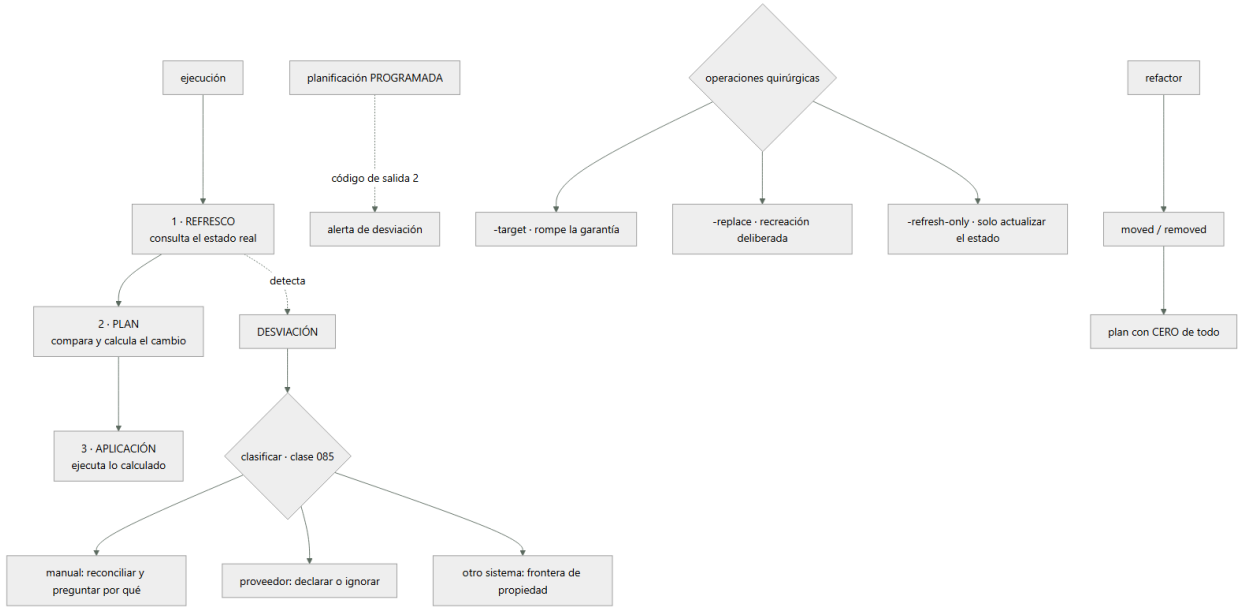
Concepto	Comprensión verificable
refresco	Consulta del estado real de cada recurso antes de comparar. Es lo que hace lenta una planificación grande y lo que detecta la desviación.
plan guardado	Fichero con el cambio calculado. Aplicarlo garantiza que se ejecuta lo revisado; sin él, la aplicación vuelve a planificar (clase 059).
detección programada	Planificación periódica cuyo resultado se vigila. Convierte la desviación en una señal en vez de en un descubrimiento durante un incidente.
operación acotada	Aplicar solo parte del grafo. Es una herramienta de emergencia: rompe la garantía de que el estado corresponde con lo declarado.
declaración de movimiento	Renombrar o mover un recurso sin destruirlo. La comprobación de que está bien es un plan con cero de todo.
declaración de retirada	Dejar de gestionar un recurso conservándolo. Es la forma declarativa y revisable de entregar algo a otro estado o a otro equipo.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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. Tres fases, y qué hace cada una

Una ejecución tiene tres fases y confundirlas lleva a diagnósticos equivocados:

```
1. REFRESCO    consulta el estado REAL de cada recurso al proveedor
               actualiza en memoria lo que ha cambiado
2. PLAN        compara lo declarado con lo refrescado y calcula el cambio
3. APLICACIÓN  ejecuta ese cambio y escribe el estado
```

De ahí salen tres hechos operativos.

El refresco es lo que tarda. Una planificación de mil recursos hace mil llamadas a la API, y ahí está casi todo el tiempo. Por eso repartir el estado (clase 087) acelera tanto: se refrescan menos recursos.

Y se puede saltar, con una advertencia grande:

```
$ terraform plan -refresh=false
```

Eso planifica contra lo que el estado cree que hay, sin comprobarlo. Es útil para una iteración rápida durante el desarrollo y peligroso para aplicar: si algo cambió por fuera, el plan no lo verá y la aplicación puede pisarlo.

El refresco es lo que detecta la desviación. Y desde hace varias versiones ya no escribe el estado por su cuenta durante una planificación: la desviación aparece en el plan, como información:

```
Note: Objects have changed outside of Terraform

  # aws_security_group.app has been changed
  ~ ingress { + cidr_blocks = ["0.0.0.0/0"] }
```

Esa sección es la más importante de un plan y la que menos se lee. Dice qué ha cambiado alguien por fuera, y en el ejemplo dice que alguien abrió un grupo de seguridad a internet.

Y para actualizar el estado sin cambiar nada de la infraestructura, existe una modalidad propia:

```
$ terraform plan  -refresh-only
$ terraform apply -refresh-only
```

Sirve para el caso legítimo de la clase 085: aceptar un cambio del proveedor que no se quiere revertir. Actualiza el estado para que refleje la realidad y deja de proponerlo en cada plan.

La aplicación debe ejecutar un plan guardado. Es la regla de la clase 059 y no ha cambiado:

```
$ terraform plan -out=tfplan -lock-timeout=10m
$ terraform show -no-color tfplan > plan.txt     # esto se revisa
$ terraform apply tfplan                          # esto ejecuta lo revisado
```

Y una precisión sobre el plan guardado que conviene conocer: contiene los valores sensibles de lo que va a crear. Si se publica como artefacto de la canalización, hereda el problema del estado de la clase 087. La versión legible que se publica en la revisión no debe ser el fichero binario sino su representación, con los valores sensibles ya ocultos.

2. La desviación como señal programada

La clase 084 dejó la pregunta: sin un bucle continuo, nada garantiza que dentro de un mes la realidad se parezca al repositorio. La respuesta parcial es convertir la planificación en una comprobación periódica.

```
$ terraform plan -detailed-exitcode -lock-timeout=5m
# 0 = sin cambios
# 2 = hay cambios  ← esto es la señal
# 1 = error
```

Ejecutado cada noche sobre cada estado, ese código de salida es la métrica que faltaba:

```
código 0   la realidad coincide con lo declarado
código 2   hay desviación, o hay cambios sin aplicar en el repositorio
código 1   la ejecución falló: credenciales, bloqueo, un recurso que ya no existe
```

Y la segunda línea tiene dos causas que hay que distinguir, porque una es normal y la otra no:

```
cambios sin aplicar   alguien fusionó código y no se ha desplegado todavía
                      → normal si la canalización aplica al fusionar; sospechoso si no
desviación real       la infraestructura cambió por fuera
                      → la sección "objetos han cambiado fuera de Terraform"
```

La distinción se automatiza mirando el plan en formato estructurado:

```
$ terraform show -json tfplan | jq -r '
    if (.resource_drift // []) | length > 0
    then "DESVIACIÓN: " + ([.resource_drift[].address] | join(", "))
    else "sin desviación" end'
```

Y lo que se hace con lo que encuentre es la clasificación de la clase 085, que aquí se convierte en un procedimiento:

```
1. ¿es un campo que el proveedor rellena?
   → declararlo, o aceptarlo con una aplicación de solo refresco
2. ¿lo gestiona otro sistema?
   → declararlo como ignorado, con el dueño anotado (clase 086)
3. ¿lo cambió una persona?
   → averiguar POR QUÉ antes de revertir
     si el cambio era necesario, va al repositorio
     si no, se revierte y se registra
```

El paso 3 es el que la clase 085 midió: dos de cada tres cambios manuales resultaron ser necesarios.

Y lo que esta detección no hace, para no exagerar su alcance:

```
no corrige nada: solo avisa
no detecta lo que el proveedor no expone en su API
no detecta recursos que existen y no están en ningún estado (clase 087)
y si el propio proceso deja de ejecutarse, no avisa nadie — ley 13
```

La última exige lo que la ley 13 pide siempre: una señal de que la comprobación se ejecutó. Una alerta sobre desviación que lleva tres semanas sin ejecutarse es indistinguible de un sistema sin desviación.

3. Leer un plan por donde importa

Un plan grande se lee en un orden concreto, porque lo peligroso está al final si se lee de arriba abajo.

Primero, lo que destruye:

```
$ terraform show -json tfplan | jq -r '.resource_changes[]
  | select(.change.actions | index("delete"))
  | "\(.change.actions | join(","))  \(.address)"'
delete,create  aws_db_instance.pedidos
delete         aws_s3_bucket.temporal
```

Dos líneas y las dos merecen atención distinta: la primera recrea una base de datos y la segunda borra un bucket.

Segundo, por qué se reemplaza. El plan lo dice al final de la línea del campo culpable:

```
~ resource "aws_db_instance" "pedidos" {
    ~ engine_version = "15.4" -> "16.2" # forces replacement
```

Ese comentario es la información más valiosa de un plan. Y la protección de las clases 059 y 086 lo convierte en un error de planificación en vez de en una pérdida:

```
lifecycle { prevent_destroy = true }
```

Tercero, lo que se modifica en el sitio, que normalmente no tiene riesgo. Y cuarto, lo que se crea, cuyo único riesgo es el coste.

Y la comprobación que conviene automatizar en la canalización, que es la de la clase 059 con el criterio afinado:

```
$ BORRADOS=$(terraform show -json tfplan | jq -r '.resource_changes[]
    | select(.change.actions | index("delete")) | .address')
$ if [ -n "$BORRADOS" ]; then
    echo "$BORRADOS" | tee borrados.txt
    echo "::warning::este plan destruye recursos; exige aprobación explícita"
  fi
```

No se trata de prohibir los borrados —a veces son el objetivo— sino de que ninguno pase sin que alguien lo haya leído y aceptado.

Y el ruido, que es el enemigo de todo lo anterior. La clase 085 midió que el 84 % de los cambios propuestos no eran desviación. Las tres causas y su corrección:

```
campos que el proveedor rellena     declararlos explícitamente
valores normalizados por el proveedor  escribirlos en su forma normalizada
recursos gestionados por otro        declararlos como ignorados
```

Y la métrica que dice si la revisión sigue siendo legible:

```
líneas del plan en una ejecución sin cambios de código
  0        correcto
  1-20     revisable
  >100     nadie lo lee, y el día que aparezca algo real pasará desapercibido
```

Esa es exactamente la lección que la clase 047 aprendió con el what-if y la 081 con el diferencial de manifiestos. Tercera aparición: una previsualización ruidosa es una previsualización que no se usa.

4. Operaciones quirúrgicas y su precio

Hay tres operaciones que actúan sobre una parte del sistema, y las tres tienen un coste que conviene conocer antes de usarlas en una urgencia.

Aplicar solo una parte.

```
$ terraform apply -target=aws_instance.app tfplan
```

Aplica ese recurso y sus dependencias, e ignora el resto. Es útil para salir de una situación bloqueada y rompe la garantía central del sistema: después de una aplicación acotada, el estado ya no corresponde con lo declarado, y nadie sabe qué queda pendiente.

```
reglas de uso
  solo en emergencia, nunca en la canalización
  siempre seguido de una aplicación completa que deje el plan vacío
  y con un registro de por qué se hizo
```

La segunda regla es la importante: una aplicación acotada no ha terminado hasta que una completa dé cero cambios.

Forzar la recreación de un recurso.

```
$ terraform plan -replace=aws_instance.app -out=tfplan
```

Marca ese recurso para destruir y crear. Es la sustituta de la antigua marca de contaminado, y su ventaja es que el efecto se ve en el plan antes de ejecutarse. Sirve para el caso de la clase 085 —el recurso existe, está en el estado y está mal— y para reemplazar una máquina que ha degradado sin cambiar nada declarado.

Actualizar el estado sin tocar la infraestructura.

```
$ terraform apply -refresh-only
```

Acepta la realidad. Es lo correcto cuando la desviación es del proveedor y no se quiere revertir, y hay que hacerlo con el mismo cuidado que lo demás: revisando qué se está aceptando.

Y una operación que no es quirúrgica pero se usa como si lo fuera:

```
$ terraform destroy
```

Su radio de impacto es el estado entero, que es la razón por la que la clase 087 insistía en repartirlo. En entornos efímeros es la operación normal; en producción no debería poder ejecutarse desde la canalización en absoluto.

```
protecciones acumuladas
  prevent_destroy en lo que no se puede perder      (086)
  protección del proveedor contra el borrado        (042, 077)
  permisos de la identidad que aplica: sin borrar    (059)
  y aprobación humana explícita para cualquier plan con borrados
```

Las cuatro son complementarias y ninguna sustituye a las demás: la primera falla la planificación, la segunda falla la llamada, la tercera impide que la identidad lo intente y la cuarta pone a una persona en medio.

5. Refactorizar sin destruir

El refactor es donde el estado se cobra su precio, porque la dirección de un recurso es su identidad. Renombrar sin más lo destruye y lo recrea.

El procedimiento seguro tiene cuatro pasos y una comprobación que decide:

```
1. hacer el cambio en el código
2. declarar los movimientos
3. planificar
4. exigir CERO de todo
```

```
# extraer recursos a un módulo
moved {
  from = aws_instance.web
  to   = module.tienda.aws_instance.web
}
moved {
  from = aws_security_group.web
  to   = module.tienda.aws_security_group.web
}

# pasar de índice numérico a clave estable (el arreglo de la clase 059)
moved {
  from = aws_vpc_security_group_ingress_rule.reglas[0]
  to   = aws_vpc_security_group_ingress_rule.reglas["permitir-https"]
}
```

```
$ terraform plan
Plan: 0 to add, 0 to change, 0 to destroy.
```

Esa línea con tres ceros es la comprobación. Cualquier otra cosa significa que la declaración de movimientos está incompleta o que el refactor cambia algo más de lo previsto, y en los dos casos hay que pararse.

Y para dejar de gestionar algo sin destruirlo, la forma declarativa:

```
removed {
  from = aws_s3_bucket.legado
  lifecycle { destroy = false }
}
```

Eso lo saca del estado y lo deja vivo. Es lo correcto al entregar un recurso a otro equipo, al moverlo a otro estado o al retirar de la gestión algo que pasa a ser de un servicio gestionado. Y frente a la orden equivalente tiene la ventaja de siempre: queda en el repositorio, revisable, con el resto del cambio.

Un refactor grande —repartir un estado en cuatro, extraer módulos, renombrar todo— se hace por pasos y cada paso termina con el plan vacío. Intentarlo de una vez produce un plan de doscientas líneas que nadie puede verificar, y ahí es donde se cuela la destrucción de algo.

Y dos avisos sobre los movimientos:

```
se pueden retirar del código una vez aplicados, y conviene dejarlos
  un tiempo: si otro entorno aún no ha aplicado, los necesita
no sirven para mover entre ESTADOS distintos
  → para eso, la orden de mover con estado de destino (clase 087)
```

Y la lista de comprobación de la clase:

- aplicación siempre con plan guardado
- el plan legible se publica en la revisión, sin valores sensibles
- detección de desviación programada, con señal de que se ejecutó
- la sección de cambios fuera de Terraform, leída y clasificada

- lo que destruye el plan, extraído y aprobado explícitamente
- ruido del plan en cero para una ejecución sin cambios de código
- operaciones acotadas solo en emergencia, seguidas de una completa
- destrucción imposible desde la canalización en producción
- refactores con movimientos declarados y plan con cero de todo
- retiradas declaradas, no con órdenes sueltas

Ejemplo trabajado

CloudShop programa la detección de desviación por primera vez. La primera ejecución sobre los cuatro estados devuelve código 2 en los cuatro, y clasificar lo que encuentra ocupa tres días y cambia cuatro cosas.

La primera ejecución.

estado	código	cambios propuestos	de ellos, desviación real
red	2	31	4
datos	2	18	2
plataforma	2	47	1
tienda	2	9	3
		■■■■■	■■■
		105	10

Noventa y cinco de ciento cinco cambios eran ruido — la misma proporción que la clase 085 midió, en otro repositorio y con otro equipo.

La clasificación de los diez reales.

origen	casos	decisión
campos que el proveedor rellena	—	(ya estaban en el ruido)
otro sistema los gestiona	3	declarados como ignorados
cambio manual necesario	5	incorporados al repositorio
cambio manual innecesario	2	revertidos

Y los cinco necesarios merecen detalle, porque son el argumento contra revertir sin preguntar:

1. una regla de salida hacia un proveedor de pago nuevo, añadida hace 5 meses
2. el tamaño de una instancia subido durante un incidente de julio
3. una alarma de presupuesto que finanzas pidió por correo
4. un permiso de lectura para el equipo de auditoría
5. la retención de una copia de seguridad subida por requisito legal

Cinco decisiones legítimas que solo existían en la infraestructura y no en ningún sitio revisable. Revertirlas habría producido cinco incidentes, y uno de ellos —el quinto— habría sido un incumplimiento.

Y los dos innecesarios:

1. un grupo de seguridad abierto a internet "para probar", hacía 3 semanas
2. un registro de auditoría desactivado durante una migración, hacía 2 meses

El primero es el hallazgo de seguridad del ejercicio: tres semanas con un puerto abierto a internet que ninguna alerta había detectado, porque la detección de desviación no existía.

El ruido, y cuánto costó eliminarlo.

cambios propuestos sin cambios de código	105	0
campos del proveedor no declarados	61	declarados
valores normalizados de otra forma	19	escritos ya normalizados
recursos de otros sistemas	15	declarados como ignorados
esfuerzo	—	2,5 días-persona

Dos días y medio para que la señal sea legible. Y la consecuencia inmediata: a partir de ahí, cualquier cambio propuesto en una ejecución nocturna es real, lo que convierte el código de salida en una alerta útil.

Los tres meses siguientes.

ejecuciones nocturnas	368
con desviación	14
de ellas, cambios manuales	9
de ellas, del proveedor	5
tiempo medio hasta detectar un cambio manual	< 24 h

Y el efecto que nadie había previsto: los cambios manuales bajaron de nueve a dos en el tercer mes. No por una norma, sino porque cada uno generaba una notificación con nombre y apellidos al día siguiente, y la conversación que seguía terminaba siempre igual —«¿esto debería estar en el repositorio?»—.

Y un refactor grande, hecho por pasos.

La separación del estado de la clase 087 se completó extrayendo módulos:

```
paso                              movimientos   plan tras el paso
1. extraer el módulo de red            18       0 add, 0 change, 0 destroy
2. extraer el de datos                 11       0, 0, 0
3. pasar índices a claves estables     26       0, 0, 0
4. retirar recursos legados             4       0, 0, 0  (con retirada declarada)
```

Cincuenta y nueve movimientos en cuatro pasos, sin destruir ni recrear nada. Y el paso 3 fue el que más valor tuvo: eliminó de una vez el riesgo de reindexación que la clase 059 había identificado y que seguía presente en veintiséis recursos.

```
recursos identificados por posición            26               0
riesgo de destrucción al quitar un elemento    alto          ninguno
plan tras cada paso del refactor                —          0, 0, 0
```

Resumen:

```
cambios propuestos sin cambios de código        105              0
desviación detectada                           nunca        en menos de 24 h
cambios manuales al mes                          9              2
días con un puerto abierto sin detectar         21              0
recursos identificados por posición             26              0
refactor con destrucción                       n/a              0
```

La lección que esta clase traslada al resto de la parte 07: la detección programada no corrige nada y aun así cambió el comportamiento del equipo, porque convirtió el cambio manual en algo visible con nombre al día siguiente. Y el trabajo que la hizo posible no fue configurar la ejecución nocturna sino los dos días y medio de eliminar ruido: una señal que grita ciento cinco veces al día no es una señal. La corrección de fondo —un bucle que además reconcilie— es la parte 08.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/090-plan-apply-drift-import-y-refactor-con-moved/lab.py
```

El laboratorio selecciona el motor de práctica iac 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 ciclo-cambio-iac 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 plan reproducible sin secretos ni cambios inesperados. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye ciclo-cambio-iac 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 planificación grande tarda varios minutos	El refresco consulta cada recurso al proveedor	Reparte el estado por radio de impacto; para iterar en desarrollo, planifica sin refresco sabiendo que no verás cambios externos.
La desviación se descubre durante un incidente	Nadie planifica salvo cuando va a cambiar algo	Programa la planificación con código de salida detallado y vigila también que la propia comprobación se ejecute.
Nadie lee el plan porque siempre propone decenas de cambios	Campos del proveedor sin declarar y recursos gestionados por otros sistemas	Elimina el ruido hasta que una ejecución sin cambios de código proponga cero; es la condición para que la señal sirva.
Tras una aplicación acotada nadie sabe qué queda pendiente	Aplicar una parte rompe la correspondencia entre estado y declaración	Úsala solo en emergencia y termina siempre con una aplicación completa que deje el plan vacío.
Un refactor propone destruir y recrear recursos	La dirección del recurso es su identidad y ha cambiado	Declara los movimientos y exige un plan con cero de todo; si propone algo, la declaración está incompleta.
Revertir la desviación provoca varios incidentes	Los cambios manuales respondían a decisiones legítimas que no estaban en el repositorio	Clasifica antes de revertir y averigua el motivo de cada uno: la mayoría suele ser necesaria.

■ 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 hace cada una de las tres fases de una ejecución y cuál detecta la desviación?
2. ¿Qué distingue el código de salida 2 causado por desviación del causado por cambios sin aplicar?
3. ¿En qué orden se lee un plan y cuál es la información más valiosa que contiene?
4. ¿Qué garantía rompe una aplicación acotada y cómo se cierra esa brecha?
5. ¿Qué comprobación demuestra que un refactor con movimientos declarados es seguro?

Referencias

- HashiCorp (2025). The core Terraform workflow — refresco, plan y aplicación. <https://developer.hashicorp.com/terraform/intro/core-workflow>
- HashiCorp (2025). Managing resource drift — detección, refresco y aceptación de cambios externos. <https://developer.hashicorp.com/terraform/tutorials/state/resource-drift>
- HashiCorp (2025). Command: plan and machine-readable output — códigos de salida y formato estructurado. <https://developer.hashicorp.com/terraform/cli/commands/plan>
- HashiCorp (2025). Resource targeting — por qué es una operación excepcional. <https://developer.hashicorp.com/terraform/cli/commands/plan#resource-targeting>

• HashiCorp (2025). The removed block — dejar de gestionar un recurso sin destruirlo. <https://developer.hashicorp.com/terraform/language/resources/syntax#removing-resources>

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

Descargar: Parte 07 en PDF · Manual integral

Anterior	Índice	Siguiente
← 089 · Variables, outputs, locals y data sources	Parte 07 · Programa	091 · Validación, lint, pruebas y policy as code →

Evaluación — 090 Plan, apply, drift, import y refactor con moved

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 ciclo-cambio-iac 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.

091 — Validación, lint, pruebas y policy as code

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: testing · Estado: EXECUTABLECORE

Propósito

Comprobar la infraestructura antes de aplicarla, con una distinción que decide la eficacia de todo lo demás: una comprobación sobre el código no conoce los valores; una sobre el plan sí. Por eso una regla escrita contra el plan detecta lo que ninguna revisión del código puede ver —lo que un módulo produce de verdad con las variables de ese entorno— y por eso es donde deben vivir las reglas de la organización. La clase ordena las comprobaciones por coste y por lo que cada una atrapa, y aplica por cuarta vez la secuencia de adopción que este programa ya ha usado en tres partes.

Resultados de aprendizaje

Al finalizar podrás:

1. Ordenar las comprobaciones por coste y por el tipo de error que detecta cada una.
2. Escribir reglas contra el plan y justificar por qué no valen sobre el código.
3. Adoptar una política sin bloquear al equipo, con la secuencia de tres fases.
4. Gestionar excepciones con motivo, responsable y caducidad.
5. Decidir qué merece una prueba que crea y destruye, y qué no.

Conceptos centrales

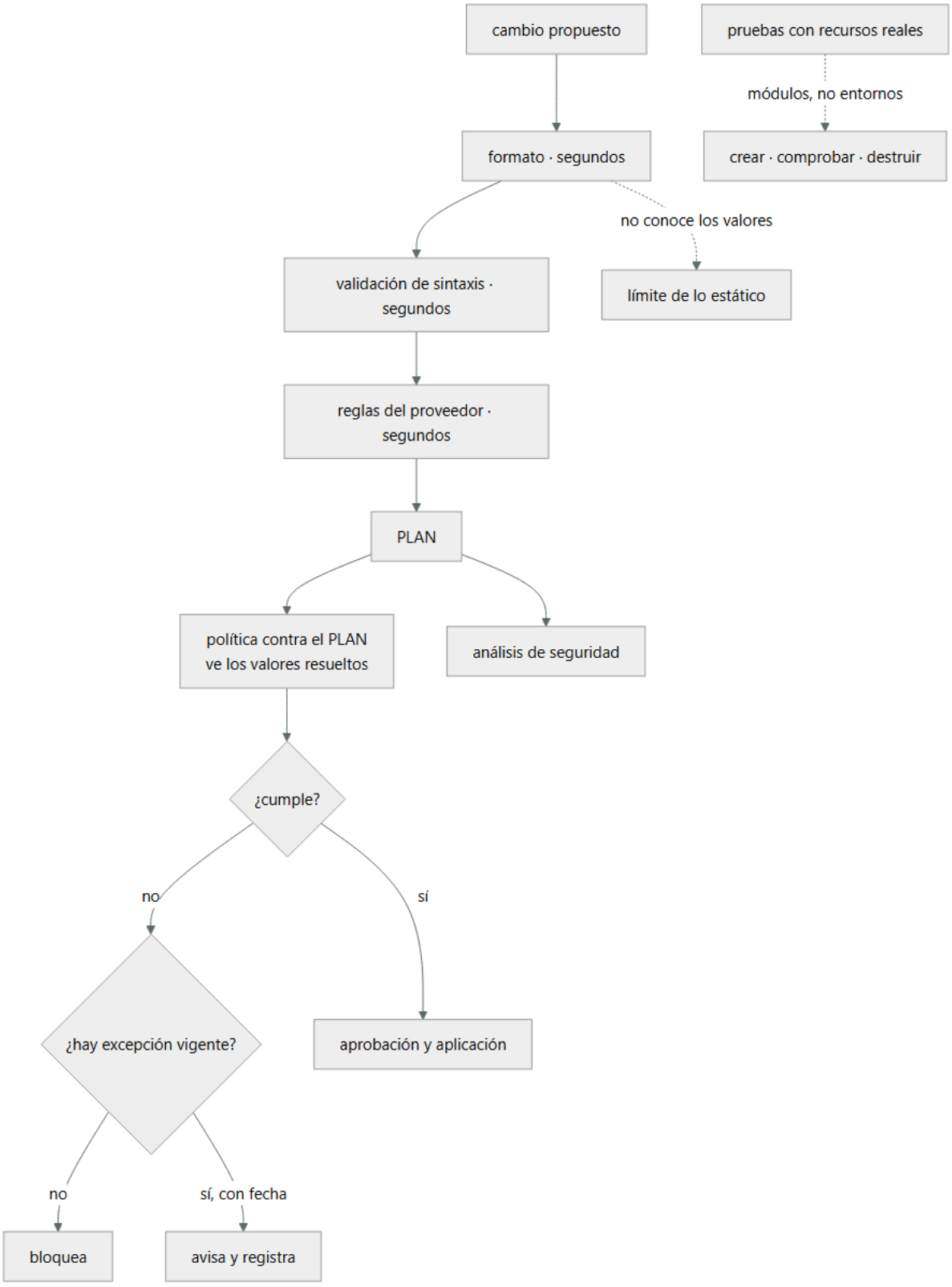
Concepto	Comprensión verificable
comprobación sobre el código	Analiza los ficheros. Es rápida y no conoce los valores: no sabe qué producirá un módulo con las variables de producción.
comprobación sobre el plan	Analiza el cambio calculado, con todos los valores resueltos. Ve lo que de verdad se va a crear, incluido lo que sale de módulos ajenos.
política como código	Reglas de la organización escritas como programa y evaluadas automáticamente. Sustituyen a la revisión humana de lo que se puede comprobar solo.
secuencia de adopción	Avisar, después bloquear lo nuevo, después bloquear siempre. Cuarta aparición en el programa de la misma secuencia: inventariar, corregir, imponer.
excepción con caducidad	Permiso temporal para incumplir, con motivo, responsable y fecha. Sin fecha, la lista de excepciones crece para siempre.
prueba con recursos reales	Ejecución que crea, comprueba y destruye. Es lo único que demuestra que una combinación funciona, y cuesta minutos y dinero.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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 comprobaciones, ordenadas por coste

Cada comprobación atrapa un tipo de error y cuesta un tiempo distinto. Ponerlas en orden importa porque la primera que falla ahorra las siguientes:

```
1. formato            segundos    estilo; evita diferencias inútiles en la revisión
2. validación         segundos    sintaxis, tipos y referencias internas
```

```
3. reglas del proveedor  segundos  valores inválidos, argumentos obsoletos
4. PLAN + política    minutos     lo que de verdad se va a crear
5. pruebas reales     minutos+coste  que la combinación funciona
```

Las tres primeras se ejecutan sin credenciales y sin tocar nada:

```
$ terraform fmt -check -recursive
$ terraform validate
$ tflint --recursive
```

Y su límite es el que define esta clase: no conocen los valores. La validación comprueba que una variable existe y no qué contendrá; las reglas del proveedor detectan un tipo de instancia inexistente y no que el tamaño elegido para producción sea el de desarrollo.

```
resource "aws_db_instance" "pedidos" {
  instance_class      = var.tamano_bd        # ¿cuál es? el código no lo sabe
  publicly_accessible = var.acceso_publico   # ¿true? tampoco
  storage_encrypted   = var.cifrado
}
```

Un análisis estático de ese fragmento no puede decir nada útil. Y con módulos es peor: lo que un módulo crea depende de sus variables, y quien lo consume no ve sus recursos.

De ahí la afirmación central:

> Las reglas de la organización se escriben contra el plan, no contra el código. El plan tiene todos los valores resueltos, incluidos los que salen de módulos que nadie ha leído.

Y una nota sobre la tercera comprobación, que rinde más de lo que parece: detecta argumentos obsoletos antes de que una actualización del proveedor los retire, que es el mismo trabajo preventivo que la métrica de APIs obsoletas de la clase 083.

Y sobre el coste total, un criterio operativo que decide si todo esto se usa:

```
una comprobación de menos de 2 minutos se ejecuta siempre
una de 10 minutos se ejecuta cuando alguien se acuerda
una de 30 minutos se desactiva
```

Es la misma lección que la clase 067 dejó con el escáner de vulnerabilidades desactivado durante ocho meses. La consecuencia práctica: lo rápido en cada cambio, lo caro en un horario o antes de fusionar.

2. Reglas contra el plan

El plan en formato estructurado contiene los cambios con todos sus valores, y sobre eso se pueden escribir las decisiones que este programa ha ido tomando:

```
$ terraform show -json tfplan > plan.json
$ conftest test --policy politicas/ plan.json
```

Y las reglas son legibles:

```
package terraform.cloudshop
import rego.v1

recursos_creados[r] if {
  r := input.resource_changes[_]
  r.change.actions[_] in {"create", "update"}
}

# clase 041: nada de almacenamiento sin cifrar
deny contains msg if {
  r := recursos_creados[_]
  r.type == "aws_s3_bucket_server_side_encryption_configuration"
  not r.change.after.rule
  msg := sprintf("%s: falta configuración de cifrado", [r.address])
}

# clase 054: ninguna base de datos con acceso público
deny contains msg if {
  r := recursos_creados[_]
  r.type == "aws_db_instance"
  r.change.after.publicly_accessible == true
  msg := sprintf("%s: acceso público prohibido (clase 054)", [r.address])
}
```

```
# clases 025, 049: atribución de costo obligatoria
deny contains msg if {
  r := recursos_creados[_]
  etiquetables[r.type]
  not r.change.after.tags.equipo
  msg := sprintf("%s: falta la etiqueta 'equipo'", [r.address])
}

# clase 090: ningún borrado sin aprobación explícita
warn contains msg if {
  r := input.resource_changes[_]
  "delete" in r.change.actions
  msg := sprintf("%s se DESTRUYE: requiere aprobación", [r.address])
}
```

Y tres propiedades que hacen que esto funcione donde la revisión humana no llega:

```
ve el resultado de los módulos       aunque el consumidor no los haya leído
ve los valores del entorno concreto  la misma plantilla puede cumplir en
                                     desarrollo e incumplir en producción
no se cansa                          la comprobación 400 es igual de rigurosa
                                     que la primera
```

La segunda es la que más veces salva: una regla contra el código aprueba una plantilla que en producción recibirá un valor prohibido.

Y sobre el mensaje de error, la misma exigencia que la clase 088 pedía a las validaciones: debe decir qué se esperaba y por qué. Un mensaje que cita la clase o la decisión convierte el bloqueo en algo que se entiende sin preguntar.

Y una regla que conviene tener y casi nadie escribe, porque codifica una ley del programa:

```
# ley 14: lo que se decide al crear no se cambia después
warn contains msg if {
  r := input.resource_changes[_]
  r.change.actions == ["delete", "create"]
  r.type in recursos_con_datos
  msg := sprintf("%s se RECREA y contiene datos: ¿es intencionado?", [r.address])
}
```

3. Adoptar sin bloquear al equipo

Este programa ya ha aplicado esta secuencia tres veces —Azure Policy en la clase 046, políticas de organización en la 049 y perfiles de seguridad de pods en la 080— y aquí es la cuarta:

```
fase 1 · AVISAR        la política se evalúa y solo informa
                       objetivo: medir el tamaño del problema
fase 2 · BLOQUEAR LO NUEVO   solo falla si el recurso se crea o si el
                       incumplimiento es nuevo
                       objetivo: dejar de empeorar
fase 3 · BLOQUEAR SIEMPRE  cuando el inventario ya cumple
```

Empezar por la fase 3 produce lo que la clase 067 midió: la comprobación se desactiva y figura como implantada.

La fase 1 tiene un entregable concreto y es lo que decide el plan de trabajo:

```
$ for e in dev pre pro; do
    terraform plan -var-file=entornos/$e.tfvars -out=/tmp/$e.plan >/dev/null
    terraform show -json /tmp/$e.plan > /tmp/$e.json
    conftest test --policy politicas/ --output json /tmp/$e.json \
      | jq -r --arg e "$e" '.[].failures[]? | "\($e) \(.msg)"'
  done | sort | uniq -c | sort -rn
```

Eso da la lista de incumplimientos por entorno y por regla, ordenada por frecuencia. Y esa lista es la que se reparte y se corrige, no una tarea genérica de «cumplir la política».

La fase 2 exige distinguir lo nuevo de lo existente, y hay dos formas:

```
por acción     bloquear solo si el recurso se CREA; avisar si se modifica
por referencia  comparar con el resultado de la rama principal
               → bloquear si el número de incumplimientos SUBE
```

La segunda es más justa y más fácil de defender: nadie puede empeorar el estado, y quien toque un recurso existente no está obligado a arreglar todo lo que había.

Y las excepciones con la disciplina de las clases 046, 049 y 067:

```
# excepciones.yaml
- regla: "bd-sin-acceso-publico"
  recurso: "aws_db_instance.legado"
  motivo: "Sistema heredado; su cliente no puede usar el proxy. Migración prevista."
  responsable: "equipo-pedidos"
  caduca: "2026-11-30"
```

Y la comprobación que hace que la lista no crezca para siempre:

```
$ yq -r '.[] | select(.caduca < now | strftime("%Y-%m-%d")) | .regla + " " + .recurso' \
    excepciones.yaml | tee caducadas.txt
$ [ ! -s caducadas.txt ] || { echo "excepciones caducadas"; exit 1; }
```

Una excepción caducada rompe la canalización, lo que fuerza la conversación en vez de dejar que se olvide. Es más agresivo de lo que parece y es lo único que funciona: sin ello, la lista de excepciones se convierte en la política real.

4. Qué merece una prueba que crea recursos

Las pruebas con recursos reales cuestan minutos y dinero, así que hay que ser selectivo. El criterio:

```
SÍ merece prueba real
  los MÓDULOS de la organización: sus combinaciones declaradas (clase 088)
  las decisiones que la política no puede comprobar
    (que el servicio arranca, que la conmutación funciona, que la copia restaura)
  los caminos que solo existen en producción, en un entorno efímero (clase 089)

NO merece prueba real
  que el proveedor crea lo que dice que crea
  cada combinación de un módulo con veinte opciones
    → eso indica un problema de diseño, no de pruebas (clase 088)
  los entornos completos en cada cambio: demasiado lento
```

La segunda línea del segundo bloque es importante: probar es caro, así que un módulo que necesita cientos de pruebas está mal diseñado. La corrección es dividir por decisión, como la clase 088 estableció.

Y una estructura de pruebas que cubre lo que importa:

```
# pruebas/valores.tftest.hcl — sin crear nada: valida entradas
run "rechaza_replicas_insuficientes" {
  command   = plan
  variables { escalado = { minimo = 1 } }
  expect_failures = [var.escalado]
}

# pruebas/crea.tftest.hcl — crea, comprueba y destruye
run "crea_conforme" {
  command = apply
  assert {
    condition     = aws_db_instance.este.storage_encrypted
    error_message = "la base debe estar cifrada"
  }
  assert {
    condition     = !aws_db_instance.este.publicly_accessible
    error_message = "la base no debe ser accesible públicamente"
  }
}
```

Y las condiciones que hacen sostenible probar de verdad, que la clase 088 enumeró y conviene repetir porque es donde se descuidan:

```
cuenta o proyecto propio, con presupuesto y alerta
nombres únicos por ejecución
destrucción garantizada incluso si la prueba falla
limpieza periódica de lo que quedó por el camino
```

Y una comprobación que no es una prueba y detecta más que muchas: el coste estimado del cambio.

```
$ infracost breakdown --path tfplan --format json | jq -r '.totalMonthlyCost'
```

Publicar en la revisión cuánto va a costar el cambio convierte una decisión invisible en una explícita. Y con un umbral, en una decisión que necesita aprobación:

```
incremento < 50 USD/mes    informativo
incremento > 50 USD/mes    requiere aprobación de quien tiene el presupuesto
```

Es la aplicación directa del criterio de las clases 025 y 049 —lo que no tiene número se opina— al momento en que todavía se puede decidir.

5. La canalización completa, y qué falla dónde

Reuniendo todo, la canalización de un cambio de infraestructura tiene esta forma:

```
en el pull request
  1. formato y validación                        segundos
  2. reglas del proveedor                        segundos
  3. plan con la identidad de LECTURA (clase 059)
  4. política contra el plan                     bloquea
  5. análisis de seguridad sobre el plan         bloquea lo crítico
  6. estimación de coste                         informa, y bloquea sobre umbral
  7. lista de recursos que se destruyen          exige aprobación
  8. publicar el plan legible en el pull request

al fusionar
  9. aplicar el plan GUARDADO, con la identidad de escritura
 10. verificar: una segunda planificación debe dar cero cambios

programado
 11. detección de desviación (clase 090)
 12. excepciones caducadas
 13. pruebas de los módulos, con creación real
```

El paso 10 es la comprobación de idempotencia de la clase 085 en su sitio definitivo: si tras aplicar sigue habiendo cambios, algo no está declarado.

Y el reparto de identidades del paso 3 y el 9 es el de la clase 059: quien planifica en una rama no puede aplicar. Sin eso, ejecutar la canalización desde una rama arbitraria es dar permiso de escritura a cualquiera que abra un cambio.

Y una lista de qué error atrapa cada paso, que es lo que justifica tenerlos todos:

```
formato            diferencias inútiles en la revisión
validación         una variable que no existe, un tipo incompatible
reglas de proveedor  un argumento retirado, un valor imposible
política           un bucket público, una base sin cifrar, una etiqueta ausente
seguridad          configuraciones inseguras que la política propia no cubre
coste              una instancia diez veces más cara por un cero de más
borrados           una recreación de algo con datos
plan publicado     todo lo demás, que sigue necesitando una persona
```

La última línea merece decirse: ninguna de estas comprobaciones sustituye a leer el plan. Lo que hacen es reducir lo que hay que leer a lo que de verdad requiere criterio.

Y la lista de comprobación de la clase:

- formato, validación y reglas del proveedor en cada cambio
- reglas de la organización escritas contra el PLAN, no contra el código
- mensajes que dicen qué se esperaba y citan la decisión
- adopción por fases: avisar, bloquear lo nuevo, bloquear siempre
- excepciones con motivo, responsable y caducidad, y la caducidad rompe la canalización
- pruebas con recursos reales para los módulos, no para los entornos
- estimación de coste publicada, con umbral de aprobación
- borrados extraídos y aprobados explícitamente
- identidad de lectura para planificar, de escritura solo al fusionar
- segunda planificación tras aplicar, con cero cambios

Ejemplo trabajado

CloudShop tiene un analizador de seguridad instalado y desactivado. La reactivación se hace con la secuencia de tres fases y produce cuatro hallazgos, uno de ellos imposible de ver desde el código.

El punto de partida.

```
$ git log --oneline -S 'continue-on-error' .github/workflows/infra.yml | tail -1
8c1f4a2  "desbloquear la canalizacion"
```

Catorce meses atrás, con 812 hallazgos. Desde entonces figuraba como implantado — la misma historia que la clase 067 encontró con el escáner de imágenes.

Fase 1 — medir.

```
hallazgos totales                              812
  informativos y de estilo                     504
  medios sin corrección clara                  187
  altos y críticos                              97
  de ellos, en recursos que ya no existen       73
  reales y accionables                          24
```

Veinticuatro. De ochocientos doce. Y ese es el motivo por el que se desactivó: la señal estaba enterrada.

```
criterio de bloqueo                  todo lo que sea    alto y crítico,
                                     un hallazgo        sobre recursos del plan
hallazgos que bloquean                  812                 24
tiempo de la comprobación             6 min 40 s           50 s
```

La reducción de tiempo viene de analizar el plan en vez de todos los ficheros del repositorio: solo se evalúa lo que va a cambiar.

Fase 2 — bloquear lo nuevo. Durante seis semanas, la canalización comparaba con la rama principal y fallaba solo si el número subía.

```
semana 1-6
  intentos bloqueados por empeorar               7
  hallazgos corregidos por los equipos          24 → 6
  quejas por bloqueo injusto                     0
```

Cero quejas es el dato: nadie tuvo que arreglar lo que había, solo no empeorarlo.

Fase 3 — bloquear siempre, con las seis excepciones restantes documentadas.

Hallazgo 1 — lo que el código no podía ver.

Al añadir política contra el plan, la primera ejecución encontró algo que catorce meses de análisis del código no habían visto:

```
module.datos.aws_db_instance.informes: acceso público prohibido (clase 054)
```

El módulo tenía una variable accesopublico con valor por defecto false, y el fichero de valores de un entorno la ponía a true desde hacía dos años.

```
en el código del módulo    publicly_accessible = var.acceso_publico
                           → un análisis estático no puede saber su valor
en el plan                 publicly_accessible = true
                           → visible, y bloqueado
```

Es la demostración exacta de la tesis de la clase. La base de datos era de un entorno de integración con datos anonimizados, así que no fue una brecha; y había estado accesible desde internet durante dos años sin que ninguna comprobación lo viera.

Hallazgo 2 — la estimación de coste, y el cero de más.

```
incremento mensual estimado del cambio:  +4.180 USD
```

Un cambio rutinario de tamaño de instancia con un cero de más: db.r6g.16xlarge en vez de db.r6g.xlarge. El plan lo mostraba, nadie lo habría leído con esa atención, y la estimación lo puso en el título de la revisión.

```
estimación de coste en la revisión         no había         publicada
umbral de aprobación                       no había      +50 USD/mes
cambios detenidos por coste el primer trimestre  —              3
el mayor de ellos                                —          4.180 USD/mes
```

Hallazgo 3 — las excepciones que se convirtieron en la política.

Al revisar la lista heredada:

```
excepciones registradas                        31
  con motivo escrito                            8
  con responsable                               3
  con fecha de caducidad                        0
  cuyo recurso ya no existe                    14
```

Ninguna caducaba. Diecisiete se retiraron por obsoletas, ocho se corrigieron y seis quedaron con motivo, responsable y fecha.

```
excepciones                                    31               6
```

```
con los tres campos obligatorios                0                6
la caducidad rompe la canalización              no               sí
```

Hallazgo 4 — las pruebas de módulos encontraron una regresión.

Al añadir las pruebas de la clase 088 a la canalización de los módulos:

```
primera ejecución tras un cambio rutinario
  falla: "la base debe estar cifrada"
```

Un cambio que reorganizaba variables había cambiado el valor por defecto del cifrado de true a false, sin que nadie lo notara: la interfaz no cambiaba y el código compilaba.

```
pruebas de módulos en la canalización      no había        6 módulos
regresiones detectadas el primer trimestre     —              2
de ellas, que habrían llegado a producción     —              2
duración de la suite                           —          14 min, programada
```

Las dos regresiones son el argumento para el coste: catorce minutos y unos dólares al mes evitaron dos cambios silenciosos en valores de seguridad por defecto.

Resumen:

```
analizador de seguridad                    desactivado       activo
hallazgos que bloquean                        812             6
reglas de organización contra el plan          0             19
recursos con acceso público                    1              0
estimación de coste en la revisión            no             sí
excepciones con caducidad                   0 de 31        6 de 6
pruebas de módulos                             0              6
tiempo total de comprobación en un cambio   6 min 40 s     1 min 50 s
```

La lección que esta clase traslada al resto de la parte 07: el analizador llevaba catorce meses desactivado por la misma razón que el escáner de la clase 067 y que las alertas de la clase 057 — una señal con ochocientos elementos no es una señal. Y el hallazgo que justifica la clase entera es el primero: una base de datos accesible desde internet durante dos años, invisible para cualquier análisis del código y visible en el primer plan que se evaluó. Las reglas de la organización van contra el plan.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/091-validacion-lint-pruebas-y-policy-as-code/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 pipeline-iac 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 pipeline-iac 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 analizador de seguridad acaba desactivado	Bloquea por cientos de hallazgos, la mayoría informativos o de recursos que ya no existen	Evalúa sobre el plan, bloquea solo lo alto y crítico, y adopta por fases: avisar, no empeorar, bloquear siempre.
Una comprobación del código aprueba algo que en producción incumple	El análisis estático no conoce los valores que las variables tendrán	Escribe las reglas de la organización contra el plan, donde todos los valores están resueltos.
La lista de excepciones crece hasta convertirse en la política real	Ninguna tiene fecha de caducidad	Exige motivo, responsable y fecha, y haz que una excepción caducada rompa la canalización.
Un cambio multiplica el coste sin que nadie lo note	El plan lo muestra y nadie lo lee con esa atención	Publica la estimación de coste en la revisión y pon un umbral que exija aprobación.
Un cambio de un módulo altera un valor de seguridad por defecto sin romper nada	La interfaz no cambia, así que ninguna comprobación estática lo ve	Pruebas con recursos reales que verifiquen las decisiones del módulo, no su sintaxis.
Las comprobaciones tardan tanto que se saltan	Todo se ejecuta en cada cambio, incluidas las pruebas caras	Lo rápido en cada cambio, lo caro programado o antes de fusionar; por encima de dos minutos, la gente busca cómo evitarlo.

■ 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 límite tienen las comprobaciones sobre el código, y por qué las reglas de la organización van contra el plan?
2. Describe la secuencia de tres fases de adopción y qué produce empezar por la última.
3. ¿Qué tres campos debe tener una excepción y qué mecanismo impide que la lista crezca para siempre?
4. ¿Qué merece una prueba que crea recursos reales y qué indica que un módulo necesita demasiadas?
5. ¿Qué error atrapa cada paso de la canalización, y por qué ninguno sustituye a leer el plan?

Referencias

- Open Policy Agent (2025). Terraform plan testing with Conftest — reglas contra el plan en formato estructurado. <https://www.openpolicyagent.org/docs/latest/terraform/>
- HashiCorp (2025). Terraform JSON output format — estructura del plan para automatización. <https://developer.hashicorp.com/terraform/internals/json-format>
- TFLint (2025). Rules and provider plugins — errores de proveedor y argumentos obsoletos. <https://github.com/terraform-linters/tflint>

- HashiCorp (2025). Terraform test framework — pruebas con creación real y fallos esperados. <https://developer.hashicorp.com/terraform/language/tests>
- Infracost (2025). Cost estimation in pull requests — estimación sobre el plan y umbrales. <https://www.infracost.io/docs/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 07 en PDF · Manual integral

Anterior	Índice	Siguiente
← 090 · Plan, apply, drift, import y refactor con moved	Parte 07 · Programa	092 · Secretos y datos sensibles en IaC →

Evaluación — 091 Validación, lint, pruebas y policy as code

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-iac 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.

092 — Secretos y datos sensibles en IaC

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: security · Estado: EXECUTABLECORE

Propósito

Manejar secretos en infraestructura como código, que es donde la ley 11 del programa —lo que un sistema de solo añadir «borra» sigue siendo recuperable— se cobra sus peores casos. Un secreto que pasa por una plantilla deja rastro en seis sitios, y solo uno de ellos es el repositorio. La clase ordena las tres estrategias posibles, empezando por la única que resuelve el problema de raíz, y cierra con la auditoría que reúne los cinco escapes que este programa ha ido encontrando desde la clase 047.

Resultados de aprendizaje

Al finalizar podrás:

1. Enumerar los seis lugares donde un secreto deja rastro al pasar por una plantilla.
2. Elegir entre no tener el secreto, referenciarlo o materializarlo, en ese orden.
3. Explicar qué protege y qué no protege marcar un valor como sensible.
4. Rotar sin que la rotación produzca desviación permanente.
5. Auditar los rastros con una comprobación ejecutable en la canalización.

Conceptos centrales

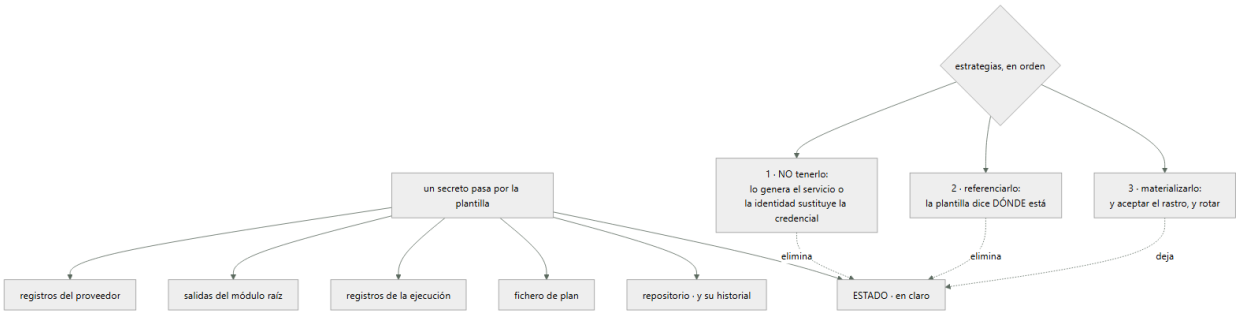
Concepto	Comprensión verificable
rastro de un secreto	Cada sitio donde queda una copia al pasar por la plantilla: repositorio, estado, fichero de plan, registros de la ejecución, salidas y registros del propio proveedor.
no tener el secreto	Estrategia en la que el valor nunca existe para la plantilla: lo genera y lo custodia el servicio, o la autenticación es por identidad y no hay credencial.
referencia en vez de valor	La plantilla declara dónde está el secreto y no su contenido. Quien lo consume lo resuelve al ejecutar (clases 058, 076).
marca de sensible	Oculta el valor en la salida por pantalla. No lo cifra en el estado ni en el fichero de plan, así que no es una protección de almacenamiento.
valor efímero	Argumento que se usa durante la operación y no se guarda en el estado. Es la dirección en la que evolucionan las herramientas y la única solución estructural.
desviación por rotación	Efecto de que la plantilla declare un valor que otro sistema rota. Cada plan propone devolverlo, y cada aplicación deshace la rotación.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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. Seis rastros, y solo uno es el repositorio

Cuando un secreto pasa por una plantilla, queda en más sitios de los que se supone. Conviene la lista completa porque las auditorías suelen revisar el primero y ninguno de los otros cinco:

```
1. el repositorio          y su HISTORIAL: borrarlo del fichero no lo borra
2. el ESTADO               en claro, siempre (clases 059, 087)
3. el fichero de plan      contiene los valores de lo que va a crear
4. los registros de la ejecución  si algo falla, el mensaje puede incluirlo
5. las salidas             si se expone, queda en el estado (clase 089)
6. los registros del proveedor    la llamada a la API queda registrada
```

Los rastros 1 y 2 son los que este programa ya ha pagado. El 1 en las clases 061 y 066 —el token en una capa, el fichero de entorno versionado—; el 2 en las clases 059 y 087 —catorce personas leyendo siete contraseñas—.

El 3 merece atención porque es reciente y se olvida: un plan guardado que se publica como artefacto de la canalización contiene los valores de todo lo que va a crear. Si el artefacto se conserva treinta días y lo puede descargar cualquiera del equipo, el secreto ha viajado a un sistema más, con otra retención y otros permisos.

El 4 aparece en los peores momentos:

```
$ TF_LOG=DEBUG terraform apply     # ← imprime cuerpos de petición
```

Esa variable se activa para depurar un problema y a menudo se queda activada en la canalización. Y con ella, cada credencial que viaja en una llamada a la API queda escrita en los registros del sistema de integración continua, que retiene noventa días.

Y el 6 es el que no se puede evitar: el proveedor registra las llamadas a su API, y algunas incluyen el valor. No hay corrección posible salvo la estrategia 1 de la sección siguiente.

Y la consecuencia general, que ya es la ley 11 del programa en su quinta aparición:

> Un secreto que ha pasado por una plantilla está comprometido en seis sitios. La corrección no es limpiar los seis: es no ponerlo ahí.

Y el orden de actuación cuando ya ha pasado, que este programa ha aplicado cinco veces y no cambia:

```
1. rotar el secreto     estuvo expuesto; el resto es secundario
2. corregir el mecanismo
3. purgar lo que se pueda purgar
```

Invertir el orden —corregir primero y rotar «cuando haya tiempo»— deja la credencial válida en seis sitios recuperables.

2. Tres estrategias, y la primera es la que resuelve

Estrategia 1: que el secreto no exista para la plantilla.

Es la única que elimina los seis rastros, y tiene dos formas.

La primera: el servicio lo genera y lo custodia. Muchos servicios gestionados admiten que su credencial la genere el propio proveedor y la deposite en su gestor de secretos:

```
resource "aws_db_instance" "pedidos" {
  identifier                  = "cls-pedidos"
  manage_master_user_password = true         # el valor NUNCA pasa por aquí
  master_username             = "app"
```

```
  # …
}
```

La plantilla nunca ve la contraseña: la genera el servicio, la guarda en el gestor y la rota según su política. El estado guarda una referencia, no el valor.

La segunda forma es la que este programa lleva siete clases repitiendo: que no haga falta ninguna credencial.

```
la aplicación se autentica con su identidad de carga
  rol de instancia · identidad administrada · cuenta de servicio
  federada con la nube (clases 026, 038, 050, 069, 080, 083)
→ no hay contraseña que crear, que guardar ni que rotar
```

Séptima aparición, y sigue siendo la mejor respuesta a cualquier pregunta sobre secretos: el secreto que no existe no se filtra.

Estrategia 2: referenciar, no materializar.

Cuando el valor tiene que existir pero la plantilla no necesita conocerlo, la plantilla declara dónde está:

```
# se crea el contenedor del secreto, sin valor
resource "aws_secretsmanager_secret" "api_pago" {
  name = "${local.prefijo}/api-pago"
}

# el valor lo pone otro proceso, fuera de la plantilla
# y quien lo consume lo resuelve al ejecutar (clases 058, 076)
resource "aws_ecs_task_definition" "tienda" {
  container_definitions = jsonencode([{
    secrets = [{ name = "API_PAGO", valueFrom = aws_secretsmanager_secret.api_pago.arn }]
  }])
}
```

El estado guarda el identificador del secreto, que no es sensible. Y quien lo necesita lo lee en ejecución con su propia identidad, con la distinción de las clases 058 y 076 sobre cuándo se resuelve.

Estrategia 3: materializar, y aceptar el precio.

Hay casos en los que no queda otra: un servicio que solo acepta el valor en su creación y no admite generación por el proveedor.

```
resource "random_password" "bd" {
  length  = 32
  special = true
}

resource "aws_secretsmanager_secret_version" "bd" {
  secret_id     = aws_secretsmanager_secret.bd.id
  secret_string = random_password.bd.result
}
```

Eso funciona y deja el valor en el estado. Lo que hay que hacer entonces es declararlo como riesgo y compensarlo:

```
el estado en un almacén con acceso mínimo y cifrado (clase 087)
rotación posterior fuera de la plantilla
y el campo declarado como ignorado, para que la rotación no produzca desviación
```

Y la dirección en la que evolucionan las herramientas, que conviene conocer porque cambia el análisis: los valores efímeros y los argumentos que solo se escriben, que se usan durante la operación y no se guardan en el estado. Donde estén disponibles, convierten la estrategia 3 en aceptable de verdad.

3. Lo que la marca de sensible hace y no hace

Es el malentendido más extendido de esta clase:

```
variable "contrasena" {
  type      = string
  sensitive = true
}

output "cadena" {
  value     = local.cadena_conexion
  sensitive = true
}
```

```
SÍ hace
  oculta el valor en la salida por pantalla del plan y de la aplicación
  propaga la marca a los valores derivados de él
  impide mostrarlo con `terraform output` sin pedirlo explícitamente

NO hace
  no lo cifra en el estado
  no lo cifra en el fichero de plan
  no impide que el proveedor lo registre en su API
  no impide que aparezca en un mensaje de error del proveedor
```

La propagación merece un apunte porque tiene un efecto secundario molesto: si un valor sensible entra en una expresión, el resultado entero se marca como sensible, y a veces eso oculta información que hacía falta ver:

```
(sensitive value)
```

Eso en un plan impide revisar el cambio. La salida no es quitar la marca sino no mezclar el valor sensible con lo que hay que poder leer: construir la cadena de conexión donde se consume, no en la plantilla.

Y hay dos protecciones adicionales que sí actúan sobre el almacenamiento:

```
cifrado del estado en reposo        clase 087; protege del acceso al almacén
cifrado a nivel de valor            algunas herramientas y complementos lo ofrecen
                                    y la clave pasa a ser el activo
```

Y sobre los valores cifrados en el repositorio, que es una práctica extendida: funcionan, y hay que decir con precisión qué mueven.

```
el secreto deja de estar en claro en el repositorio
y la CLAVE que lo descifra pasa a ser el activo que hay que
  custodiar, rotar, auditar y controlar
y el valor descifrado sigue acabando en el estado si se materializa
```

No es peor que las alternativas; es una decisión con su propio coste. Y solo tiene sentido si esa clave está de verdad mejor protegida que el secreto que protege — lo que exige que viva en el gestor de la nube y que su uso esté restringido a la identidad de la canalización.

Y la comprobación honesta que decide si sirve, y que conviene ejecutar una vez:

```
$ terraform state pull | grep -c 'BEGIN PRIVATE KEY\|password\|secret_string'
```

Si ese número no es cero, el cifrado del repositorio ha movido el problema y no lo ha resuelto.

4. Rotar sin producir desviación

Un secreto creado por la plantilla y rotado por otro sistema produce el conflicto de propiedad de la clase 085, y es especialmente molesto porque cada aplicación deshace la rotación.

```
la plantilla declara el valor
el gestor lo rota a los 90 días
el siguiente plan propone devolverlo al valor de la plantilla
y si alguien aplica, la rotación se pierde y el servicio deja de autenticarse
```

Las tres salidas, en orden de preferencia:

```
1. que la plantilla no gestione el valor
   crea el contenedor del secreto y nada más (estrategia 2)
   → no hay nada que rotar desde aquí

2. que el servicio lo gestione
   generación y rotación por el proveedor (estrategia 1)

3. declarar el campo como ignorado
   la plantilla pone el valor inicial y no vuelve a mirarlo
```

```
resource "aws_secretsmanager_secret_version" "bd" {
  secret_id     = aws_secretsmanager_secret.bd.id
  secret_string = random_password.bd.result
  lifecycle {
    ignore_changes = [secret_string]     # lo rota el gestor, no la plantilla
  }
}
```

Y la tercera tiene una consecuencia que conviene anticipar: a partir de ahí, el valor del estado es el inicial y no el actual. Cualquiera que lea el estado obtiene una credencial que ya no vale, lo que es una mejora accidental de seguridad y

una fuente de confusión al depurar.

Y el patrón de dos credenciales de las clases 046 y 058 sigue siendo la única forma de rotar sin ventana de fallo, y aquí tiene una particularidad: la plantilla no debe participar en la rotación. Su papel es crear las dos credenciales y las identidades; el intercambio lo hace un proceso, disparado por el evento de caducidad próxima.

Y una advertencia sobre el generador de valores aleatorios, que produce sorpresas al refactorizar:

```
un valor aleatorio se genera UNA VEZ y se guarda en el estado
si se pierde el estado, se genera OTRO
y si el recurso que lo usa no se recrea, deja de coincidir
```

Por eso un valor aleatorio que es la contraseña de algo debería llevar activada la creación antes de destrucción y, mejor aún, no existir: es la estrategia 1 otra vez.

5. La auditoría de los seis rastros

Este programa ha encontrado el mismo tipo de escape en cinco sitios distintos. Reunirlos en una sola comprobación es el entregable de esta clase:

```
#!/usr/bin/env bash
# auditoria-secretos.sh
set -euo pipefail
fallos=0

# 1 · repositorio, incluido el historial (clases 061, 066)
echo "— historial del repositorio —"
gitleaks detect --no-banner --redact || fallos=1

# 2 · estado (clases 059, 087, 089)
echo "— estado —"
for d in infra/*/; do
  n=$(terraform -chdir="$d" state pull 2>/dev/null \
      | grep -Eco '"(password|secret_string|private_key|token)"' || true)
  [ "$n" = "0" ] || { echo "  $d: $n campos sensibles"; fallos=1; }
done

# 3 · ficheros de plan publicados como artefacto
echo "— artefactos de plan —"
find . -name '*.tfplan' -newermt '-90 days' -print | tee /tmp/planes
[ ! -s /tmp/planes ] || fallos=1

# 4 · registros de la canalización
echo "— registro detallado activado —"
grep -rn 'TF_LOG' .github/ .gitlab-ci.yml 2>/dev/null && fallos=1 || true

# 5 · salidas del módulo raíz (clase 089)
echo "— salidas —"
for d in infra/*/; do
  terraform -chdir="$d" output -json 2>/dev/null \
    | jq -r 'keys[]' | grep -Ei 'password|secret|token|key' \
    && fallos=1 || true
done

exit $fallos
```

Cinco comprobaciones, una por rastro evitable. La sexta —los registros del proveedor— no se comprueba: se evita con la estrategia 1.

Y la lista de comprobación de la clase, que recoge decisiones de seis partes anteriores:

- ningún secreto en el repositorio, ni en el historial
- credenciales sustituidas por identidad donde sea posible (7.ª vez)
- contraseñas de servicios gestionadas por el propio proveedor
- la plantilla crea contenedores de secretos, no valores
- ningún secreto como salida del módulo raíz
- estado cifrado, con acceso mínimo separado de lectura y escritura
- ficheros de plan no publicados como artefacto descargable
- registro detallado desactivado en la canalización
- campos rotados por otro sistema, declarados como ignorados
- auditoría de los cinco rastros, ejecutada en la canalización
- procedimiento escrito: rotar primero, corregir después, purgar al final

Once puntos. Y el último es el que más veces se hace al revés, con la misma consecuencia: mientras se corrige el mecanismo, la credencial expuesta sigue siendo válida.

Y un cierre que conecta con la parte 08: todo lo anterior protege el secreto en reposo y en la plantilla. Lo que ocurre cuando la canalización necesita credenciales para aplicar —y cómo se obtienen sin que existan— es la federación de las clases 026, 038, 050 y 059, y su forma definitiva es materia de entrega continua. Aquí basta con la regla que ya se ha repetido siete veces: la canalización no tiene claves; tiene identidad.

Ejemplo trabajado

CloudShop audita sus secretos reuniendo, por primera vez, todos los escapes que había ido encontrando por separado. La auditoría de los cinco rastros da resultados en cuatro de ellos.

Rastro 1 — el historial del repositorio.

```
hallazgos                                        6
  ya conocidos y rotados                         2   (clases 061, 066)
  nuevos                                         4
el más antiguo                                hace 3 años
```

Los cuatro nuevos: una clave de acceso en un fichero de ejemplo, dos contraseñas en ficheros de valores borrados hace dos años y un testigo en un mensaje de commit.

```
secretos en el historial                         6              6, todos rotados
escaneo del historial en la canalización    no había       en cada cambio
fichero de ejemplo con valores reales            1          plantilla sin valores
```

La cifra no baja porque el historial no se reescribe: se rota lo expuesto y se impide que vuelva a ocurrir. Es la ley 11 en su forma más literal.

Rastro 2 — el estado.

```
$ for d in infra/*/; do echo -n "$d: "; terraform -chdir="$d" state pull \
    | grep -Eco '"(password|secret_string|private_key)"'; done
infra/red/:        0
infra/datos/:      9
infra/plataforma/: 3
infra/tienda/:     2
```

Catorce campos sensibles. Y la clasificación por estrategia posible:

```
contraseñas de bases de datos                    4   → estrategia 1: el servicio
                                                      las gestiona
claves de API de terceros                        5   → estrategia 2: contenedor
                                                      sin valor
certificados y claves privadas                   3   → estrategia 2
valores aleatorios generados por la plantilla    2   → estrategia 3, aceptada
```

```
campos sensibles en el estado                14              2
contraseñas gestionadas por el proveedor      0              4
secretos con valor puesto por la plantilla    8              0
identidades con acceso a los estados         12              2
credenciales rotadas                          —             14
```

Los dos que quedan son valores aleatorios que un servicio antiguo exige en su creación, con el riesgo declarado y el campo ignorado para que su rotación no produzca desviación.

Rastro 3 — los ficheros de plan publicados.

```
$ gh run list --limit 200 --json databaseId -q '.[].databaseId' \
  | while read id; do gh run view "$id" --json artifacts \
      -q '.artifacts[]?.name' 2>/dev/null; done | sort | uniq -c
    187 tfplan
```

Ciento ochenta y siete planes guardados como artefacto, descargables por cualquiera del repositorio, con retención de noventa días. Cada uno con los valores de lo que iba a crear.

```
planes publicados como artefacto             187              0
lo que se publica en la revisión          el fichero       su representación
                                                            legible, sin valores
                                                            sensibles
retención                                   90 días        no aplica
artefactos purgados                            —              187
```

Rastro 4 — el registro detallado.

```
$ grep -rn 'TF_LOG' .github/workflows/
.github/workflows/infra.yml:34:      TF_LOG: DEBUG
```

Activado siete meses atrás para depurar un problema de proveedor, y nunca retirado. Los registros de todas las ejecuciones desde entonces contenían cuerpos de petición.

```
registro detallado en la canalización       activo         desactivado
ejecuciones con cuerpos de petición
  en los registros                            ~1.400            0
registros purgados                            —             los 7 meses
comprobación que lo impide                  no había      falla si aparece TF_LOG
```

Rastro 5 — las salidas.

El hallazgo de la clase 089 ya estaba corregido, y la comprobación automatizada encontró uno más en un estado que no se había revisado:

```
salidas con nombre de credencial              1                 0
```

Y el resultado global de la auditoría.

```
campos sensibles en los estados                 14              2
planes publicados como artefacto               187              0
registro detallado en la canalización        activo       desactivado
salidas con credenciales                          1              0
secretos del historial rotados                    2              6
identidades con acceso a los estados             12              2
credenciales rotadas en el ejercicio              —             21
comprobación de los cinco rastros             no había     en cada cambio
```

Veintiuna credenciales rotadas en una semana. Y el orden importó: se rotaron todas antes de tocar ninguna plantilla, porque mientras el mecanismo se corrige la credencial expuesta sigue valiendo.

Y la conclusión que el equipo escribió.

```
de los seis rastros, cuatro tenían hallazgos
ninguno estaba en el repositorio actual — el sitio que sí se revisaba
y el mayor volumen estaba en el rastro que nadie había considerado:
  187 ficheros de plan descargables durante 90 días
```

La lección que esta clase traslada al resto de la parte 07: la auditoría de secretos que solo mira el repositorio revisa uno de seis sitios, y en este caso el único donde no había nada nuevo. La corrección de fondo no es limpiar los seis rastros sino la estrategia 1 —que el secreto no exista para la plantilla—, que este programa lleva recomendando desde la clase 026 y que aquí elimina catorce de dieciséis casos. El secreto que no existe no se filtra, y sigue siendo la respuesta a la mayoría de las preguntas sobre secretos.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/092-secretos-y-datos-sensibles-en-iac/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 modelo-secretos-iac 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 modelo-secretos-iac 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 auditoría de secretos revisa el repositorio y no encuentra nada, y aun así hay filtraciones	El repositorio es uno de seis rastros; el estado, los planes publicados y los registros suelen tener más	Audita los cinco rastros evitables con una comprobación ejecutable y elimina el sexto con la estrategia 1.
Marcar un valor como sensible no impide que aparezca en el estado	La marca oculta la salida por pantalla; no cifra nada	Trátala como higiene de pantalla y protege el almacenamiento con acceso mínimo y cifrado del estado.
Cada plan propone devolver un secreto a su valor anterior	La plantilla declara un valor que otro sistema rota	Que la plantilla cree el contenedor y no el valor, o que el proveedor lo gestione; en último caso, declara el campo como ignorado.
Los registros de la canalización contienen cuerpos de petición	El registro detallado se activó para depurar y quedó activado	Desactívalo, purga los registros afectados y añade una comprobación que falle si vuelve a aparecer.
Un plan publicado como artefacto contiene los valores de lo que va a crear	Se publica el fichero binario en vez de su representación legible	Publica solo la salida legible sin valores sensibles y no conserves el fichero como artefacto descargable.
Un plan muestra (sensitive value) donde hacía falta revisar el cambio	Un valor sensible entró en una expresión y marcó todo el resultado	No mezcles el valor sensible con lo que hay que poder leer: construye la cadena donde se consume.

■ 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. Enumera los seis rastros que deja un secreto al pasar por una plantilla y di cuál no se puede evitar.
2. ¿Cuáles son las tres estrategias, en orden, y cuántos rastros elimina cada una?
3. ¿Qué hace y qué no hace marcar un valor como sensible?
4. ¿Por qué una plantilla que declara el valor de un secreto rotado produce desviación permanente?
5. ¿En qué orden se actúa cuando un secreto ya se ha expuesto, y por qué ese orden?

Referencias

- HashiCorp (2025). Sensitive data in state — por qué el estado guarda los valores y cómo protegerlo. <https://developer.hashicorp.com/terraform/language/state/sensitive-data>
- HashiCorp (2025). Ephemeral values and write-only arguments — valores que no se guardan en el estado. <https://developer.hashicorp.com/terraform/language/values/ephemeral>
- AWS (2025). Managed master user password for RDS — credencial generada y rotada por el servicio. <https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/rds-secrets-manager.html>
- Gitleaks (2025). Scanning git history for secrets — detección en el historial, no solo en el árbol actual. <https://github.com/gitleaks/gitleaks>
- Mozilla (2025). SOPS: encrypted files in the repository — cifrado de valores y gestión de la clave. <https://github.com/getsops/sops>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 07 en PDF · Manual integral

Anterior	Índice	Siguiente
← 091 · Validación, lint, pruebas y policy as code	Parte 07 · Programa	093 · CloudFormation, Bicep, Pulumi y Terraform →

Evaluación — 092 Secretos y datos sensibles en IaC

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 modelo-secretos-iac 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.

093 — CloudFormation, Bicep, Pulumi y Terraform

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: decision · Estado: EXECUTABLECORE

Propósito

Comparar las cuatro familias de herramientas por lo que cambia la operación, que son cuatro ejes y ninguno es la sintaxis: quién guarda el estado, quién ejecuta el motor, qué ocurre cuando una aplicación falla a medias y si el lenguaje es un lenguaje de programación. Las clases 047 y 059 ya midieron el primero; aquí se completan los otros tres y se llega a la conclusión que este programa lleva sugiriendo desde la parte 03: la disciplina es la misma en las cuatro, y es lo que decide si funciona.

Resultados de aprendizaje

Al finalizar podrás:

1. Comparar las cuatro familias por los ejes que cambian la operación, no por funciones.
2. Anticipar qué ocurre cuando una aplicación falla a medias en cada una.
3. Valorar el uso de un lenguaje de programación general con sus dos caras.
4. Elegir herramienta con un criterio defendible y sabiendo qué se hereda con ella.
5. Reconocer la parte de la práctica que no depende de la herramienta.

Conceptos centrales

Concepto	Comprensión verificable
estado externo frente a estado del proveedor	O hay un fichero que hay que custodiar, o el propio servicio recuerda lo que gestiona. Decide la mitad de las obligaciones operativas.
motor gestionado	El proveedor ejecuta el despliegue. No hay estado que proteger ni ejecución que orquestar, y se pierde control sobre el momento y sobre el detalle.
comportamiento ante fallo parcial	Qué queda cuando la mitad se aplicó y algo falló: reversión automática, o lo aplicado se queda. Cambia por completo el procedimiento de recuperación.
lenguaje general	Escribir infraestructura en un lenguaje de programación. Da bucles, abstracciones y pruebas conocidas, y quita la restricción que hacía revisable el resultado.
retraso del proveedor	Tiempo entre que una nube publica una función y la herramienta la soporta. Es cero en las herramientas de la propia nube y variable en las demás.
disciplina portable	Previsualizar, revisar, aplicar lo revisado, verificar convergencia, política sobre el resultado y secretos fuera. Es idéntica en las cuatro familias.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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. Cuatro ejes, y la sintaxis no es ninguno

Las comparativas habituales enumeran funciones. Los cuatro ejes que de verdad cambian el trabajo diario son otros:

	Estado	Motor	Fallo parcial	Lenguaje
Nativa de la nube (plantillas del proveedor)	Del proveedor	Gestionado	Revierte solo	Específico
Terraform	Fichero propio	Tuyo	Se queda	Específico
Lenguaje general con estado	Fichero propio	Tuyo	Se queda	General
Kit de desarrollo que genera plantillas	Del proveedor	Gestionado	Revierte solo	General

Y las consecuencias de cada columna, que son lo que se hereda al elegir.

El estado ya se midió en las clases 047 y 059, y el resumen se sostiene:

```
fichero propio      hay que custodiarlo, bloquearlo, cifrarlo y recuperarlo
                    → toda la clase 087
                    a cambio: se sabe QUÉ gestiona este código
                    y se detecta que alguien BORRÓ algo

del proveedor       no hay fichero que proteger ni secretos que se filtren por ahí
                    a cambio: no hay lista de pertenencia clara,
                    y detectar un borrado externo depende del servicio
```

El motor es el eje que menos se discute y más cambia la operación:

```
motor gestionado    el proveedor despliega, con su ritmo y sus reintentos
                    no hay agentes que operar, ni bloqueos, ni concurrencia
                    y hay poco control: el paralelismo, el orden fino
                    y los tiempos de espera los decide él

motor propio        control total del momento, del paralelismo y del alcance
                    y hay que operar el agente, su identidad y su bloqueo
```

La segunda fila incluye una ventaja que se nota al depurar: con motor propio se puede aplicar una parte, planificar sin refrescar o reintentar una operación concreta (clase 090). Con motor gestionado, esas herramientas no existen.

El retraso del proveedor es un argumento real y suele exagerarse:

```
herramienta de la propia nube   soporte el mismo día del anuncio
otras                           de días a meses, según el servicio
```

Y su importancia depende de un dato que cada equipo puede medir: cuántas veces al año necesita una función publicada esta semana. En la mayoría de las organizaciones la respuesta es pocas, y para esos casos existe la salida de declarar el recurso con la API genérica del proveedor hasta que llegue el soporte.

2. Qué pasa cuando falla a medias

Este es el eje que menos aparece en las comparativas y el que más cambia el procedimiento de recuperación.

Reversión automática. El motor deshace lo que ya había aplicado y devuelve el conjunto a su estado anterior.

```
ventaja   nunca queda un estado intermedio que nadie ha probado
coste     la reversión también deshace lo que SÍ funcionó
          y puede fallar ella misma, dejando un estado del que hay que salir a mano
          y en un despliegue grande, tarda
```

La segunda línea produce una situación conocida: el motor intenta revertir un recurso que no se puede eliminar —tiene protección, tiene dependencias, tiene datos— y el conjunto queda en un estado que exige intervención. Y ahí el

procedimiento no es evidente y conviene tenerlo escrito antes.

Lo aplicado se queda. Es el comportamiento de Terraform, y la clase 059 ya lo enunció: no hay transacción.

```
ventaja   lo que funcionó, funcionó; se corrige y se vuelve a aplicar
coste     hay un estado intermedio real, y hay que asumir que existirá
          → EXIGE que las plantillas sean idempotentes y reejecutables
```

Y de ahí sale la propiedad que este programa lleva pidiendo desde la clase 047: la respuesta a un fallo parcial es corregir y volver a aplicar. Una plantilla que solo funciona sobre un entorno vacío no sirve para operar, con ninguna herramienta.

La comparación honesta entre ambos comportamientos:

```
la reversión automática es mejor para        despliegues acotados y frecuentes
                                             de un conjunto pequeño
lo aplicado se queda es mejor para           conjuntos grandes, donde revertir
                                             todo por un fallo al final es peor
                                             que corregir y continuar
```

Y una coincidencia entre las dos familias que conviene señalar: ninguna revierte los efectos que no son recursos. Una migración de datos ejecutada por un gancho, un mensaje publicado, un fichero escrito — nada de eso vuelve atrás. Es el mismo límite que la clase 079 estableció para las vueltas atrás de despliegue, y sigue exigiendo lo mismo: cambios compatibles en ambos sentidos y una pregunta previa —«si esto falla a mitad, ¿qué queda roto?»—.

Y una diferencia operativa que aparece con motores gestionados y sorprende: la detección de desviación es una función del propio servicio en algunas plataformas, con lo que se obtiene sin montar la ejecución programada de la clase 090. Y suele tener límites: no todos los tipos de recurso, no todos los campos. Conviene comprobar la cobertura real antes de darla por buena, con la misma desconfianza que la clase 080 aplicó a las políticas de red.

3. El lenguaje general: dos caras

Escribir infraestructura en un lenguaje de programación completo resuelve problemas reales y quita una restricción que era útil. Merece las dos columnas.

Lo que resuelve:

```
abstracciones de verdad     clases, funciones, herencia
                            un módulo con lógica compleja se expresa mejor
bucles y condicionales      sin las contorsiones de un lenguaje declarativo
pruebas con las herramientas del lenguaje
                            unitarias, sin desplegar nada
reutilización de código     bibliotecas ya existentes, tipos compartidos
                            con la aplicación
un solo lenguaje            el equipo no aprende una sintaxis más
```

La tercera fila es la más valiosa y la menos citada: se pueden escribir pruebas unitarias sobre la lógica que decide la infraestructura, sin crear nada.

Lo que quita:

```
la restricción que hacía revisable el resultado
```

Un lenguaje declarativo limitado obliga a que el código se parezca al resultado. Con un lenguaje general se puede escribir esto:

```
una función que lee una base de datos y decide cuántos recursos crear
un bucle cuyo número de iteraciones depende de la hora
una jerarquía de clases de cinco niveles para tres tipos de servicio
```

Y nada de eso es revisable leyendo el código: hay que ejecutarlo para saber qué hace.

Y aquí está la parte que reconcilia las dos columnas y que conviene tener clara antes de elegir:

> El artefacto de revisión no es el código, es el plan. Con un lenguaje general, el plan sigue existiendo y sigue siendo el sitio donde se revisa, se aplica la política y se aprueban los borrados.

Eso reduce mucho el riesgo, y no lo elimina: un plan de doscientos recursos generado por un bucle cuyo criterio nadie entiende es revisable en su resultado y no en su intención. Por eso la disciplina que hay que añadir es de estilo:

```
la lógica que decide QUÉ existe: simple, plana, y probada
la complejidad, en las bibliotecas de apoyo, no en la definición
ninguna consulta a sistemas externos en tiempo de definición
```

```
   → hace que el resultado dependa de cuándo se ejecute
```

La tercera es la más importante: una definición que consulta una base de datos o una API para decidir qué crear no es reproducible, y rompe la propiedad que hace útil todo lo demás.

Y un matiz sobre los kits de desarrollo que generan plantillas nativas: dan el lenguaje general y conservan el motor gestionado, así que heredan la reversión automática y la ausencia de estado propio. Es una combinación coherente y con un coste conocido: el resultado es una plantilla generada, a veces grande y poco legible, y depurar exige leerla.

4. Elegir, y qué se hereda con cada elección

Un criterio defendible en cinco preguntas, en orden:

```
1. ¿una nube o varias?
   una sola y sin previsión de cambiar  → la nativa es una opción seria
   varias, o Kubernetes, o servicios de terceros → una herramienta común

2. ¿quién va a operarlo?
   equipo pequeño sin capacidad de operar agentes → motor gestionado
   equipo de plataforma con canalizaciones → motor propio

3. ¿el equipo ya tiene un lenguaje?
   sí, y la infraestructura la escriben desarrolladores → lenguaje general
   la escribe un equipo de plataforma → específico, más restringido y más legible

4. ¿cuánta lógica hace falta de verdad?
   poca, casi siempre → un lenguaje declarativo basta y se lee mejor

5. ¿qué se hereda?
   → la tabla del primer apartado, entera
```

Y la respuesta más común en organizaciones medianas, dicha sin adornos: una herramienta común con motor propio y lenguaje específico, porque cubre varias nubes, Kubernetes y servicios de terceros con un solo flujo, y porque la lógica que hace falta es poca.

Y dos situaciones en las que la nativa gana con claridad:

```
una sola nube, equipo pequeño, sin canalización de plataforma
  → el motor gestionado quita la mitad del trabajo de las clases 087 y 090

ofrecer plantillas a clientes o a otras organizaciones
  → el formato nativo lo entiende cualquiera de esa nube sin instalar nada
```

Y la situación que aparece de verdad y que ninguna comparativa contempla: una organización con tres herramientas a la vez, porque cada equipo eligió la suya. Consolidar cuesta y no consolidar también, y la decisión se toma con dos cifras:

```
coste de consolidar    reescribir e importar, medible en días-persona
coste de no hacerlo    tres canalizaciones, tres políticas, tres auditorías,
                       tres formaciones y ninguna revisión cruzada posible
```

Y una salida intermedia que funciona mejor de lo que parece: consolidar la disciplina antes que la herramienta. Las mismas políticas, la misma secuencia de revisión, el mismo tratamiento de secretos y la misma detección de desviación, con tres herramientas distintas. Eso captura la mayor parte del beneficio y no exige reescribir nada.

Y para migrar de una herramienta a otra, cuando se decide, el procedimiento tiene una forma conocida:

```
1. la herramienta nueva IMPORTA lo que existe (clase 087)
2. se verifica con una previsualización sin cambios
3. la herramienta antigua deja de gestionarlo, sin destruirlo (clase 090)
4. se retira la antigua cuando ya no gestione nada
```

El paso 3 es el que evita el desastre: retirar de la gestión, no destruir. Y el 2 es la comprobación que decide si el paso 1 se hizo bien, exactamente como en la clase 087.

5. Lo que no depende de la herramienta

Con cuatro familias comparadas, lo que queda es la conclusión que este programa lleva sugiriendo desde la parte 03: la mayor parte de la práctica es idéntica.

```
previsualizar antes de aplicar            047 · 059 · 081 · 090
aplicar exactamente lo previsualizado     059 · 090
verificar la convergencia después         073 · 085 · 090
```

revisión legible: sin ruido	047 · 081 · 085 · 090
política sobre el RESULTADO, no el código	091
secretos fuera, y rotar lo expuesto	047 · 059 · 061 · 087 · 092
versiones fijadas de todo	047 · 059 · 062 · 081 · 088
un estado o alcance por radio de impacto	047 · 059 · 087
proteger lo que no se puede perder	042 · 059 · 077 · 086
identidad federada, sin claves	026 · 038 · 050 · 059 · 092
refactorizar sin destruir	087 · 090
detección de desviación con señal	085 · 090

Doce prácticas, todas presentes en las cuatro familias con nombres distintos, y todas responsables de más incidentes que cualquier decisión de herramienta.

Y los tres criterios que la clase 085 fijó para juzgar una solución de esta parte siguen siendo los mismos y siguen sin depender de la herramienta:

1. ¿se puede ver el cambio antes de aplicarlo, y es legible?
2. ¿lo que se revisó es exactamente lo que se aplica?
3. ¿alguien sabría que el sistema dejó de converger?

Y una observación que conviene decir con claridad, porque ahorra discusiones improductivas:

> Un equipo con una herramienta mediocre y las doce prácticas tiene una infraestructura operable. Un equipo con la mejor herramienta y sin ellas tiene los mismos incidentes que tenía antes, con más ficheros.

Y la lista de comprobación de la clase, que es de decisión y no de configuración:

- [ ] los cuatro ejes evaluados para el caso concreto, no una tabla de funciones
- [ ] el comportamiento ante fallo parcial, conocido y con procedimiento escrito
- [ ] si el lenguaje es general: reglas de estilo sobre la lógica de definición
- [ ] ninguna consulta a sistemas externos en tiempo de definición
- [ ] si hay varias herramientas: la disciplina consolidada aunque no lo esté la herramienta
- [ ] si se migra: importar, verificar sin cambios, retirar de la gestión, destruir nunca
- [ ] las doce prácticas portables, presentes con independencia de la elección

Ejemplo trabajado

CloudShop tiene tres herramientas de infraestructura como código y una discusión anual sobre cuál usar. El ejercicio de este año se hace midiendo en vez de opinando, y la conclusión no es la que nadie esperaba.

El punto de partida.

equipo de plataforma	Terraform, 4 estados, 88 % de la infraestructura
equipo de datos	plantillas nativas de la nube, 9 %
equipo de aplicación	un kit de desarrollo en el lenguaje de la aplicación, 3 %

Y los tres argumentos de la discusión anual, siempre los mismos:

"las plantillas nativas no necesitan estado"
"con el kit escribimos infraestructura en el mismo lenguaje"
"tener tres herramientas es insostenible"

La medición: incidentes por causa, últimos doce meses.

causa	incidentes	herramienta
falta de previsualización revisada	6	las tres
plantilla no idempotente	4	las tres
secreto en un rastro no auditado	3	las tres
desviación no detectada	5	las tres
recurso destruido sin querer	2	Terraform, kit
estado perdido o bloqueado	2	Terraform
reversión automática que falló a medias	1	nativa
lógica de definición imposible de revisar	1	kit

Veinticuatro incidentes. Dieciocho de los veinticuatro son de las doce prácticas portables y no de la herramienta: ocurrieron con las tres, y habrían ocurrido con cualquier otra.

Los seis restantes sí son atribuibles:

dos por destrucción	falta de protección contra el borrado (clases 059, 086)
dos por el estado	custodia y bloqueo (clase 087)
uno por reversión	el motor gestionado no pudo revertir y quedó a medias
uno por lógica	un bucle en el kit cuyo criterio nadie entendía

La decisión, que no fue consolidar.

```
consolidar en una herramienta        34 días-persona
consolidar la DISCIPLINA               9 días-persona
```

Y lo que cubría cada opción:

```
consolidar la herramienta   los 6 incidentes atribuibles, y de rebote
                            facilitaría las prácticas
consolidar la disciplina    los 18 incidentes portables, en las tres herramientas
```

Se hizo la segunda primero. Y las nueve jornadas se repartieron así:

```
política común sobre el resultado, para las tres            3 días
detección de desviación programada, para las tres           2 días
auditoría de los cinco rastros de secretos (clase 092)      2 días
protección contra destrucción en lo que no se puede perder  1 día
procedimiento escrito de fallo parcial, uno por herramienta 1 día
```

La política común fue lo más interesante del ejercicio: las tres herramientas producen una representación estructurada de lo que van a hacer, así que las mismas diecinueve reglas se pudieron aplicar a las tres con adaptadores pequeños.

```
reglas de política                    19, solo en Terraform   19, en las tres
detección de desviación                1 de 3 herramientas    3 de 3
audita rastros de secretos             1 de 3                 3 de 3
procedimiento de fallo parcial escrito 0 de 3                 3 de 3
recursos con protección de borrado     6 de 41               41 de 41
```

Los seis meses siguientes.

```
incidentes de causa portable          18 → 2
incidentes atribuibles a la herramienta 6 → 3
```

Los dos portables restantes fueron plantillas no idempotentes en el kit, corregidas.

Y con esos datos, la decisión sobre consolidar la herramienta se tomó por fin con criterio:

```
las plantillas nativas se quedan       el equipo de datos es pequeño,
                                       no opera canalizaciones, una sola nube
                                       y el motor gestionado le quita trabajo real
el kit se retira                       3 % de la infraestructura, un incidente
                                       por lógica irrevisable, y el equipo
                                       prefería no mantener dos flujos
Terraform sigue                        cubre varias nubes y Kubernetes
```

La migración del kit se hizo con el procedimiento de esta clase:

```
recursos importados                    31
verificados con previsualización vacía 31 de 31, a la tercera iteración
retirados de la gestión del kit        31, sin destruir
recursos destruidos en el proceso       0
esfuerzo                                4 días-persona
```

La conclusión que se escribió, y que cerró la discusión anual.

```
de 24 incidentes en un año, 18 no dependían de la herramienta
consolidar la disciplina costó 9 días y eliminó 16 de esos 18
consolidar la herramienta habría costado 34 días y eliminado 6
y la decisión final NO fue una sola herramienta:
  dos, cada una donde su modelo operativo encaja
```

La lección que esta clase traslada al resto de la parte 07: la discusión sobre herramientas consume tiempo desproporcionado respecto de lo que decide. Tres de cada cuatro incidentes eran de las doce prácticas portables, presentes o ausentes con independencia de la elección. Y cuando por fin se eligió, el criterio no fue cuál es mejor sino qué modelo operativo encaja con cada equipo: motor gestionado donde no hay capacidad de operar canalizaciones, motor propio donde hay varias nubes que cubrir.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/093-cloudformation-bicep-pulumi-y-
terraform/lab.py
```

El laboratorio selecciona el motor de práctica decision 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 adr-herramienta-iac 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 matriz de decisión ponderada y un ADR. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye adr-herramienta-iac 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 discusión sobre qué herramienta usar se repite cada año sin resolverse	Se compara por funciones en vez de por los ejes que cambian la operación	Mide los incidentes del último año por causa: la mayoría suele ser de prácticas portables, no de la herramienta.
Una aplicación falla a medias y el motor no consigue revertir	La reversión automática intenta eliminar algo protegido o con dependencias	Ten escrito el procedimiento de salida antes de necesitarlo; el comportamiento ante fallo parcial se conoce, no se descubre.
Un cambio en el kit genera doscientos recursos y nadie sabe por qué	La lógica de definición usa bucles o consultas cuyo criterio no se puede leer	Reglas de estilo: lógica de definición simple y plana; la complejidad, en bibliotecas de apoyo probadas.
El resultado de una ejecución depende de cuándo se ejecute	La definición consulta un sistema externo para decidir qué crear	Prohíbe las consultas en tiempo de definición: rompen la reproducibilidad, que es la propiedad que sostiene todo lo demás.
Migrar entre herramientas destruye recursos	Se retiró la antigua antes de que la nueva los gestionara, o se destruyó en vez de retirar de la gestión	Importar, verificar con previsualización vacía, retirar de la gestión sin destruir, y solo entonces retirar la antigua.
Se adopta la mejor herramienta y los incidentes siguen igual	Las doce prácticas portables no estaban antes y siguen sin estar	Consolida la disciplina antes que la herramienta: cubre más incidentes por menos esfuerzo.

■ 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 ejes que cambian la operación, y qué se hereda con cada opción?
2. ¿Qué ventaja y qué coste tiene la reversión automática frente a que lo aplicado se quede?
3. ¿Qué resuelve y qué quita usar un lenguaje de programación general, y qué reduce el riesgo?
4. ¿Qué procedimiento sigue una migración entre herramientas sin destruir nada?
5. Enumera cinco de las doce prácticas que no dependen de la herramienta.

Referencias

- AWS (2025). CloudFormation stack failure options and rollback — comportamiento ante fallo parcial. <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/stack-failure-options.html>
- Microsoft (2025). Bicep vs ARM templates and deployment behaviour — motor gestionado y modos de despliegue. <https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview>
- HashiCorp (2025). Terraform: no partial rollback — por qué las plantillas deben ser reejecutables. <https://developer.hashicorp.com/terraform/language/resources/behavior>
- Pulumi (2025). Programming model and preview — lenguajes generales con previsualización del cambio. <https://www.pulumi.com/docs/concepts/>
- AWS (2025). CDK: synthesized templates — generar plantillas nativas desde un lenguaje general. <https://docs.aws.amazon.com/cdk/v2/guide/home.html>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 07 en PDF · Manual integral

Anterior	Índice	Siguiente
← 092 · Secretos y datos sensibles en IaC	Parte 07 · Programa	094 · Ansible e imagen dorada para configuración →

Evaluación — 093 CloudFormation, Bicep, Pulumi y Terraform

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 adr-herramienta-iac 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.

094 — Ansible e imagen dorada para configuración

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: configuration · Estado: EXECUTABLECORE

Propósito

Resolver lo que queda fuera de la infraestructura declarativa: configurar lo que hay dentro de una máquina. Hay dos filosofías —hornear la configuración en una imagen y desplegarla inmutable, o crear la máquina y converger su contenido— y la elección no es de gusto: depende de si el recurso es ganado o mascota, la distinción que fijó la clase 085. La clase defiende el orden correcto —imagen primero, gestión de configuración solo para lo que quede— y muestra por qué la idempotencia aquí es una propiedad de cada tarea y no de la herramienta.

Resultados de aprendizaje

Al finalizar podrás:

1. Elegir entre imagen inmutable y convergencia según la naturaleza del recurso.
2. Construir una imagen con la misma disciplina de la clase 062: base fijada, sin secretos, versionada.
3. Escribir tareas idempotentes de verdad y detectar las que no lo son.
4. Usar la simulación de cambios sabiendo qué no puede predecir.
5. Reducir la gestión de configuración a los casos que de verdad la necesitan.

Conceptos centrales

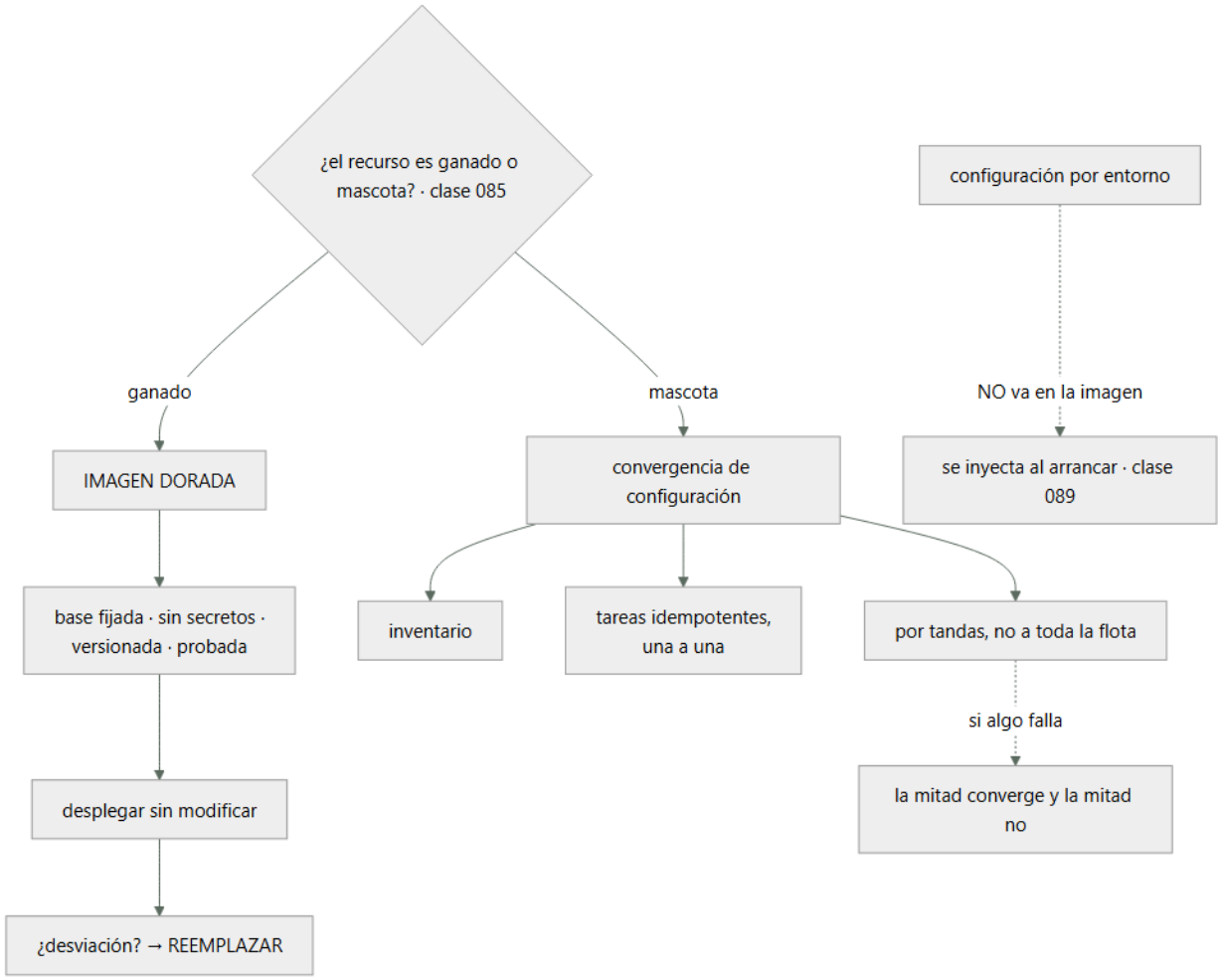
Concepto	Comprensión verificable
imagen dorada	Imagen de máquina con la configuración ya aplicada. Se despliega sin modificarla, así que no hay desviación posible dentro: la corrección de un problema es reemplazar.
convergencia de configuración	Aplicar tareas sobre una máquina existente hasta alcanzar el estado deseado. Necesaria para lo que no se puede reconstruir.
idempotencia por tarea	En estas herramientas la propiedad no la garantiza el motor: la garantiza cada módulo. Una tarea que ejecuta una orden arbitraria no es idempotente salvo que se la haga.
simulación de cambios	Ejecución que informa de lo que haría sin hacerlo. Su fidelidad depende de que cada módulo la soporte, así que no es equivalente a un plan.
ejecución por tandas	Aplicar sobre una fracción de la flota cada vez. Es el despliegue progresivo de la clase 079 en máquinas, y sin ella un error llega a todo a la vez.
canalización de imagen	Construcción versionada, probada y publicada de la imagen. Es la clase 062 aplicada a máquinas, con las mismas reglas.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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. Imagen o convergencia: lo decide la clase 085

La pregunta no es cuál herramienta es mejor sino qué es el recurso:

```
GANADO      se reemplaza sin ceremonia
  → imagen dorada: la configuración va dentro y no se toca
  → la respuesta a cualquier problema es sustituir la máquina

MASCOTA     se cuida y se repara
  → convergencia: hay que llevarla al estado deseado sin destruirla
  → y hay que asumir la desviación como estado normal
```

Y las ventajas del primero, que son las que justifican mover todo lo posible a esa categoría:

```
arranque rápido        no hay nada que instalar al arrancar
reproducible           dos máquinas de la misma imagen son idénticas
sin desviación dentro  nadie la modifica porque se reemplaza, no se repara
sin dependencias en el arranque
                       una máquina que instala paquetes al arrancar depende
                       de que el repositorio esté disponible EN ESE MOMENTO
probable de verdad     la imagen se prueba una vez y se despliega N
```

La cuarta merece detalle porque produce incidentes en el peor momento: una flota que instala software al arrancar no puede escalar si el repositorio de paquetes está caído, y el escalado ocurre justo cuando hay más carga. Con imagen dorada, arrancar no depende de nada externo.

Y el coste que hay que aceptar:

```
un cambio pequeño exige construir una imagen y reemplazar la flota
  → construir tarda minutos y desplegar, lo que tarde la sustitución (clase 079)
hay que gestionar el ciclo de vida de las imágenes
  → versiones, retención y limpieza
```

Y el error que hay que evitar por encima de todo, que es el de la clase 062 con otro artefacto:

```
la configuración POR ENTORNO no va dentro de la imagen
  → una imagen por entorno rompe "construir una vez y promover"
  → y lo que se prueba en preproducción deja de ser lo que corre en producción
```

La imagen lleva lo común: el sistema endurecido, los agentes, las herramientas, el tiempo de ejecución. Lo que distingue a un entorno se inyecta al arrancar, con los mecanismos de la clase 089.

Y los tres casos en los que la convergencia sigue siendo necesaria, que conviene enumerar para no fingir que no existen:

```
1. equipos que no se pueden reconstruir
   servidores físicos, sistemas heredados con estado local,
   aparatos de red
2. la construcción de la propia imagen
   la herramienta de configuración es una buena forma de definir qué lleva
3. operaciones puntuales sobre una flota existente
   una rotación, una recogida de evidencia, un parche urgente
   antes de que la imagen nueva esté lista
```

El segundo es el uso más productivo y el menos citado: la misma herramienta que converge máquinas sirve para definir el contenido de la imagen, con lo que se aprovecha el conocimiento sin heredar los problemas de la convergencia en producción.

2. La canalización de imagen es la clase 062 otra vez

Construir una imagen de máquina tiene exactamente las mismas reglas que construir una de contenedor, y conviene enunciarlas así porque el equipo ya las conoce:

```
source "amazon-ebs" "base" {
  source_ami   = "ami-0a1b2c3d4e5f6"        # fijada, no "la última" (clase 062)
  instance_type = "t3.medium"
  ssh_username  = "admin"
  ami_name      = "cls-base-${local.timestamp}"
  tags = {
    origen_ami  = "ami-0a1b2c3d4e5f6"
    revision    = var.revision               # trazabilidad (clase 061)
    construida  = local.timestamp
  }
}

build {
  sources = ["source.amazon-ebs.base"]

  provisioner "ansible" {
    playbook_file = "configuracion/base.yml"
  }

  provisioner "shell" {
    inline = ["sudo /usr/local/bin/verificar-endurecimiento.sh"]
  }
}
```

Las reglas heredadas de la clase 062, una a una:

```
base fijada por identificador       y un proceso que proponga actualizarla
sin secretos dentro                  ni en variables, ni en ficheros, ni en
                                     el historial de construcción
lo mínimo necesario                  menos software es menos superficie
capas estables abajo                 lo que cambia poco, primero
versionada y trazable al commit      etiquetas con la revisión
construir una vez y promover         la MISMA imagen en los cuatro entornos
```

Y la que esta clase añade y no existía con contenedores: la imagen hay que arrancarla y probarla.

```
una imagen de contenedor se puede inspeccionar por capas
una imagen de máquina hay que ARRANCARLA para saber si funciona
```

Por eso la canalización termina con una prueba real, que es el equivalente de la prueba negativa de la clase 067:

```
# arrancar una instancia de la imagen recién construida y comprobar
$ instancia=$(aws ec2 run-instances --image-id $NUEVA --instance-type t3.micro \
    --query 'Instances[0].InstanceId' --output text)
$ ./esperar-lista.sh $instancia
```

```
$ ./verificar-imagen.sh $instancia
✓ arranca y responde                        42 s
✓ el servicio arranca sin red externa       correcto
✓ ningún proceso como usuario cero salvo los del sistema
✓ puertos escuchando: solo los previstos    22, 8080
✓ agente de registro y de métricas activos
✓ sin secretos en el sistema de ficheros    0 coincidencias
✓ actualizaciones de seguridad al día       0 pendientes
7/7 correctas
$ aws ec2 terminate-instances --instance-ids $instancia
```

La segunda comprobación es la que más rinde: arrancar sin red externa demuestra que la imagen no depende de descargar nada, que era la ventaja principal.

Y el ciclo de vida de las imágenes, que se descuida y produce dos problemas:

```
imágenes antiguas sin retirar     coste de almacenamiento e inventario confuso
imágenes en uso retiradas         una plantilla de arranque que apunta a una
                                  imagen borrada NO PUEDE ESCALAR
```

El segundo es el peligroso y es el mismo que la clase 067 encontró con el registro: la retención se define por referencias, no solo por fecha. Una imagen referenciada por alguna plantilla de arranque activa no se borra, tenga la antigüedad que tenga.

3. La idempotencia aquí es de cada tarea

La clase 085 definió la idempotencia. En las herramientas de configuración hay un matiz importante: el motor no la garantiza.

```
# idempotente: el módulo comprueba el estado antes de actuar
- name: Instalar el agente
  ansible.builtin.package:
    name: agente-observabilidad
    state: present

- name: Configurar el agente
  ansible.builtin.template:
    src: agente.conf.j2
    dest: /etc/agente/agente.conf
    owner: agente
    mode: "0640"
  notify: reiniciar agente

# NO idempotente: ejecuta siempre, informa de cambio siempre
- name: Preparar el directorio
  ansible.builtin.shell: mkdir -p /opt/app && chown app /opt/app
```

La tercera tarea se ejecuta en cada pasada, informa de que ha cambiado algo aunque no haya cambiado nada, y contamina el resumen. Y en el peor caso hace algo destructivo cada vez.

Las dos formas de arreglarla:

```
# A · usar el módulo que ya es idempotente
- name: Preparar el directorio
  ansible.builtin.file:
    path: /opt/app
    state: directory
    owner: app
    mode: "0755"

# B · si no hay módulo, declarar cuándo NO hay que ejecutar
- name: Inicializar el almacén
  ansible.builtin.command: /usr/local/bin/init-almacen
  args:
    creates: /var/lib/almacen/.inicializado
```

Y la comprobación que detecta las tareas no idempotentes, que es la de la clase 085 aplicada aquí:

```
$ ansible-playbook base.yml            # primera pasada
$ ansible-playbook base.yml | tail -3  # segunda pasada seguida

PLAY RECAP
nodo-14  : ok=42  changed=0  unreachable=0  failed=0
```

changed=0 en la segunda pasada es el criterio. Cualquier otra cifra señala tareas que actúan siempre, y localizarlas es directo:

```
$ ansible-playbook base.yml --diff | grep -B3 'changed:'
```

Y la simulación de cambios es el equivalente del plan, con una limitación que hay que conocer:

```
$ ansible-playbook base.yml --check --diff
```

```
su fidelidad depende de que CADA módulo la soporte
  los módulos de fichero y paquete la soportan bien
  una orden arbitraria no puede predecir nada: se salta o se ejecuta
y una tarea cuyo resultado depende de otra anterior que no se ejecutó
  puede fallar en la simulación sin que haya ningún problema real
```

Por eso no es equivalente a un plan de infraestructura y no debe tratarse como tal. Es útil y no es una garantía.

Y dos mecanismos que evitan que un error llegue a toda la flota a la vez:

```
- hosts: web
  serial: "20%"                 # por tandas
  max_fail_percentage: 0        # si falla una, se detiene
  any_errors_fatal: false
```

Es el despliegue progresivo de la clase 079 en máquinas, y sin ello una tarea equivocada se aplica a doscientos servidores en dos minutos. Y con volumen alto conviene además comprobar antes en un subconjunto:

```
$ ansible-playbook base.yml --limit 'web[0:2]' --check --diff
```

4. Inventario, variables y la precedencia otra vez

El inventario es la lista de máquinas y sus grupos, y conviene que sea dinámico:

```
inventario estático    un fichero con nombres; se desincroniza en cuanto
                       la flota es elástica
inventario dinámico    se consulta al proveedor por etiquetas
                       → las etiquetas de las clases 025, 049 y 089 se cobran aquí
```

Y una consecuencia práctica que justifica el trabajo de etiquetado de las clases anteriores: con inventario dinámico, los grupos son consultas y no listas que mantener.

```
plugin: amazon.aws.aws_ec2
regions: [eu-west-1]
filters:
  tag:gestionado-por: terraform
keyed_groups:
  - key: tags.sistema
    prefix: sistema
  - key: tags.entorno
    prefix: entorno
```

Y las variables repiten el problema de la clase 089 con una versión aún más complicada: hay más de veinte niveles de precedencia. La consecuencia práctica es la misma y la recomendación también:

```
usar POCOS niveles, y de forma consistente
  variables de grupo         lo común por rol o por entorno
  variables de máquina       solo lo que de verdad es de esa máquina
  variables de ejecución     lo que decide quien lanza, y queda registrado

y evitar el resto
```

Un sistema que usa doce de los veinte niveles es un sistema donde nadie puede predecir de dónde sale un valor, y la depuración pasa por una orden que conviene conocer:

```
$ ansible-inventory --host nodo-14 --yaml
$ ansible nodo-14 -m ansible.builtin.debug -a "var=hostvars[inventory_hostname]"
```

Y sobre los secretos, la regla es la de la clase 092 y no cambia:

```
la herramienta ofrece un mecanismo de cifrado en el repositorio
  → funciona, y la clave pasa a ser el activo
mejor: leer del gestor de secretos con la identidad de la máquina
  → séptima aparición: la máquina se autentica con su identidad
```

```
- name: Obtener la credencial
  ansible.builtin.set_fact:
    bd_password: "{{ lookup('amazon.aws.aws_secret', 'cls/bd', region='eu-west-1') }}"
```

```
no_log: true
```

La última línea es obligatoria y se olvida siempre: sin ella, el valor aparece en la salida de la ejecución, que es el rastro 4 de la clase 092.

Y una precaución sobre los registros de estas herramientas: por defecto son verbosos y muestran el contenido de los ficheros que escriben. Una plantilla con una contraseña dentro la imprime en la salida salvo que la tarea se marque explícitamente.

5. Reducir la convergencia a lo imprescindible

El objetivo de la clase es que quede poco. La secuencia para conseguirlo:

```
1. inventariar qué configura de verdad la herramienta
2. mover a la imagen todo lo que sea común y estable
3. mover a la inyección en el arranque lo que dependa del entorno
4. dejar en convergencia solo lo que no se puede reconstruir
```

Y el paso 3 merece precisión porque hay tres mecanismos y conviene elegir:

```
datos de arranque de la instancia   lo que se sabe al crearla; queda visible
                                     en la definición de la instancia
servicio de metadatos               consulta del propio recurso
configuración leída al arrancar     del gestor de configuración o de secretos,
                                     con la identidad de la máquina
```

La tercera es la mejor y es la que hace innecesaria la convergencia para casi todo lo que queda: la máquina arranca, se identifica y lee lo que le corresponde.

Y el resultado esperable de este ejercicio, con las cifras que suele dar:

```
tareas de configuración                ~200            ~25
tiempo de arranque de una máquina      varios minutos  menos de un minuto
dependencia de repositorios externos
  al arrancar                          sí              no
desviación dentro de las máquinas      inevitable      imposible: se reemplazan
```

Y la lista de comprobación de la clase:

```
imagen
  ■ base fijada, con proceso que proponga actualizarla
  ■ sin secretos, ni en variables ni en el historial de construcción
  ■ una sola imagen para todos los entornos
  ■ versionada y trazable al commit
  ■ probada arrancándola, incluida la prueba sin red externa
  ■ retención por referencias, no solo por fecha

convergencia
  ■ segunda pasada con cero cambios
  ■ ninguna tarea de orden arbitraria sin condición de omisión
  ■ ejecución por tandas, con detención ante fallo
  ■ inventario dinámico por etiquetas
  ■ pocos niveles de precedencia de variables, y consistentes
  ■ secretos del gestor con la identidad de la máquina, y sin registrar
  ■ una lista escrita de por qué cada tarea restante no puede ir en la imagen
```

El último punto es el que mantiene el resultado en el tiempo. Sin él, la lista de tareas vuelve a crecer: cada urgencia añade una, y en dos años se está otra vez en doscientas.

Y el cierre que conecta con la clase siguiente: todo lo anterior produce un artefacto —una imagen probada— y un mecanismo para llevarla a producción. Ofrecer eso a otros equipos, de forma que puedan usarlo sin conocer nada de esta parte, es lo que la clase 095 llama un camino asfaltado.

Ejemplo trabajado

CloudShop configura su flota de máquinas con doscientas tareas que se ejecutan al arrancar. El arranque tarda seis minutos, hay desviación en un tercio de la flota y un incidente de escalado obliga a revisarlo todo.

El incidente que lo desencadenó.

```
11:40  pico de tráfico; el escalado pide 12 máquinas nuevas
11:42  las 12 arrancan y empiezan a instalar paquetes
11:43  el repositorio de paquetes de la distribución no responde
```

```
11:49  las 12 máquinas siguen sin estar listas
11:52  el escalado pide 8 más; mismo resultado
12:04  el repositorio vuelve; las máquinas terminan de arrancar
```

Veinticuatro minutos sin poder escalar durante un pico, por una dependencia externa en el momento del arranque. Y el diagnóstico posterior encontró más:

```
tiempo medio de arranque              6 min 20 s
desviación: máquinas cuya configuración
  no coincide con la esperada         31 % de la flota
causa de la desviación                tareas ejecutadas en momentos distintos,
                                      con versiones distintas de los paquetes
```

La segunda cifra es la consecuencia inevitable de instalar al arrancar: dos máquinas creadas con una semana de diferencia no son iguales, porque los paquetes que descargan no son los mismos.

El inventario de las doscientas tareas.

```
común y estable
  sistema endurecido, agentes, herramientas    118    → imagen
  tiempo de ejecución y bibliotecas             34    → imagen
depende del entorno
  configuración de la aplicación                21    → inyección al arrancar
  destinos y credenciales                       12    → gestor, con identidad
no se puede reconstruir
  dos servidores heredados de facturación       11    → convergencia
no hacía falta
  tareas de sistemas retirados                   4    → eliminadas
```

La canalización de imagen.

```
construcción                          8 min 40 s
prueba con arranque real              2 min 10 s
comprobaciones de la prueba           7, todas obligatorias
cadencia                              semanal, y ante parche de seguridad
```

Y las dos comprobaciones que fallaron la primera vez:

```
✗ arranca sin red externa
    el agente de observabilidad descargaba su configuración al arrancar
✗ sin secretos en el sistema de ficheros
    una clave de registro de paquetes quedaba en /root/.netrc
```

La segunda es la ley 11 otra vez, ahora en una imagen de máquina: el fichero se creaba durante la construcción y se borraba al final, y seguía en la imagen porque la imagen se captura del disco, no de las capas. Se corrigió limpiando antes de la captura y rotando la clave.

El resultado.

```
tareas ejecutadas al arrancar              200              0
tiempo de arranque                    6 min 20 s         38 s
dependencias externas al arrancar     repositorio       ninguna
desviación en la flota                  31 %          imposible
máquinas idénticas entre sí               no              sí
tareas de convergencia restantes          200             11
```

Y la prueba del escalado, repetida a propósito con el repositorio de paquetes bloqueado:

```
12 máquinas nuevas, con el repositorio inalcanzable
  todas listas y sirviendo en 41 s                                    ✓
```

Las once tareas restantes, y por qué se quedan.

```
dos servidores de facturación heredados, con estado local que no se puede
reconstruir y un contrato de soporte que exige esa configuración exacta
  → convergencia, ejecutada por tandas de uno, con simulación previa
  → y una fecha de revisión: cuando se sustituya ese sistema, desaparecen
```

Y la lista escrita de por qué cada una no puede ir en la imagen, que es lo que impide que la cifra vuelva a crecer.

Y dos hallazgos de la revisión de las tareas.

```
$ ansible-playbook base.yml           # primera pasada
$ ansible-playbook base.yml | tail -1
nodo-14 : ok=200 changed=37 unreachable=0 failed=0
```

Treinta y siete tareas informaban de cambio en cada pasada. Al revisarlas:

```
tareas con orden arbitraria sin condición        29
tareas que reescribían un fichero con marca de tiempo dentro   6
tareas que reiniciaban un servicio siempre        2
```

Las dos últimas eran las importantes: reiniciaban dos servicios en cada ejecución de la herramienta, que se ejecutaba cada hora. Sesenta reinicios diarios de dos servicios, que nadie había relacionado con los picos de latencia que aparecían «a la hora en punto».

```
cambios en la segunda pasada                  37               0
reinicios de servicio no intencionados     ~60/día             0
picos de latencia a la hora en punto          sí               no
```

Resumen:

```
tiempo de arranque                          6 min 20 s        38 s
dependencias externas al arrancar           repositorio     ninguna
desviación en la flota                         31 %         imposible
tareas de convergencia                          200            11
cambios en la segunda pasada                     37             0
reinicios no intencionados al día               ~60             0
secretos en la imagen                             1             0
escalado con el repositorio caído          no funciona      41 s
```

La lección que esta clase traslada al resto de la parte 07: el incidente que lo desencadenó no fue de configuración sino de dependencia en el momento del arranque, y desaparece por completo al hornear. Y el hallazgo colateral —treinta y siete tareas no idempotentes, dos de ellas reiniciando servicios sesenta veces al día— confirma lo que la clase 085 enunció: la idempotencia aquí es de cada tarea, y se comprueba ejecutando dos veces seguidas. Nadie lo había hecho en cuatro años.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/094-ansible-e-imagen-dorada-para-configuracion/lab.py
```

El laboratorio selecciona el motor de práctica configuration 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 configuracion-servidor en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es configuración separada, validada y promovible. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye configuracion-servidor 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 flota no puede escalar cuando un repositorio externo está caído	Las máquinas instalan software al arrancar, así que dependen de él en el peor momento	Hornea el software en la imagen y comprueba en la canalización que arranca sin red externa.
Dos máquinas creadas con semanas de diferencia no son iguales	Cada una descargó las versiones disponibles ese día	Imagen dorada versionada: dos máquinas de la misma imagen son idénticas por construcción.
Cada ejecución informa de decenas de cambios sin que nada haya cambiado	Tareas con órdenes arbitrarias que se ejecutan siempre	Usa módulos idempotentes o declara la condición de omisión; exige cero cambios en la segunda pasada.
Aparecen picos de latencia a horas exactas	Una tarea no idempotente reinicia un servicio en cada ejecución programada	La comprobación de la segunda pasada las localiza; conviértelas en idempotentes o usa manejadores.
Un secreto sigue dentro de la imagen aunque la construcción lo borre	La imagen se captura del disco, no de capas: el fichero borrado al final ya estaba	Limpia antes de la captura, comprueba la imagen construida y rota lo expuesto.
Una plantilla de arranque no puede crear instancias	La imagen que referencia se retiró por una política de retención por fecha	Define la retención por referencias: no se borra ninguna imagen que alguna plantilla activa use.

■ 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 criterio decide entre imagen dorada y convergencia, y de qué clase viene?
2. ¿Por qué una flota que instala al arrancar falla justo cuando más se necesita escalar?
3. ¿Por qué la idempotencia aquí es una propiedad de cada tarea y cómo se comprueba?
4. ¿Qué limitación tiene la simulación de cambios frente a un plan de infraestructura?
5. ¿Qué tres casos justifican mantener convergencia, y qué impide que la lista vuelva a crecer?

Referencias

- HashiCorp (2025). Packer: building machine images — construcción, aprovisionadores y publicación. <https://developer.hashicorp.com/packer/docs>
- Ansible (2025). Desired state and idempotency — idempotencia por módulo y condiciones de omisión. <https://docs.ansible.com/ansible/latest/playbookguide/playbooksintro.html>
- Ansible (2025). Check mode and diff mode — simulación de cambios y sus límites. <https://docs.ansible.com/ansible/latest/playbookguide/playbookscheckmode.html>
- Ansible (2025). Variable precedence — los más de veinte niveles y su orden. <https://docs.ansible.com/ansible/latest/playbookguide/playbooksvariables.html#variable-precedence-where-should-i-put-a-variable>
- Ansible (2025). Rolling update strategies — ejecución por tandas y detención ante fallo. <https://docs.ansible.com/ansible/latest/playbookguide/playbooksstrategies.html>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Anterior	Índice	Siguiente
← 093 · CloudFormation, Bicep, Pulumi y Terraform	Parte 07 · Programa	095 · Plantillas, golden paths y catálogo interno →

Evaluación — 094 Ansible e imagen dorada para configuración

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 configuracion-servidor 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.

095 — Plantillas, golden paths y catálogo interno

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: platform · Estado: EXECUTABLECORE

Propósito

Poner todo lo de esta parte a disposición de equipos que no la han estudiado, que es el problema que separa una plataforma útil de un conjunto de herramientas bien hechas. La pieza central es el camino asfaltado: la forma soportada de hacer lo habitual, que no gana por ser obligatoria sino por ser más fácil que hacerlo a mano. La clase fija ese criterio, mide la adopción en vez de suponerla, y trata los dos fracasos simétricos de todo equipo de plataforma: el camino que nadie usa y el camino que se convierte en un peaje.

Resultados de aprendizaje

Al finalizar podrás:

1. Definir un camino asfaltado por el caso mayoritario, con una vía de escape documentada.
2. Medir la adopción y el tiempo hasta el primer despliegue, en vez de suponerlos.
3. Actualizar decenas de servicios creados desde una plantilla sin tocarlos uno a uno.
4. Registrar la propiedad de cada componente, que es la pregunta que este programa lleva ocho partes haciendo.
5. Reconocer los dos fracasos de un equipo de plataforma y la señal que los delata.

Conceptos centrales

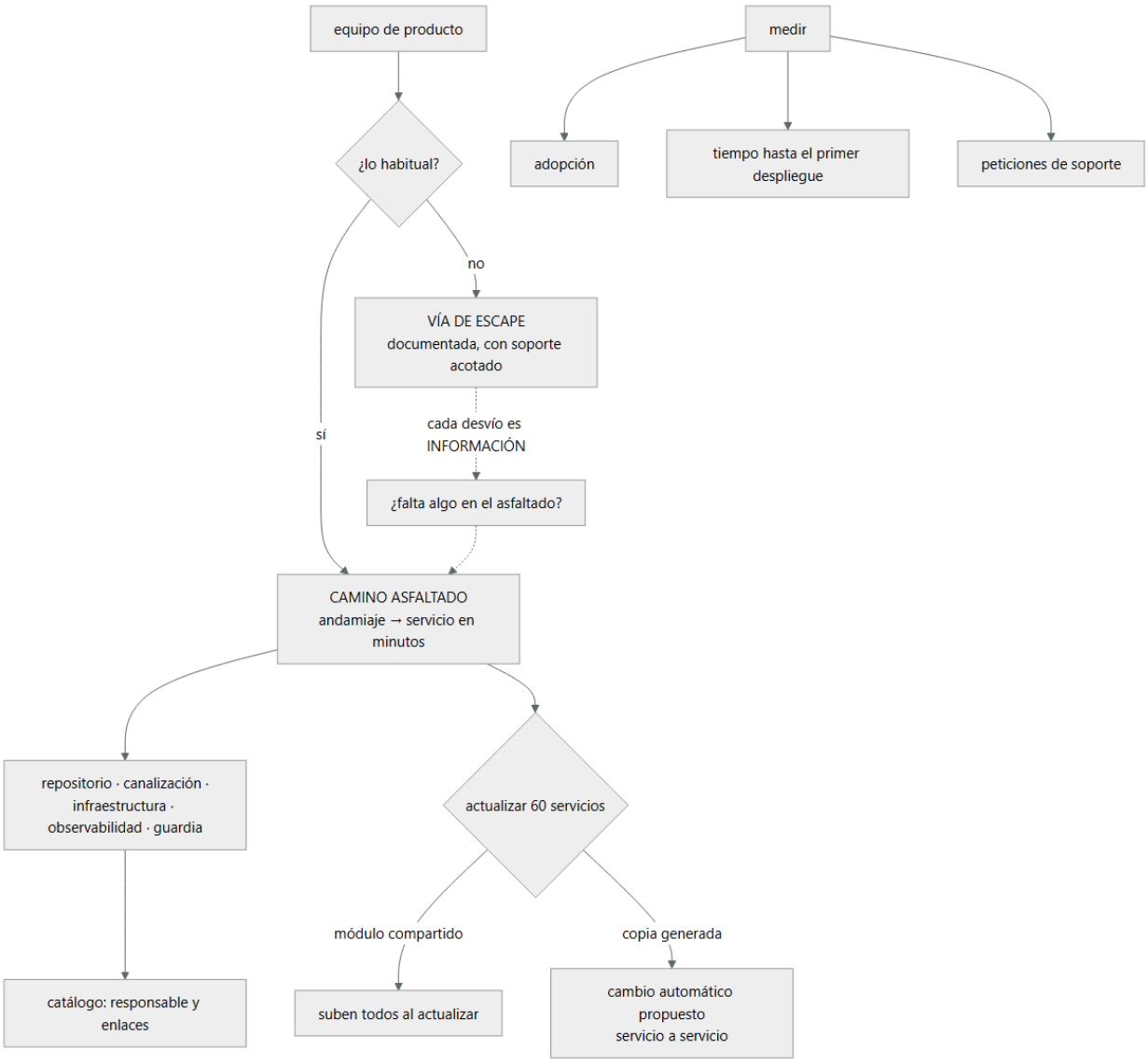
Concepto	Comprensión verificable
camino asfaltado	La forma soportada de hacer lo habitual: rápida, documentada y con las decisiones ya tomadas. No es obligatoria: gana porque es más cómoda.
andamiaje	Generador que produce un servicio funcionando —repositorio, canalización, infraestructura, observabilidad y guardia— en minutos.
vía de escape	Camino documentado para el caso que el asfaltado no cubre. Sin ella, el camino se convierte en un peaje; con ella, los desvíos son información.
desviación de plantilla	Distancia entre lo que generó la plantilla y lo que el servicio tiene hoy. Sin mecanismo de actualización, cada servicio envejece por su cuenta.
catálogo interno	Inventario de servicios con su responsable, su repositorio, sus dependencias y sus enlaces de operación. Es la respuesta a «¿de quién es esto?».
adopción	Proporción de servicios que usan el camino asfaltado. Es la única medida honesta de si la plataforma sirve.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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. Asfaltado, no obligatorio

Un equipo de plataforma que ha construido lo de las clases 085 a 094 tiene módulos versionados, políticas, canalizaciones e imágenes probadas. Y aun así, un equipo de producto puede tardar días en poner en marcha un servicio nuevo. La diferencia no es de capacidad: es de acceso.

El camino asfaltado es la forma soportada de hacer lo habitual, y su definición tiene dos mitades que hay que respetar a la vez:

```
es la forma RÁPIDA        con las decisiones ya tomadas y probadas
no es la ÚNICA forma      hay una vía de escape documentada
```

La segunda mitad es la que casi siempre se incumple, y su ausencia produce el segundo fracaso de esta clase. Un camino obligatorio deja de ser un camino y pasa a ser un peaje: los equipos lo rodean, y la plataforma pierde tanto la adopción como la información sobre por qué no encajaba.

Y el criterio para decidir qué va dentro:

```
el caso mayoritario, no todos los casos
  → si cubre el 80 %, se usa
  → si intenta cubrir el 100 %, vuelve al problema de la clase 088:
    decenas de opciones y ninguna probada
```

Lo que un camino asfaltado de servicios suele incluir, y de dónde viene cada pieza:

```
repositorio con estructura y propietarios              esta clase
canalización: construir una vez y promover             062
imagen por huella, firmada, con inventario             061 · 067
manifiestos o plantillas desde módulos versionados     081 · 088
política sobre el resultado                            091
secretos por identidad, sin claves                     092
observabilidad: métricas, registros y trazas           057 · 082
SLO inicial y alerta de presupuesto de error           057
presupuesto de interrupción y comprobaciones           068 · 079
entrada en el catálogo, con responsable                esta clase
rotación de guardia y enlace al manual de operación    045 · 057
```

Once piezas que un equipo de producto no debería tener que montar, y que además codifican decisiones que este programa ha ido pagando incidente a incidente.

Y la prueba de si el camino está de verdad asfaltado es una sola pregunta, con respuesta medible:

```
¿cuánto tarda una persona nueva en tener un servicio
 desplegado en preproducción, sirviendo tráfico y con guardia?
```

Si la respuesta se mide en días, no hay camino asfaltado: hay documentación.

2. El andamiaje y lo que produce

El generador tiene que producir algo que funciona, no un esqueleto que hay que completar:

```
$ cls nuevo-servicio
? Nombre del servicio           precios
? Equipo responsable            pedidos
? Lenguaje                      go
? Tipo                          API HTTP
? ¿Necesita base de datos?      sí, PostgreSQL pequeña
? Nivel de criticidad           2 (interno, horario laboral)

✓ repositorio creado con protecciones de rama y propietarios
✓ canalización configurada, con identidad federada
✓ infraestructura declarada, desde módulos v2.3.0
✓ manifiestos generados, con presupuesto y comprobaciones
✓ SLO inicial 99,5 % y alerta de consumo de presupuesto
✓ entrada en el catálogo, con responsable y enlaces
✓ canal de guardia y rotación asignada
✓ primer despliegue a desarrollo: correcto

URL: https://precios.dev.cloudshop.example    (2 min 40 s)
```

La última línea es la que importa: el servicio existe y responde al terminar el generador. Un andamiaje que produce ficheros y deja el despliegue como ejercicio no cambia el tiempo hasta el primer despliegue, que es la métrica que se busca.

Y tres decisiones de diseño que deciden si se usa:

Preguntar poco. Cada pregunta es una decisión que el equipo de producto no debería tener que tomar. Seis preguntas es razonable; veinte es la clase 088 otra vez.

Elegir por criticidad, no por opciones técnicas. La pregunta correcta no es «¿cuántas réplicas?» sino «¿qué pasa si esto se cae?». La respuesta determina réplicas, presupuesto, SLO, guardia y alertas, con valores que la organización ya decidió:

```
nivel 1 · crítico       3 réplicas en 3 zonas, SLO 99,9 %, guardia 24×7
nivel 2 · interno       2 réplicas, SLO 99,5 %, guardia en horario
nivel 3 · auxiliar      1 réplica, sin SLO, sin guardia
```

Generar referencias, no copias. Es la decisión que determina si los sesenta servicios creados se pueden actualizar. La siguiente sección la trata.

Y una advertencia sobre lo que no debe hacer el andamiaje: crear infraestructura de producción sin revisión. Genera el código y abre el cambio; la aplicación sigue el flujo de la clase 091, con su plan revisado y su política. Un generador que aplica directamente en producción es un camino que salta todos los controles de esta parte.

3. Actualizar sesenta servicios

Este es el problema que aparece a los seis meses y que decide si la plataforma escala.

```
sesenta servicios creados desde la plantilla v1
la plantilla va por la v4
y los sesenta siguen en v1, cada uno con sus modificaciones
```

Eso es desviación de plantilla, y tiene dos soluciones según cómo se generó cada pieza:

Lo que es una referencia sube solo. Si el servicio consume un módulo versionado (clase 088) en vez de una copia, actualizar el módulo actualiza a todos los que suban su versión:

```
module "servicio" {
  source  = "interno.example/plataforma/servicio-web/aws"
  version = "~> 3.1"          # sube con las menores automáticamente
}
```

Por eso la regla de diseño del andamiaje: generar lo mínimo y referenciar lo máximo. Un servicio nuevo debería tener pocos ficheros propios y muchas referencias.

Lo que es una copia hay que empujarlo. Los ficheros que el generador copió —la canalización, la configuración del repositorio, un fichero de arranque— no se pueden actualizar solos. La solución es un robot que abra un cambio en cada repositorio:

```
un robot detecta que la plantilla tiene una versión nueva
abre un cambio en cada servicio con la diferencia aplicada
y el equipo lo revisa y lo fusiona cuando quiere
```

Y con la métrica correspondiente, que es la que dice si la flota envejece:

```
distribución de versiones de plantilla en los servicios
  v4  38
  v3  17
  v2   4
  v1   1     ← este lleva año y medio sin actualizarse
```

Esa distribución debería estar en un panel. Una cola larga de servicios en versiones antiguas es deuda que se paga entera el día que haya que corregir algo urgente en todos.

Y una política de soporte, como la de la clase 088:

```
las dos últimas versiones mayores tienen soporte
lo anterior se actualiza o se documenta como excepción con fecha
```

Y un caso especial que conviene anticipar: los cambios incompatibles de la plantilla. Un cambio que exige modificar el código del servicio no se puede aplicar con un robot. La forma de gestionarlo es la de la clase 088 —expandir, esperar, contraer— y con un plazo real:

```
se publica la versión nueva con ambas formas soportadas
se avisa con la fecha de retirada de la antigua
el robot propone la migración
y en la fecha, lo que no haya migrado deja de recibir soporte
```

4. El catálogo responde la pregunta de ocho partes

Este programa lleva ocho partes haciendo la misma pregunta desde ángulos distintos:

```
¿de quién es este volumen?                       clase 064
¿quién responde de esta alerta?                  clase 057
¿quién es el dueño de este campo?                clases 085 · 086
¿quién mantiene este módulo?                     clase 088
¿de quién es este proyecto que aparece facturado?  clase 049
¿quién puede autorizar este cambio?              clase 091
```

El catálogo es donde esa pregunta tiene una respuesta única y consultable:

```
# catalogo/precios.yaml — generado por el andamiaje, versionado con el servicio
apiVersion: cloudshop/v1
kind: Servicio
metadata:
  nombre: precios
  descripcion: Cálculo de precios y descuentos.
spec:
  equipo: pedidos
  criticidad: 2
  repositorio: https://git.interno/pedidos/precios
  plantilla: servicio-web@v4.1.0
  depende_de: [catalogo, promociones]
```

```
  datos:
    - tipo: postgresql
      clasificacion: interna
  enlaces:
    panel: https://…
    manual: https://…
    guardia: https://…
    slo: https://…
```

Y lo que se puede hacer con eso, que es lo que justifica mantenerlo:

```
responder "¿quién responde de esto?" en segundos, a las 3 de la madrugada
calcular el impacto de retirar un servicio: quién depende de él
saber qué servicios manejan datos de una clasificación concreta
ver la distribución de versiones de plantilla
y cruzar el catálogo con la infraestructura real, como en las clases 049 y 087
```

La última es la comprobación que este programa ha usado tres veces —proyectos facturados frente a gobernados, recursos reales frente a gestionados— y aquí tiene su tercera forma:

```
# servicios en producción que no están en el catálogo
$ kubectl get deploy -A -o json | jq -r '.items[].metadata.labels["app.kubernetes.io/name"]' \
  | sort -u > desplegados.txt
$ ls catalogo/*.yaml | xargs -n1 basename | sed 's/.yaml//' | sort > catalogados.txt
$ comm -23 desplegados.txt catalogados.txt
```

Cualquier resultado es un servicio en producción del que nadie sabe quién responde.

Y la condición que hace que el catálogo no se convierta en documentación desactualizada, que es su destino habitual:

```
se genera con el servicio, no se rellena después
vive en el repositorio del servicio, no en un sistema aparte
la canalización falla si falta o si el responsable no existe
y se consulta de verdad: si nadie lo usa, nadie lo mantendrá
```

La tercera es la que lo mantiene vivo. Un catálogo cuya ausencia no rompe nada se queda vacío en seis meses.

5. Los dos fracasos, y cómo se ven

Un equipo de plataforma falla de dos formas simétricas, y las dos tienen una señal medible.

Fracaso 1: el camino que nadie usa.

```
señal      adopción baja, y equipos que montan lo suyo
causas     es más lento que hacerlo a mano
           no cubre el caso real del equipo
           nadie sabe que existe
           se rompe y no hay quien lo arregle
```

Y el diagnóstico no se hace preguntando si les gusta, sino midiendo:

```
tiempo hasta el primer despliegue, por el camino y a mano
proporción de servicios nuevos que lo usan
número de desvíos, y su motivo
```

La tercera es la más valiosa: cada desvío es información sobre lo que falta, y por eso la vía de escape tiene que existir y tiene que dejar rastro. Un equipo que se desvía en silencio no aporta nada; uno que se desvía declarando el motivo está haciendo el trabajo de producto de la plataforma.

Fracaso 2: el camino que se convierte en peaje.

```
señal      el equipo de plataforma aparece en la ruta crítica de otros
           tiempos de espera para cosas que deberían ser automáticas
           una cola de peticiones que crece
causas     el camino es obligatorio y no cubre todo
           hay pasos manuales que solo la plataforma puede hacer
           los permisos están tan cerrados que todo pasa por ahí
```

Y su medida es directa:

```
peticiones de soporte al mes, y su tipo
  las repetidas son automatizables: cada una es un fallo del camino
tiempo medio de espera de una petición
proporción de cambios que un equipo puede hacer sin pedir permiso
```

La última es la que define si la plataforma habilita o bloquea, y es exactamente el compromiso que la clase 051 negoció con la VPC compartida: el equipo de red gobierna y no es un cuello de botella, siempre que el reparto de permisos esté bien hecho.

Y las métricas que conviene tener, que son de producto y no de infraestructura:

```
tiempo hasta el primer despliegue de un servicio nuevo
adopción del camino asfaltado
distribución de versiones de plantilla
peticiones de soporte por mes y por tipo
proporción de servicios con responsable en el catálogo
servicios en producción que no están en el catálogo
```

Seis números que caben en un panel y que responden a si la plataforma sirve. Y una advertencia sobre cómo se leen: una adopción del 100 % es sospechosa. O el camino cubre de verdad todos los casos —improbable— o es obligatorio, que es el fracaso 2.

Y la lista de comprobación de la clase:

- el camino asfaltado produce un servicio DESPLEGADO, no ficheros
- pocas preguntas, y por criticidad, no por opciones técnicas
- vía de escape documentada, con soporte acotado y registro del motivo
- genera lo mínimo y referencia lo máximo, para poder actualizar
- robot que propone la actualización de lo copiado
- política de soporte de versiones de plantilla
- catálogo generado con el servicio, y su ausencia rompe la canalización
- comprobación de servicios desplegados que no están en el catálogo
- las seis métricas de producto, en un panel
- ninguna aplicación en producción sin pasar por el flujo de revisión

Ejemplo trabajado

El equipo de plataforma de CloudShop lleva ocho meses construyendo lo de las clases 085 a 094. Un servicio nuevo sigue tardando dos días en llegar a preproducción, y tres equipos han montado su propia infraestructura por su cuenta. La medición explica por qué.

Lo que se midió antes de tocar nada.

```
tiempo hasta el primer despliegue de un servicio nuevo
  por el camino de la plataforma           2 días
  a mano, copiando de otro servicio        4 horas
servicios nuevos del último trimestre       11
  que usaron el camino                       3
servicios en producción sin responsable conocido   9
peticiones de soporte al mes                ~35
```

La primera cifra explica la segunda: el camino de la plataforma era ocho veces más lento que copiar de otro servicio. Nadie iba a usarlo por convicción.

Y el desglose de las dos jornadas:

```
crear el repositorio y configurarlo          20 min
escribir la infraestructura desde módulos     3 h
escribir los manifiestos                      2 h
configurar la canalización                    2 h
pedir la identidad federada al equipo de plataforma   1 día de ESPERA
configurar observabilidad y alertas           2 h
registrar la guardia                          30 min
```

La espera de un día es el fracaso 2 en estado puro: una petición manual que solo la plataforma podía atender, en la ruta crítica de todos los servicios nuevos.

Lo que se hizo, por orden de impacto.

```
1. automatizar la identidad federada          quita 1 día de espera
2. andamiaje que genera y despliega           quita 7 h de trabajo
3. módulos referenciados en vez de copiados   permite actualizar después
4. catálogo generado, con responsable         responde "¿de quién es esto?"
```

El primero fue el más barato y el que más valió: una política que crea la confianza federada al crear el repositorio, acotada al espacio y a la cuenta como exige la clase 083.

El resultado.

```
tiempo hasta el primer despliegue          2 días          38 minutos
preguntas del generador                       —               6
ficheros propios de un servicio nuevo        41               7
referencias a módulos versionados             2               9
servicios nuevos que usan el camino        3 de 11         14 de 15
peticiones de soporte al mes                 ~35              ~9
```

Y la última fila desglosada, que es lo que dice si el trabajo sirvió:

```
peticiones que desaparecieron             tipo
  "necesito identidad federada"           automatizada
  "cómo configuro las alertas"            en la plantilla
  "qué módulo uso para X"                 el generador lo elige
  "por qué falla mi canalización"         la plantilla ya funciona
```

El servicio decimoquinto, que no usó el camino.

Un equipo necesitaba un servicio con procesamiento de vídeo, con aceleradores y almacenamiento local rápido. El camino asfaltado no lo cubría, y usó la vía de escape declarando el motivo.

```
lo que faltaba en la plantilla
  tipo de máquina con acelerador
  almacenamiento local efímero de alto rendimiento
  comprobación de estado con arranque de 4 minutos
```

Y la decisión que se tomó con ese dato:

```
no se añadió al camino asfaltado
  → es un caso único; añadirlo habría llevado el generador
    hacia el problema de la clase 088
se documentó como perfil aparte, con sus módulos propios
  → soporte acotado y por escrito
y la comprobación de estado con arranque largo SÍ se incorporó
  → resultaba útil para otros tres servicios
```

Ese desvío fue la información más valiosa del trimestre, y solo existió porque la vía de escape estaba documentada y dejaba rastro.

Los nueve servicios sin responsable.

```
$ comm -23 desplegados.txt catalogados.txt
informes-legado
sincronizador-precios
exportador-fiscal
… (9)
```

De los nueve: cuatro tenían responsable identificable preguntando, tres pertenecían a un equipo que ya no existía y dos nadie supo de quién eran. Los dos se estudiaron: uno estaba sin tráfico desde hacía ocho meses y se retiró; el otro procesaba un fichero diario que finanzas seguía usando.

```
servicios sin responsable                    9               0
servicios retirados por estar sin uso        —               1
servicios adoptados por un equipo            —               8
comprobación de catálogo en la canalización  no había        falla si falta
```

Y la distribución de versiones a los seis meses.

```
v4  21
v3   6
v2   2
v1   0
```

Con el robot proponiendo actualizaciones, ningún servicio se quedó en la primera versión. Los dos de la v2 tienen una excepción documentada con fecha.

Resumen:

```
tiempo hasta el primer despliegue          2 días          38 min
adopción del camino asfaltado              3 de 11         14 de 15
peticiones de soporte al mes                 ~35              ~9
esperas manuales en la ruta crítica           1               0
servicios sin responsable                     9               0
ficheros propios por servicio                41               7
desvíos declarados                            0               1, útil
```

La lección que esta clase traslada al proyecto de la clase 096: la plataforma tenía construido todo lo necesario desde hacía ocho meses y la adopción era del 27 %, porque su camino era ocho veces más lento que copiar de otro servicio. El cambio que más valió no fue técnico sino quitar una espera manual de la ruta crítica. Y el único desvío del trimestre fue la información más útil que recibió el equipo — lo que confirma que la vía de escape no es una concesión: es el mecanismo por el que la plataforma se entera de lo que le falta.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/095-plantillas-golden-paths-y-catalogo-interno/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 plantilla-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 plantilla-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
La plataforma tiene todo construido y los equipos montan lo suyo	El camino asfaltado es más lento que copiar de otro servicio	Mide el tiempo hasta el primer despliegue por ambas vías; si la tuya no gana, no se usará por convicción.
Un equipo de plataforma aparece en la ruta crítica de todos los servicios nuevos	Hay pasos manuales que solo esa plataforma puede hacer	Automatiza cada petición repetida: cada una es un fallo del camino, no una tarea de soporte.
Sesenta servicios creados desde una plantilla siguen en su primera versión	El generador copió ficheros en vez de referenciar módulos, y nadie empuja actualizaciones	Genera lo mínimo y referencia lo máximo; para lo copiado, un robot que proponga el cambio, con política de soporte.
El camino asfaltado intenta cubrir todos los casos y nadie sabe qué combinaciones funcionan	Es el problema de la clase 088 en el generador	Cubre el caso mayoritario y documenta una vía de escape; los desvíos dicen qué falta.
Hay servicios en producción de los que nadie sabe quién responde	El catálogo se rellena después, o su ausencia no rompe nada	Genéralo con el servicio, versiónalo con él y haz que la canalización falle si falta o si el responsable no existe.
La adopción del camino es del cien por cien	O cubre todos los casos, lo que es improbable, o es obligatorio	Comprueba si existe vía de escape y si alguien la ha usado; un camino sin desvíos es un peaje.

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. ¿Por qué un camino asfaltado no debe ser obligatorio, y qué se pierde si lo es?
2. ¿Qué diferencia hay entre un generador que produce ficheros y uno que produce un servicio desplegado?
3. ¿Cómo se actualizan sesenta servicios creados desde una plantilla, según cómo se generó cada pieza?
4. ¿Qué pregunta de ocho partes anteriores responde el catálogo, y qué lo mantiene vivo?
5. Enumera los dos fracasos de un equipo de plataforma y la señal medible de cada uno.

Referencias

- Evan Bottcher (2018). What I talk about when I talk about platforms — la plataforma como producto y el camino asfaltado. <https://martinfowler.com/articles/talk-about-platforms.html>
- Matthew Skelton, Manuel Pais (2019). Team Topologies, cap. 5 — equipos de plataforma y modos de interacción. <https://teamtopologies.com/book>
- Backstage (2025). Software catalog and scaffolder — catálogo de servicios, propiedad y plantillas. <https://backstage.io/docs/features/software-catalog/>
- Google (2023). DORA: platform engineering and developer experience — métricas de adopción y de tiempo de entrega. <https://dora.dev/research/>
- CNCF (2025). Platforms White Paper — capacidades esperadas de una plataforma interna y sus señales. <https://tag-app-delivery.cncf.io/whitepapers/platforms/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 07 en PDF · Manual integral

Evaluación — 095 Plantillas, golden paths y catálogo interno

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 plantilla-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.

096 — Proyecto: infraestructura multiambiente promovible

Parte: 07 — Infraestructura como código y configuración Nivel: intermedio-avanzado · Horas estimadas: 8 Laboratorio: capstone · Estado: EXECUTABLECORE

Propósito

Integrar las once clases anteriores en una infraestructura que se promueve entre cuatro entornos con el mismo código, y calificar la hipótesis de la clase 084. Acertó en dos afirmaciones y su primera parte partía de un supuesto que esta parte no cumplió: predijo lo que haría un bucle continuo, y lo que se construyó fue detección programada. El resultado es más interesante que el acierto: la detección sola, sin corregir nada, consiguió la mayor parte del efecto — y eso reordena lo que la parte 08 tiene que aportar.

Resultados de aprendizaje

Al finalizar podrás:

1. Trazar cada decisión de la infraestructura hasta la clase que la tomó.
2. Calificar la hipótesis de la clase 084, incluida la parte que partía de un supuesto no cumplido.
3. Enunciar las dos leyes que esta parte añade, con sus apariciones.
4. Provocar tres fallos propios de infraestructura como código y medir la recuperación.
5. Entregar cuatro entornos cuya diferencia cabe en un fichero legible.

Conceptos centrales

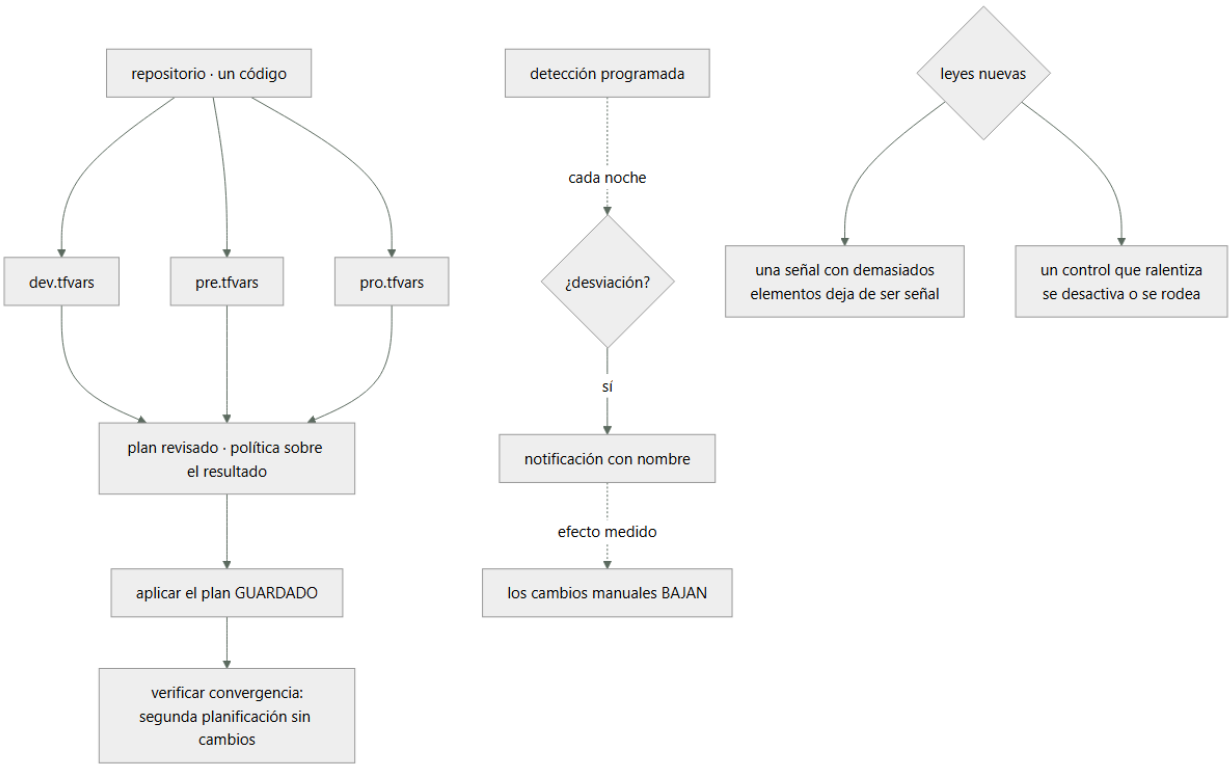
Concepto	Comprensión verificable
detección sin corrección	Comprobación periódica que informa de la desviación y no la arregla. Consigue la mayor parte del efecto de un bucle por su efecto sobre el comportamiento, no sobre los recursos.
frontera de propiedad	Acuerdo sobre qué sistema manda en cada campo. Apareció tres veces en esta parte con tres mecanismos, y su ausencia produce oscilación permanente.
señal con demasiados elementos	Previsualización, informe o alerta con tantas entradas que nadie la lee. Su corrección es eliminar lo que nadie consulta, no filtrar al leer.
control que ralentiza	Comprobación o camino que hace el trabajo más lento. Se desactiva o se rodea, con independencia de lo correcto que sea.
promoción	Llevar el mismo código a entornos sucesivos cambiando solo un fichero de valores. Lo que no cabe ahí es erosión o es una decisión de arquitectura mal expresada.
recuperación del estado	Volver a gestionar la infraestructura tras perder el fichero. Cuatro minutos con versionado, días reconstruyendo: es la diferencia que justifica todo lo de la clase 087.

Modelo mental

IaC trata la infraestructura como producto versionado: el plan explica intención, el estado conecta intención y realidad, y la revisión reduce cambios accidentales.

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 infraestructura entregada y de dónde sale cada decisión

Doce decisiones con su alternativa descartada y la trampa que evita:

Decisión	Requisito	Alternativa descartada	Trampa que evita
Clasificar la desviación por origen	Revisión legible (085)	Tratar todo como manual	84 % de ruido que hace ilegible el plan
Dependencias implícitas por referencia	Orden correcto (086)	Declararlas por si acaso	Ejecuciones que fallan 2 de cada 5 veces
Campos de otros sistemas, ignorados	Frontera de propiedad (086)	Declararlos	Oscilación permanente en cada aplicación
Un estado por radio de impacto	Bloqueo y alcance (087)	Un estado para todo	12 esperas al día y destroy global
Recuperación del estado ensayada	Continuidad (087)	Confiar en el versionado	4 minutos frente a 4 días
Módulos que codifican decisiones	Reutilización real (088)	Envolturas con 47 variables	Tres equipos escribiendo lo suyo
Fichero de valores explícito siempre	Promoción segura (089)	Carga automática	Aplicar producción sobre preproducción
Comprobación de cuenta esperada	Última defensa (089)	Confiar en el fichero	Lo mismo, pero sin darse cuenta
Detección de desviación programada	Ver lo que cambia (090)	Descubrirlo en un incidente	21 días con un puerto abierto
Política sobre el plan, no el código	Ver los valores (091)	Análisis estático	2 años con una base de datos pública
Secretos que no existen para la plantilla	Sin rastros (092)	Marcar como sensible	14 campos en claro en los estados
Imagen dorada para lo que es ganado	Sin dependencias al arrancar (094)	Configurar al arrancar	24 minutos sin poder escalar

Y tres decisiones tomadas en contra de lo que sugiere el hábito:

1. No se consolidó en una sola herramienta.
   Dos, cada una donde su modelo operativo encaja, después de medir que

```
   18 de 24 incidentes no dependían de la herramienta (093).

2. El camino asfaltado NO es obligatorio, y su único desvío del trimestre
   fue la información más útil que recibió la plataforma (095).

3. Los dos valores aleatorios que quedan en el estado se aceptan como riesgo
   declarado, en vez de forzar una solución que complicaría el sistema (092).
```

2. La hipótesis de la clase 084, calificada

La clase 084 dejó escrito:

> El bucle continuo eliminará la desviación entre repositorio y realidad y hará visible la frontera de propiedad entre quienes escriben sobre el mismo objeto. Y volverá a aparecer la ley 13, porque un bucle de infraestructura detenido tampoco produce ningún error.

Afirmación 1 — el bucle eliminará la desviación. NO SE PUEDE CALIFICAR: el supuesto no se cumplió.

Esta parte no construyó un bucle continuo. Construyó detección programada (clase 090): una planificación nocturna que informa y no corrige. Y ese cambio de supuesto produce el resultado más interesante de la parte:

```
lo que se esperaba de un bucle    corregir la desviación automáticamente
lo que hizo la detección          informar, con nombre, al día siguiente

efecto medido sobre los recursos  ninguno: no corrige nada
efecto medido sobre las personas  los cambios manuales bajaron de 9 a 2 al mes
```

La segunda línea es el hallazgo. La mayor parte del beneficio esperado del bucle se obtuvo sin bucle, porque el problema no era técnico: era que el cambio manual funcionaba y no dejaba rastro. En cuanto dejó rastro, dejó de ser un método cómodo.

Y eso reordena lo que la parte 08 tiene que aportar: no es «hacer visible la desviación» —eso ya está— sino corregirla sola y sostener la corrección, que es un problema distinto y con riesgos propios.

Afirmación 2 — la frontera de propiedad se hará visible. CIERTA, y apareció tres veces.

```
réplicas entre manifiesto y escalado automático           081
campos que gestiona otro sistema, como origen 3           085
capacidad deseada que volvía a tres cada noche            086
```

Tres mecanismos distintos y el mismo conflicto: dos sistemas creyéndose dueños del mismo campo. Y la corrección fue siempre la misma —un solo dueño por campo, declarado— lo que la convierte en una regla de diseño y no en un ajuste.

Afirmación 3 — reaparecerá la ley 13. CIERTA.

```
el analizador de seguridad, desactivado 14 meses              091
el escaneo de secretos que solo miraba el repositorio         092
y la propia detección: si deja de ejecutarse, no avisa nadie  090
```

La tercera es la más literal: una comprobación que lleva tres semanas sin ejecutarse es indistinguible de un sistema sin desviación. Y su antídoto es el que la ley 13 pide siempre: una señal de que la comprobación se ejecutó.

3. Dos leyes nuevas, con sus apariciones

Ley 15. Una señal con demasiados elementos deja de ser una señal, y su corrección es eliminar lo que nadie consulta.

Seis apariciones, en seis partes distintas y con seis mecanismos:

```
214 líneas de previsualización, con el cambio real dentro        047
340 alertas al mes, 6 con impacto real                           057
812 hallazgos de seguridad, 24 accionables                       067 · 091
105 cambios propuestos, 10 de desviación real                    085 · 090
diferencial de manifiestos con ruido de campos calculados        081
62 GiB/día de registros, el 78 % nunca consultado                057 · 082
```

Y lo que las une no es el volumen sino la corrección, que es la misma en las seis y siempre contraintuitiva:

```
la respuesta NO es filtrar al leer, ni acostumbrarse
la respuesta es ELIMINAR lo que nadie consulta
  → declarar los campos que el proveedor rellena
  → bloquear solo por lo corregible
  → alertar sobre la experiencia, no sobre umbrales
  → excluir antes de ingerir
```

Y la razón por la que hay que eliminarlo en vez de tolerarlo: una señal ruidosa entrena a no mirar, y el día que aparece algo real pasa desapercibido. En esta parte ocurrió literalmente dos veces: una base de datos pública durante dos años y un puerto abierto durante tres semanas.

Ley 16. Un control o un camino que hace el trabajo más lento se desactiva o se rodea, con independencia de lo correcto que sea.

Cinco apariciones:

```
el escáner de imágenes, desactivado ocho meses                  067
el analizador de infraestructura, desactivado catorce meses     091
las alertas silenciadas: 46 de 340                              057
los módulos que nadie usaba: tres equipos escribiendo lo suyo   088
el camino asfaltado con 27 % de adopción, ocho veces más lento  095
```

Y su consecuencia de diseño, que es lo que convierte la ley en algo accionable:

```
un control se diseña por su COSTE, no solo por su corrección
  bajo dos minutos, se ejecuta siempre
  a diez, cuando alguien se acuerda
  a treinta, se desactiva

y un camino se adopta si es MÁS FÁCIL, no si es el correcto
```

Las dos leyes están relacionadas y no son la misma: la 15 es sobre la señal que produce un control y la 16 sobre su coste. Un control puede ser rápido y ruidoso —el analizador de la clase 091— o lento y preciso. Los dos se acaban desactivando, por motivos distintos.

4. Tres fallos provocados y lo que enseñó cada uno

Fallo 1 — perder el estado de un entorno. Se borra el estado de preproducción.

```
recuperación desde una versión anterior          4 min 10 s
verificación con plan sin cambios                correcta
infraestructura afectada                         ninguna
```

Y el hallazgo: el ensayo se hizo también sobre el estado de la plataforma, y ahí falló.

```
el bucket del estado de plataforma tenía versionado
y la copia diaria a otra cuenta NO lo incluía: se había añadido
después de configurar la copia, y nadie revisó la lista
```

La recuperación funcionó porque el versionado estaba; si el borrado hubiera sido del bucket, ese estado no tenía segunda copia.

```
corrección   la lista de la copia se genera del inventario, no se mantiene a mano
             y el ensayo cubre los cuatro estados, no uno
```

Fallo 2 — aplicar con el fichero de valores equivocado. Se intenta a propósito aplicar producción sobre preproducción.

```
la comprobación de cuenta esperada detiene la planificación     ✓
mensaje                                                          claro
tiempo hasta el fallo                                            8 s
```

Y el hallazgo: funcionó en tres de los cuatro estados. En el cuarto, la comprobación existía y la cuenta era la misma para dos entornos, porque preproducción y desarrollo compartían cuenta.

```
corrección   la comprobación pasa a validar cuenta Y prefijo de nombres
             y se añade una segunda: el estado en uso debe contener el entorno
```

Es el mismo patrón de la clase 080 con las políticas de red: una comprobación que existe y no cubre lo que se cree. Séptima aparición de la familia de fallos de la clase 060, ahora en la propia comprobación.

Fallo 3 — detener la detección de desviación. Se desactiva la ejecución nocturna sin avisar.

```
tiempo hasta que alguien lo notó                 no se notó en 3 semanas
lo que se detectó durante ese tiempo             nada
desviación real acumulada en esas 3 semanas      2 cambios manuales
```

Y el hallazgo: exactamente lo que la ley 13 predice. La detección no avisa de su propia ausencia, y tres semanas de silencio son indistinguibles de tres semanas sin desviación.

```
corrección   métrica de "última ejecución con éxito" por estado
             alerta si envejece más de 36 horas
             y comprobación de que esa alerta funciona, con un simulacro
```

Los tres hallazgos comparten forma con los de las clases 048, 060, 072 y 084: el mecanismo funcionó y había algo que ninguna revisión de configuración podía mostrar — una lista de copias mantenida a mano, dos entornos compartiendo cuenta y una comprobación que no vigilaba su propia ejecución.

5. La entrega y la pregunta que abre la parte 08

La entrega, sin conocimiento tácito.

```
código              un repositorio, cuatro ficheros de valores
módulos             6, versionados y probados con creación real
estados             4, por radio de impacto, con recuperación ensayada
línea base          24 afirmaciones, cada una con su comprobación
verificar.sh        ejecuta las 24 y devuelve código de salida
política            19 reglas sobre el plan, en las dos herramientas
ADR                 12 decisiones con su alternativa descartada
riesgos residuales  4, con responsable y condición de revisión
camino asfaltado    servicio nuevo desplegado en 38 minutos
catálogo            todos los servicios con responsable
```

Los cuatro riesgos residuales:

```
1. dos valores aleatorios permanecen en el estado, con acceso mínimo
2. dos servidores heredados siguen gestionados por convergencia (094)
3. la aplicación no es automática al fusionar: requiere aprobación humana
4. la detección informa y no corrige: la desviación vive hasta que alguien actúa
```

El cuarto es precisamente lo que la parte 08 tiene que resolver.

La comparación con el punto de partida:

```
cambios propuestos en ejecución limpia          140             0
recursos huérfanos                               34             0
esperas por bloqueo al día                      ~12             0
diferencias accidentales entre entornos          31             0
recursos solo en producción                      31             4
campos sensibles en los estados                  14             2
secretos en rastros no auditados            4 rastros           0
tiempo hasta el primer despliegue             2 días        38 min
cambios manuales al mes                           9             2
tiempo de arranque de una máquina          6 min 20 s        38 s
costo no atribuible                      2.180 USD/mes         0
pruebas negativas ejecutadas                 0 de 24       24 de 24
```

Y la pregunta que abre la parte 08.

Esta parte deja una infraestructura descrita, revisada, verificada y vigilada, y con un hueco declarado como riesgo: la desviación se detecta y no se corrige, y la aplicación la dispara una persona. El mecanismo que cierra ese hueco ya está descrito desde la clase 073 — un bucle que reconcilia — y la pregunta correcta no es si funciona, sino qué cuesta:

> Si un bucle reconcilia la infraestructura continuamente, se convierte en el actor más privilegiado del sistema: aplica sin supervisión, tiene permisos sobre todo y su compromiso equivale al de la plataforma entera. ¿Qué hay que darle, qué hay que negarle, y qué NO debe tocar nunca?

La hipótesis que se escribe ahora, para poder equivocarse de forma comprobable:

> El bucle eliminará la desviación y trasladará el problema a dos sitios nuevos: la identidad de la canalización, que pasa a ser el objetivo más valioso del sistema, y la lista de lo que el bucle no debe revertir, que nadie sabrá mantener. Y aparecerá la ley 16 con un mecanismo nuevo: si el flujo automatizado es más lento que aplicar a mano, alguien aplicará a mano.

La parte 08 la califica.

Ejemplo trabajado

Entrega del capstone de la parte 07, con las cifras que se llevan a la parte 08.

Verificación completa. Las 24 afirmaciones de la línea base:

```
$ ./verificar.sh
✓ ejecución limpia sin cambios propuestos        4 estados, 0 cambios
✓ segunda aplicación seguida sin cambios         idempotencia comprobada
✓ ningún fichero de carga automática             0 encontrados
```

```
✓ comprobación de cuenta y prefijo                4 de 4 estados
✓ estados con bloqueo, versionado y cifrado      4 de 4
✓ acceso al estado: lectura y escritura separadas 2 identidades
✓ recuperación del estado ensayada                4 min 10 s
✓ ningún recurso huérfano                         0 de 412
✓ módulos consumidos por versión                  22 de 22
✓ módulos con pruebas de creación real            6 de 6
✓ política sobre el plan                          19 reglas, 2 herramientas
✓ excepciones con motivo, responsable y fecha     6 de 6
✓ ninguna excepción caducada                      0
✓ ningún secreto en el historial del repositorio  escaneo limpio
✓ campos sensibles en los estados                 2, declarados como riesgo
✓ planes no publicados como artefacto             0
✓ registro detallado desactivado                  confirmado
✓ diferencia entre entornos en un fichero         8 valores
✓ ningún condicional por entorno                  1, con riesgo declarado
✓ recursos con protección contra destrucción      41 de 41
✓ imagen probada arrancando, sin red externa      7 comprobaciones
✓ segunda pasada de convergencia sin cambios      0 cambios
✓ servicios con responsable en el catálogo        15 de 15
✓ detección de desviación con señal de ejecución  4 estados
24/24 correctas
```

Línea base medida:

```
tiempo de una planificación por estado           38 s
plan a aplicación, con revisión                  ~20 min (aprobación humana)
servicio nuevo hasta desplegado                  38 min
arranque de una máquina                          38 s
recuperación de un estado                        4 min 10 s
detección de desviación                          nocturna, < 24 h
```

Los tres fallos, con lo aprendido en cada uno:

```
estado borrado                inmediata     ninguno; recuperado   la lista de
                                            en 4 min              copias se
                                                                  mantenía a mano
                                                                  y no cubría todo

valores del entorno            8 s           ninguno en 3 de 4     una comprobación
  equivocado                                 estados               puede existir y
                                                                   no cubrir lo que
                                                                   se cree

detección detenida            no se detectó  2 cambios manuales    la comprobación
                              en 3 semanas   sin ver               no vigila su
                                                                   propia ejecución
```

El hallazgo que justificó el capstone. Al ensayar la recuperación del estado en los cuatro entornos:

```
$ aws s3api list-object-versions --bucket cls-tfstate-plataforma \
    --prefix terraform.tfstate --query 'length(Versions)'
47
$ aws s3api get-bucket-replication --bucket cls-tfstate-plataforma
An error occurred (ReplicationConfigurationNotFoundError)
```

El versionado estaba y la copia a otra cuenta no. El bucket se había creado tres meses después de configurar la copia, y la lista de buckets a copiar se mantenía a mano.

```
síntoma observable   ninguno: la recuperación por versionado funcionaba
consecuencia real    ese estado no tenía protección frente al borrado
                     del bucket ni frente a la pérdida de acceso a la cuenta
causa                una lista mantenida a mano que envejeció
qué lo destapó       ensayar en los CUATRO estados, no en uno
```

Corrección y comprobación posterior:

```
lista de buckets a copiar              mantenida a mano   generada del inventario
estados con copia fuera de la cuenta         3 de 4            4 de 4
ensayo de recuperación                     1 estado          los 4, trimestral
comprobación de cobertura de la copia      ninguna       en el guion de verificación
```

Se entrega a la parte 08 con:

```
dieciséis leyes observadas, dos de ellas nuevas de esta parte
las doce prácticas portables entre herramientas (093)
```

```
24 afirmaciones y su guion de verificación
12 decisiones con alternativa descartada
4 riesgos residuales, uno de ellos el hueco que la parte 08 debe cerrar
cuatro entornos cuya diferencia cabe en ocho valores
```

Y la hipótesis escrita para la parte 08:

> El bucle eliminará la desviación y trasladará el problema a la identidad de la canalización —que pasa a ser el objetivo más valioso— y a la lista de lo que el bucle no debe revertir. Y reaparecerá la ley 16: si el flujo automatizado es más lento que aplicar a mano, alguien aplicará a mano.

La lección que esta parte deja al programa: la hipótesis de la clase 084 predijo lo que haría un bucle, y esta parte no construyó ninguno. Construyó detección, y la detección sola consiguió la mayor parte del efecto esperado — los cambios manuales bajaron de nueve a dos al mes sin corregir un solo recurso, porque el problema no era técnico sino que el cambio manual no dejaba rastro. Ese resultado reordena lo que falta: la parte 08 no tiene que hacer visible la desviación, sino corregirla sola y sostener la corrección sin convertir el bucle en el punto único de compromiso del sistema.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-07-infrastructure-as-code-configuration/096-proyecto-infraestructura-multiambiente-promovible/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-iac 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-iac 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 copia de seguridad existe y no cubre todo lo que debería	La lista de lo que se copia se mantiene a mano y envejece	Genera la lista del inventario y ensaya la recuperación en todos los elementos, no en uno de muestra.
Una comprobación de entorno existe y no impide el error	Dos entornos comparten cuenta, así que la condición se cumple en ambos	Valida varias señales —cuenta, prefijo y estado en uso— y comprueba la protección en los cuatro entornos.
La detección de desviación lleva semanas sin ejecutarse y nadie lo sabe	Es la ley 13: la comprobación no avisa de su propia ausencia	Métrica de última ejecución con éxito, alerta de envejecimiento, y un simulacro que compruebe que esa alerta funciona.
Una previsualización con cientos de líneas hace que nadie la lea	Es la ley 15: la señal tiene demasiados elementos	Elimina lo que nadie consulta —declarando campos, bloqueando solo por lo corregible— en vez de filtrar al leer.
Un control correcto acaba desactivado o rodeado	Es la ley 16: hace el trabajo más lento	Diseña el control por su coste además de por su corrección; por encima de dos minutos, la gente busca cómo evitarlo.
La desviación se detecta y sigue ahí durante días	La detección informa y no corrige, y la corrección depende de que alguien actúe	Es un riesgo declarado de esta parte; cerrarlo exige un bucle que reconcilie, con los cuidados de la parte 08.

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. ¿Por qué la primera afirmación de la hipótesis de la clase 084 no se puede calificar, y qué resultado dio en su lugar?
2. Enuncia la ley 15 y di por qué su corrección es eliminar en vez de filtrar.
3. Enuncia la ley 16 y qué implica para el diseño de un control.
4. Al ensayar la recuperación en los cuatro estados apareció un fallo que no se veía en uno. ¿Cuál y por qué?
5. ¿Qué hueco deja esta parte declarado como riesgo, y qué pregunta abre para la parte 08?

Referencias

- Kief Morris (2020). Infrastructure as Code, 2.ª ed., cap. 20 — entornos, promoción y organización del código. <https://infrastructure-as-code.com/book/>
- HashiCorp (2025). Recommended practices for Terraform in production — estados, entornos y flujo de trabajo. <https://developer.hashicorp.com/terraform/cloud-docs/recommended-practices>
- Google (2023). DORA: change failure rate and lead time — métricas de entrega aplicadas a infraestructura. <https://dora.dev/research/>
- Nicole Forsgren et al. (2018). Accelerate, cap. 4 — efecto de la visibilidad sobre el comportamiento del equipo. <https://itrevolution.com/product/accelerate/>
- OWASP (2025). Infrastructure as Code security cheat sheet — controles y su coste de adopción. <https://cheatsheetseries.owasp.org/cheatsheets/InfrastructureasCodeSecurityCheatSheet.html>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Evaluación — 096 Proyecto: infraestructura multiambiente promovible

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-iac 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.