Multi-Cloud Engineering Program

Parte 06 — Kubernetes y plataformas administradas

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	06 de 23
Clases	073–084
Evaluaciones	12
Horas estimadas	52

Índice

Parte 06 — Kubernetes y plataformas administradas

← Parte 05 · Índice completo · Parte 07 →

Descargar: Esta parte en PDF · Manual integral

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

Operar cargas declarativas y portables sobre EKS, AKS o GKE.

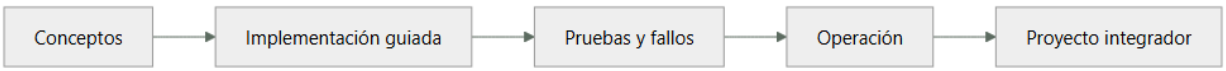
Resultados de la parte

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

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

Secuencia



Clases

ID	Clase	Laboratorio	Horas
073	API server, etcd, scheduler, controllers y kubelet	kubernetes	4
074	Pods, ReplicaSets, Deployments y Jobs	kubernetes	4
075	Services, DNS, Ingress y Gateway API	network	4
076	ConfigMaps, Secrets y configuración externa	configuration	4
077	Volumes, PersistentVolumes, CSI y StatefulSets	storage	4
078	Requests, limits, scheduling y autoscaling	capacity	4
079	Probes, rollouts, rollback y PodDisruptionBudget	reliability	4
080	Namespaces, RBAC, NetworkPolicy y admission	security	4
081	Helm, Kustomize y gestión de paquetes	configuration	4
082	Logs, métricas, eventos y depuración	observability	4
083	EKS, AKS y GKE: similitudes y diferencias	decision	4
084	Proyecto: plataforma Kubernetes portable	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

- Kubernetes: Up and Running — Burns, Beda y Hightower.
- Kubernetes Patterns — Ibryam y Huß.
- Production Kubernetes — Rosso et al..

073 — API server, etcd, scheduler, controllers y kubelet

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: kubernetes · Estado: EXECUTABLECORE

Propósito

Entender Kubernetes por su mecanismo y no por su catálogo de objetos, porque de ese mecanismo salen todos sus comportamientos característicos: es una base de datos con bucles de reconciliación alrededor. Escribir en ella devuelve «registrado», nunca «funcionando», y esa distancia entre lo aceptado y lo real es exactamente la ley nueva que la clase 072 predijo. La clase muestra el camino completo de una escritura, quién hace qué en cada bucle, y cómo se diagnostica un bucle que no avanza.

Resultados de aprendizaje

Al finalizar podrás:

1. Describir el camino de una escritura desde el cliente hasta el almacén, y qué significa exactamente la respuesta.
2. Explicar por qué la reconciliación por nivel hace robusto al sistema y a la vez consistente solo con el tiempo.
3. Atribuir un estado atascado al bucle concreto que no avanza, con la señal de cada uno.
4. Anticipar el efecto de que un controlador o un complemento de admisión deje de estar disponible.
5. Operar el almacén de estado sabiendo qué lo deja en solo lectura y qué significa su copia de seguridad.

Conceptos centrales

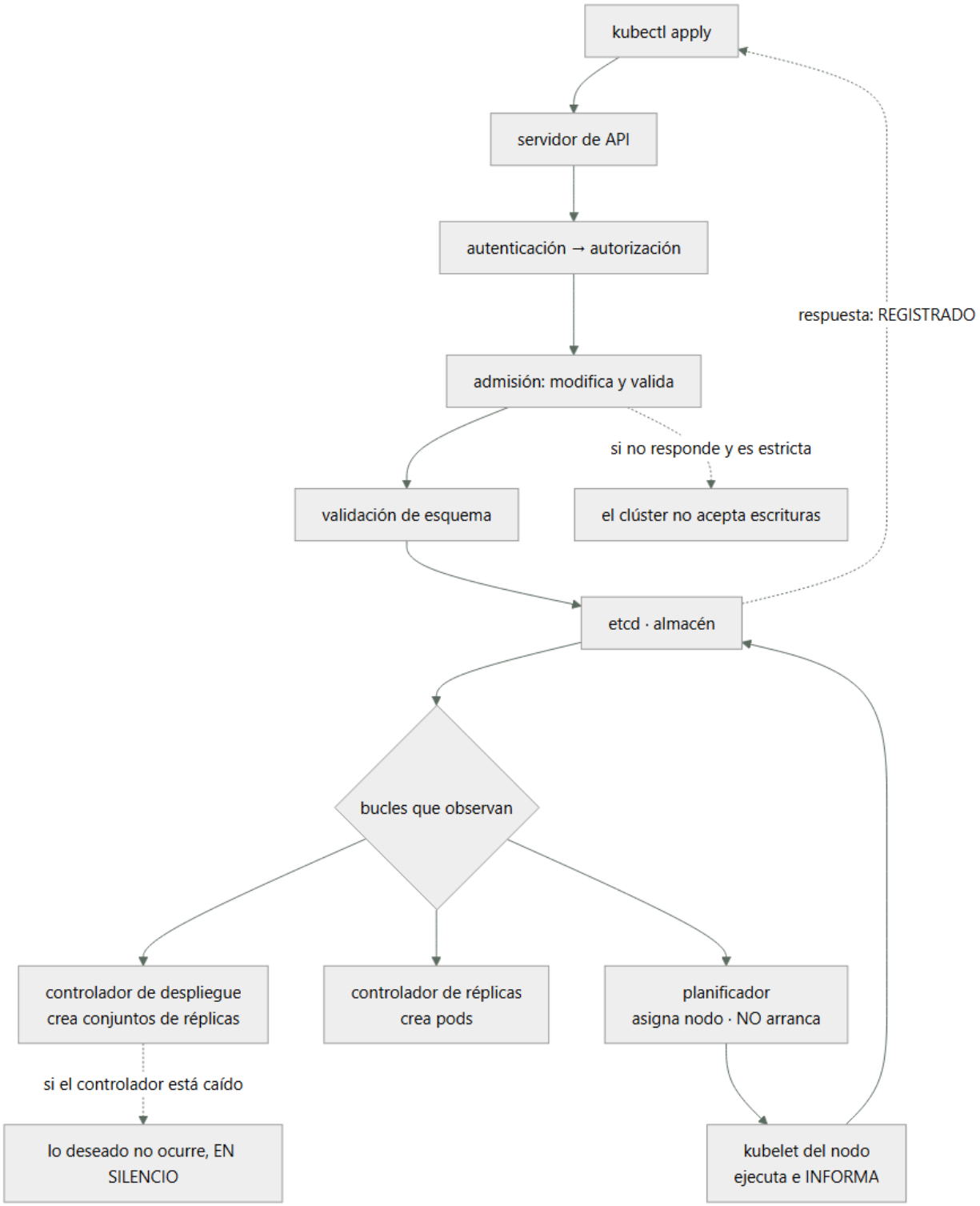
Concepto	Comprensión verificable
reconciliación por nivel	El controlador compara el estado deseado con el observado y actúa, en vez de reaccionar a sucesos. Perder un aviso no rompe nada: en la vuelta siguiente vuelve a comparar.
aceptado frente a funcionando	La respuesta de la API significa que el objeto está almacenado y es válido. No dice nada sobre si existe algo ejecutándose.
complemento de admisión	Código que puede modificar o rechazar una petición antes de guardarla. Si no está disponible y su política es estricta, el clúster deja de aceptar escrituras.
planificador	Filtra nodos, los puntúa y escribe el elegido en el objeto. No arranca nada: su trabajo termina asignando.
kubelet	El único componente que ejecuta contenedores. Observa lo asignado a su nodo, lo ejecuta y informa del estado.
estado informado	El campo de estado lo escribe quien observa, con retraso. Un nodo caído sigue figurando como sano durante decenas de segundos.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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. Una base de datos con bucles alrededor

Kubernetes se explica mal como «un orquestador de contenedores» porque esa descripción no permite predecir nada. La descripción que sí lo permite es otra:

```
un almacén de objetos con validación
+ un conjunto de bucles que comparan lo deseado con lo observado
  y actúan para acercarlos
```

Todo lo demás se deduce de ahí. Un despliegue no «lanza» contenedores: escribe un objeto, y un controlador que lo observa crea otro objeto, y otro controlador crea pods, y el planificador les asigna nodo, y el kubelet de cada nodo los ejecuta. Cinco bucles, cinco objetos, ninguna llamada directa entre componentes.

```
Deployment  →  ReplicaSet  →  Pod  →  (asignado)  →  contenedor en ejecución
  controlador   controlador    planificador           kubelet
  de despliegue de réplicas
```

Esa cadena explica por qué el diagnóstico consiste siempre en preguntar qué eslabón no avanzó.

Y la propiedad que hace robusto al sistema es que los bucles funcionan por nivel, no por suceso:

```
por suceso   "ha ocurrido X, reacciono"
             si se pierde el aviso, la acción no se hace nunca
por nivel    "esto es lo que debería haber; esto es lo que hay; actúo"
             si se pierde un aviso, la vuelta siguiente lo corrige
```

De ahí sale la resistencia característica: un controlador puede reiniciarse, perder conexión durante minutos y volver, y el sistema converge igualmente porque no depende de haber visto nada. Y de ahí sale también su contrapartida, que es la ley que esta parte tenía que comprobar: la convergencia lleva tiempo, y durante ese tiempo lo aceptado y lo real no coinciden.

La demostración cabe en dos órdenes:

```
$ kubectl apply -f despliegue.yaml
deployment.apps/tienda created                ← esto significa: ALMACENADO

$ kubectl get pods -l app=tienda
No resources found                            ← todavía no existe nada
```

La primera respuesta es correcta y no dice nada sobre contenedores. Es la diferencia que la clase 072 predijo, y merece enunciarse como norma de operación:

> Una canalización que da por bueno un despliegue porque apply devolvió éxito no ha verificado nada. Lo que verifica es esperar a la convergencia y comprobarla.

```
$ kubectl rollout status deploy/tienda --timeout=180s
$ kubectl wait --for=condition=Available deploy/tienda --timeout=180s
```

2. El camino de una escritura, y dónde se puede romper

Una petición de escritura atraviesa cinco etapas antes de existir, y cada una falla de una forma distinta:

```
1. autenticación     ¿quién eres?            → 401
2. autorización      ¿puedes hacerlo?        → 403
3. admisión que MODIFICA  añade o cambia campos
4. admisión que VALIDA    acepta o rechaza   → 400 con el motivo
5. validación de esquema y persistencia en el almacén
```

Las etapas 3 y 4 son el punto de extensión más potente del sistema y el más peligroso de operar. Ahí se enganchan las políticas de la clase 067 —verificar firma y procedencia—, la inyección de configuración y los controles de la clase 080.

Y tienen una propiedad que produce el incidente más severo de esta clase: un complemento de admisión que no responde bloquea la escritura si su política es estricta.

```
webhooks:
  - name: politica.cloudshop.example
    failurePolicy: Fail          # si no respondo, RECHAZA
    timeoutSeconds: 10
    rules:
      - operations: ["CREATE", "UPDATE"]
        resources: ["pods"]
```

Con esa configuración, si el servicio que atiende el complemento se cae, nadie puede crear pods en el clúster. Y hay una forma de que eso se convierta en un bloqueo del que no se sale solo: si el propio complemento se ejecuta como pods dentro del clúster y todos caen a la vez, no se pueden crear los pods que lo restaurarían.

Las tres protecciones, y hacen falta las tres:

```
1. excluir del alcance el espacio de nombres donde vive el complemento
   → puede recrearse a sí mismo
2. tiempo de espera corto: segundos, no decenas
3. decidir la política con criterio, no por costumbre
   Fail    para lo que de verdad no debe ocurrir sin comprobar (firma, 067)
```

```
    Ignore  para lo que aporta y no es crítico (inyección de etiquetas)
```

Y el almacén de estado merece cuatro hechos operativos, porque es el único componente cuya pérdida es irrecuperable:

```
consenso con quórum      3 o 5 miembros; con 3, se tolera la pérdida de 1
tamaño acotado           al superar su cuota, el clúster pasa a SOLO LECTURA
objetos grandes          un objeto de más de ~1 MiB no se puede guardar
la copia de seguridad ES el clúster   restaurarla devuelve todo el estado
```

El segundo produce un incidente que despista, porque el mensaje de error no menciona el almacén:

```
etcdserver: mvcc: database space exceeded
```

Y lo que se ve desde fuera es que ningún cambio se aplica, con apply devolviendo un error genérico. La causa habitual es la acumulación de revisiones históricas sin compactar, y la corrección es compactar y desfragmentar — con una alerta sobre el tamaño para no volver a llegar.

El cuarto hecho conviene decirlo con todas sus consecuencias: una copia del almacén contiene todos los secretos del clúster, en claro si no se configuró el cifrado en reposo. Es la misma propiedad del estado de Terraform de la clase 059 y del historial de despliegues de la 047 — tercera y cuarta aparición de la ley que la clase 072 enunció: lo que se guarda en un sistema central se recupera entero, incluido lo que se creía protegido.

3. Quién hace qué, y qué significa cada estado atascado

Saber qué bucle es responsable de cada transición convierte el diagnóstico en una tabla:

Estado	Quién debería actuar	Qué falta
El objeto no existe tras apply	Admisión o autorización	Mirar el error de la petición
Existe el despliegue y no hay pods	Controlador de despliegue o de réplicas	¿Está vivo el gestor de controladores?
Pending	Planificador	Ningún nodo cumple los requisitos
ContainerCreating	kubelet, red o almacenamiento	Volumen que no monta, imagen que no baja
CrashLoopBackOff	Tu aplicación	El contenedor arranca y sale
Terminating que no acaba	kubelet o finalizadores	Proceso que ignora la señal, o un finalizador atascado

Y cada fila tiene una orden que da la respuesta:

```
$ kubectl describe pod tienda-7f4c9 | sed -n '/Events/,$p'
Events:
  Warning  FailedScheduling  0/6 nodes are available:
    3 Insufficient memory, 2 node(s) had untolerated taint, 1 Insufficient cpu.
```

Eso es el planificador explicando su decisión, nodo a nodo. No hace falta adivinar: dice exactamente cuántos nodos descartó y por qué motivo cada grupo.

El planificador trabaja en dos fases y conviene conocerlas porque explican decisiones que parecen arbitrarias:

```
filtrado    descarta nodos que no pueden: recursos solicitados, marcas,
            afinidades obligatorias, puertos ocupados, volúmenes de otra zona
puntuación  ordena los que quedan: reparto, afinidad preferida,
            imágenes ya presentes en el nodo
```

La última condición del filtrado —volúmenes ligados a una zona— es la primera aparición en esta parte de la fuga de almacenamiento de la clase 064: un volumen de bloque vive en una zona, así que ata el pod a esa zona. La clase 077 lo desarrolla.

Y el kubelet tiene dos propiedades que hay que interiorizar:

```
es el único que ejecuta contenedores
  → si el kubelet de un nodo está caído, sus pods siguen ejecutándose
    y nadie los sustituye ni actualiza su estado
informa del estado con retraso
  → lo que la API dice sobre un pod es una foto de hace unos segundos
```

La primera es la que produce el desconcierto más común: un nodo cuyo kubelet murió sigue sirviendo tráfico con sus contenedores vivos, mientras el clúster lo marca como no listo. Nada se mueve automáticamente hasta que se cumplen los plazos del apartado siguiente.

Y el gestor de controladores merece una advertencia porque su fallo es silencioso, en el sentido exacto de la ley de la clase 060: si el controlador de despliegues no está funcionando, apply sigue devolviendo éxito y no ocurre nada. Ningún error, ninguna alerta por defecto. La comprobación es directa y debería estar en el panel:

```
$ kubectl get --raw '/readyz?verbose' | grep -v ok$
$ kubectl -n kube-system get pods -l component=kube-controller-manager
$ kubectl get leases -n kube-system kube-controller-manager \
    -o jsonpath='{.spec.renewTime}'
```

La tercera es la más útil: los controladores que trabajan en modo activo-pasivo renuevan un arrendamiento continuamente. Una marca de tiempo vieja significa que nadie está reconciliando, aunque los procesos figuren en ejecución.

4. El estado es un informe, y llega tarde

Esta sección responde a una pregunta que aparece en el primer incidente serio: ¿cuánto tarda el clúster en enterarse de que un nodo ha caído, y qué hace mientras tanto?

La secuencia por defecto, con los plazos que conviene conocer:

```
t+0 s     el nodo pierde la red
t+~40 s   deja de renovar su estado; se marca como no listo
t+~40 s   los puntos de conexión de sus pods empiezan a retirarse del reparto
t+~5 min  se aplica la expulsión: los pods se marcan para eliminar
t+5 min+  los controladores crean sustitutos en otros nodos
```

Cinco minutos entre la caída y la sustitución, con los pods figurando como en ejecución durante casi todo ese tiempo. Y hay un matiz importante: si el nodo solo perdió la red pero sigue vivo, sus contenedores siguen ejecutándose y pueden seguir escribiendo en una base de datos aunque el clúster los dé por perdidos.

Eso tiene tres consecuencias de diseño:

```
1. la retirada del tráfico es más rápida que la sustitución
   → el reparto deja de enviar antes; es lo que evita errores visibles
2. una carga con estado no puede asumir que su sustituto es único
   → dos instancias pueden coexistir durante la ventana (clase 077)
3. los plazos se pueden acortar, y acortarlos tiene precio
   → un clúster que expulsa a los 30 s reacciona a un parpadeo de red
     reubicando cargas que no hacía falta mover
```

La tercera es un intercambio real: reaccionar antes significa reaccionar también a falsos positivos. La configuración por defecto está calibrada para no mover nada por un problema de red pasajero, y modificarla exige saber qué se prefiere.

Y el mismo principio se aplica a lo que se lee en la API sobre cualquier objeto:

```
$ kubectl get pod tienda-7f4c9 -o jsonpath='{.status.phase} {.status.conditions[?(@.type=="Ready")].
lastTransitionTime}'
Running 2026-08-03T09:12:44Z
```

Ese Running lo escribió el kubelet cuando pudo. Si el nodo está incomunicado, la información es de la última vez que se pudo comunicar. Para saber si un servicio responde, hay que pedirle una respuesta, no consultar su estado en la API. Es la misma distinción entre configuración y realidad que este programa ha usado en las cinco partes anteriores, ahora con un mecanismo propio.

Y una precisión sobre el borrado, que sorprende la primera vez:

```
$ kubectl delete pod tienda-7f4c9
pod "tienda-7f4c9" deleted            ← también significa: registrado
```

El borrado marca el objeto con una marca de tiempo y un plazo de gracia; el kubelet envía la señal, espera y confirma; solo entonces el objeto desaparece. Y si el objeto tiene finalizadores —marcas que exigen que otro controlador termine su trabajo antes—, el borrado se queda esperando indefinidamente si ese controlador no está:

```
$ kubectl get ns pruebas -o jsonpath='{.spec.finalizers}'
["kubernetes"]
$ kubectl get ns pruebas -o jsonpath='{.status.conditions}' | jq -r '.[].message'
Some resources are remaining: pods has 3 resource instances
```

Un espacio de nombres que lleva horas en estado de eliminación casi siempre es esto: un controlador que debía limpiar algo y ya no existe. Quitar el finalizador a mano lo desbloquea y deja el trabajo sin hacer, así que antes hay que saber qué era.

5. El mismo bucle para todo lo demás

La última propiedad del modelo explica por qué el ecosistema tiene la forma que tiene: los bucles y los objetos no son privilegiados. Se pueden añadir tipos nuevos y controladores propios que funcionan exactamente igual.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata: {name: catalogos.cloudshop.example}
spec:
  group: cloudshop.example
  scope: Namespaced
  names: {kind: Catalogo, plural: catalogos}
  versions:
    - name: v1
      served: true
      storage: true
      schema: {openAPIV3Schema: {type: object, properties: {spec: {type: object,
        properties: {origen: {type: string}, refrescoMinutos: {type: integer}}}}}}
```

Con eso, Catalogo es un objeto de primera clase: se valida, se guarda, se consulta con las mismas órdenes y se le pueden aplicar los mismos permisos. Y un controlador propio que lo observe y actúe convierte una operación manual en algo declarativo.

Eso es lo que hay detrás de los operadores: un controlador que sabe operar un sistema concreto —una base de datos, una cola, un certificado— y que expresa esa operación como reconciliación. Y trae la misma ventaja y el mismo riesgo que el resto del modelo:

```
ventaja   la operación se declara y el bucle la mantiene, incluso tras un reinicio
riesgo    si el controlador no está, lo declarado no ocurre y nadie avisa
```

La segunda es, otra vez, la familia de fallos que la clase 060 identificó como la más cara, y en esta parte va a aparecer varias veces. Merece una regla de operación:

```
cada controlador propio o de terceros necesita:
  una métrica de "última reconciliación con éxito"
  una alerta si esa marca envejece más de lo esperado
```

Sin eso, un controlador caído es indistinguible de un sistema en el que no hay nada que hacer.

Y el diagnóstico general del modelo, que sirve para cualquier bucle, propio o no:

```
# 1. ¿qué dice el objeto de sí mismo?
$ kubectl get catalogo principal -o yaml | yq '.status'

# 2. ¿qué sucesos ha generado?
$ kubectl get events --field-selector involvedObject.name=principal \
    --sort-by=.lastTimestamp

# 3. ¿qué dice el controlador?
$ kubectl -n cloudshop-system logs deploy/catalogo-controller --tail=100

# 4. ¿está reconciliando alguien?
$ kubectl get leases -A | grep catalogo
```

Cuatro preguntas, en ese orden, para cualquier objeto que no llega al estado que debería. Y la conclusión que cierra la clase: en Kubernetes no se diagnostica «qué falló» sino «qué bucle no avanzó», y esa reformulación es lo que hace manejable un sistema con decenas de controladores funcionando a la vez.

Ejemplo trabajado

CloudShop estrena su clúster. La aplicación de la parte 05 se despliega sin cambios, y los cuatro incidentes del primer mes son todos del modelo, no de la aplicación.

Incidente 1 — el despliegue que devolvió éxito y no hizo nada.

```
$ kubectl apply -f tienda.yaml
deployment.apps/tienda configured
$ kubectl get pods -l app=tienda
NAME             READY   STATUS    AGE
tienda-6b1d4-x   1/1     Running   6d      ← la versión ANTIGUA, seis días después
```

La canalización daba el despliegue por bueno porque apply había devuelto éxito. El controlador de despliegues llevaba veinte minutos sin reconciliar:

```
$ kubectl get leases -n kube-system kube-controller-manager \
    -o jsonpath='{.spec.renewTime}'
2026-08-03T08:41:12Z          ← hace 22 minutos
```

```
verificación en la canalización        `apply` sin error   `rollout status`
                                                           con tiempo límite
alerta sobre el arrendamiento de los
  controladores                              ninguna       marca > 60 s → aviso
despliegues silenciosamente no aplicados     3 (en un mes)      0
```

Es la ley que la clase 072 predijo, con su primer coste medido: aceptado no es funcionando, y una canalización que confunde ambas cosas informa de despliegues que no ocurrieron.

Incidente 2 — nadie puede crear pods en todo el clúster.

```
Error from server (InternalError): Internal error occurred:
  failed calling webhook "politica.cloudshop.example": context deadline exceeded
```

El complemento de admisión que verifica firma y procedencia (clase 067) se ejecutaba como pods dentro del clúster, con política estricta y sin excluir su propio espacio de nombres. Un mantenimiento del nodo dejó sus dos réplicas fuera a la vez, y a partir de ahí no se podían crear los pods que lo habrían restaurado.

```
espacio de nombres del complemento     dentro del alcance   excluido
tiempo de espera del complemento             10 s              3 s
réplicas y reparto                    2, sin restricción   3, en zonas distintas
                                                           y con presupuesto de
                                                           interrupción (clase 079)
duración del bloqueo                          41 min             —
procedimiento de emergencia                  ninguno        documentado y ensayado
```

La salida fue retirar la configuración del complemento con una credencial de administración directa contra la API. El procedimiento existe ahora por escrito, con la advertencia de que durante ese intervalo no se verifica ninguna firma.

Incidente 3 — el clúster deja de aceptar cambios, sin decir por qué.

```
Error from server: etcdserver: mvcc: database space exceeded
```

```
$ kubectl get --raw /metrics | grep etcd_mvcc_db_total_size_in_bytes
etcd_mvcc_db_total_size_in_bytes 2.147221504e+09       ← contra una cuota de 2 GiB
```

La causa: un controlador de terceros actualizaba el estado de sus objetos cada segundo, generando revisiones sin parar. El clúster pasó a solo lectura y, mientras tanto, todo lo que ya estaba en ejecución siguió funcionando — lo que retrasó la detección.

```
compactación                          por defecto, insuficiente   cada 5 min
desfragmentación                          nunca           mensual, por nodo
alerta sobre tamaño del almacén          ninguna         > 60 % de la cuota
frecuencia de escritura del controlador   1 s          30 s (corregido con
                                                          el proveedor)
cifrado en reposo de los secretos           no               sí
copia de seguridad probada                  no        semanal, restaurada
                                                       en un clúster de prueba
```

La última fila apareció al revisar el incidente: había copias del almacén, nunca se había restaurado ninguna, y contenían todos los secretos en claro. Tercera aparición en el programa de la ley del sistema de solo añadir de la clase 072.

Incidente 4 — cinco minutos sirviendo desde un nodo aislado.

Un nodo perdió la red del centro de datos. Sus contenedores siguieron vivos.

```
t+0 s      el nodo pierde la red
t+42 s     se marca como no listo · los puntos de conexión se retiran
t+43 s     el tráfico deja de llegarle           ← el usuario deja de notarlo
t+5 min    se expulsan los pods
t+5 min 20 s  los sustitutos están listos
```

El comportamiento fue el correcto y descubrió un supuesto falso del equipo: creían que un nodo caído se sustituye en segundos. Lo que ocurre en segundos es dejar de enviarle tráfico; sustituirlo tarda minutos por diseño.

```
tiempo hasta dejar de recibir tráfico       "inmediato"       43 s
tiempo hasta la sustitución                 "~1 min"        5 min 20 s
```

```
capacidad necesaria para tolerarlo        no calculada    margen para perder
                                                          un nodo sin degradar
```

Y una consecuencia que se documentó como riesgo: durante esos cinco minutos, los contenedores del nodo aislado siguieron escribiendo en la base de datos. Para cargas sin estado da igual; para las de la clase 077 no, y ahí está la primera aparición del problema que aquella clase resuelve.

Resumen del primer mes:

```
despliegues aceptados y no aplicados              3             0
verificación de despliegue en la canalización  `apply`     `rollout status`
duración del bloqueo por admisión              41 min      procedimiento escrito
alertas sobre el almacén de estado                0             3
copia del almacén restaurada alguna vez          no         sí, semanal
secretos cifrados en reposo                      no            sí
supuestos sobre tiempos de caída          no medidos      43 s y 5 min 20 s
```

La lección que esta clase traslada al resto de la parte 06: los cuatro incidentes salen del mismo mecanismo y ninguno es un fallo del sistema. Escribir devuelve «registrado», los bucles convergen con el tiempo, el estado es un informe con retraso y un bucle que no funciona no produce ningún error. Kubernetes es predecible en cuanto se deja de leerlo como un ejecutor de órdenes y se lee como lo que es: un almacén con controladores alrededor.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/073-api-server-etcd-scheduler-controllers-y-kubelet/lab.py
```

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

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

Evidencia esperada

El artefacto principal es manifiestos declarativos con estado observado. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye mapa-control-plane 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 canalización informa de un despliegue correcto y en producción sigue la versión anterior	apply devuelve éxito cuando el objeto queda almacenado, no cuando converge	Espera la convergencia con rollout status o wait y trata el tiempo límite como fallo del despliegue.
Nadie puede crear pods en todo el clúster	Un complemento de admisión con política estricta no responde, y a veces se ejecuta dentro del propio clúster	Excluye su espacio de nombres del alcance, usa tiempos de espera cortos y ten un procedimiento de emergencia ensayado.
El clúster deja de aceptar cambios con un error que no menciona el almacén	El almacén de estado superó su cuota por acumulación de revisiones	Compacta y desfragmenta con cadencia, alerta sobre el tamaño y corrige al controlador que escribe sin parar.
Un pod lleva horas en Pending	Ningún nodo pasa el filtrado del planificador	Lee los sucesos del objeto: el planificador enumera cuántos nodos descartó y por qué motivo cada grupo.
Un espacio de nombres lleva horas eliminándose	Tiene un finalizador y el controlador que debía completarlo ya no existe	Averigua qué trabajo quedaba pendiente antes de quitar el finalizador a mano; quitarlo desbloquea y deja el trabajo sin hacer.
Se asume que un nodo caído se sustituye en segundos	Retirar el tráfico es rápido; expulsar y recrear tarda minutos por diseño	Mide los dos tiempos, dimensiona el margen de capacidad y documenta que los contenedores del nodo aislado siguen vivos.

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 significa exactamente la respuesta de kubectl apply, y qué hay que hacer para verificar un despliegue?
2. ¿Por qué la reconciliación por nivel hace robusto al sistema y a la vez impide la consistencia inmediata?
3. Un pod está en Pending y otro en ContainerCreating. ¿Qué bucle es responsable de cada uno?
4. ¿Cómo puede un complemento de admisión dejar el clúster sin capacidad de recuperarse, y qué tres medidas lo evitan?
5. ¿Cuánto tarda el clúster en dejar de enviar tráfico a un nodo caído y cuánto en sustituir sus pods?

Referencias

- Kubernetes (2025). Cluster architecture — componentes del plano de control y del nodo. <https://kubernetes.io/docs/concepts/architecture/>
- Kubernetes (2025). Dynamic admission control — complementos que modifican y validan, y política de fallo. <https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/>
- Kubernetes (2025). Node status and eviction timings — plazos de detección, retirada y expulsión. <https://kubernetes.io/docs/concepts/architecture/nodes/>
- etcd (2025). Operations guide: maintenance — cuota, compactación, desfragmentación y copias. <https://etcd.io/docs/v3.5/op-guide/maintenance/>
- Kubernetes (2025). Custom resources and the operator pattern — extender el modelo con los mismos bucles. <https://kubernetes.io/docs/concepts/extend-kubernetes/operator/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 06 en PDF · Manual integral

Anterior	Índice	Siguiente
← 072 · Proyecto: stack OCI endurecido y observable	Parte 06 · Programa	074 · Pods, ReplicaSets, Deployments y Jobs →

Evaluación — 073 API server, etcd, scheduler, controllers y kubelet

Preguntas

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

Reto verificable

Entrega mapa-control-plane 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.

074 — Pods, ReplicaSets, Deployments y Jobs

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: kubernetes · Estado: EXECUTABLECORE

Propósito

Trabajar con las cargas de Kubernetes sabiendo que el pod no es un contenedor: es un grupo de contenedores que comparten espacio de red y ciclo de vida, exactamente el modo compartido de la clase 065. De esa definición salen los patrones auxiliares y sus problemas de orden, y de la naturaleza declarativa salen dos comportamientos que producen incidentes reales: un contenedor de inicialización se ejecuta una vez por pod —no una vez por despliegue— y los objetos terminados no desaparecen solos, con la consecuencia sobre el almacén de estado que la clase 073 acaba de mostrar.

Resultados de aprendizaje

Al finalizar podrás:

1. Explicar qué comparten los contenedores de un pod y qué patrones habilita eso.
2. Distinguir el trabajo que corresponde a un contenedor de inicialización del que no, con la trampa de la ejecución por réplica.
3. Describir cómo un despliegue progresa y cómo vuelve atrás, y qué lo impide.
4. Configurar trabajos y trabajos programados con límite de reintentos, plazo y limpieza.
5. Anticipar el efecto de los objetos terminados que nadie retira.

Conceptos centrales

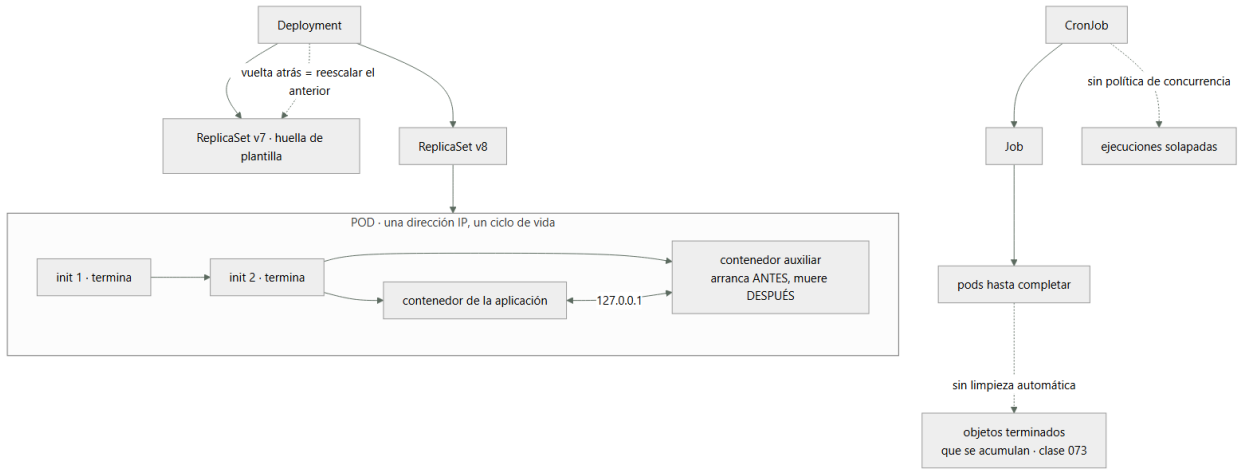
Concepto	Comprensión verificable
pod	Unidad de planificación: uno o varios contenedores que comparten espacio de red y almacenamiento efímero, se ubican juntos y viven y mueren juntos. Es el modo compartido de la clase 065 con nombre propio.
contenedor de inicialización	Se ejecuta hasta terminar, en orden, antes que los de la aplicación. Se ejecuta una vez por pod, así que con tres réplicas ocurre tres veces.
contenedor auxiliar	Acompaña al principal durante toda su vida. Su orden de arranque y de parada respecto del principal es lo que decide si hay errores en cada despliegue.
conjunto de réplicas	Mantiene N pods idénticos. No se crea a mano: lo crea el despliegue, y su nombre incluye una huella de la plantilla — que es como funciona la vuelta atrás.
límite de reintentos y plazo	Cuántas veces se reintenta un trabajo y cuánto puede durar. Sin ellos, un trabajo roto reintenta indefinidamente y uno colgado no termina nunca.
limpieza automática	Plazo tras el cual un trabajo terminado se elimina. Sin él, los objetos completados se acumulan para siempre en el almacén de estado.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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 pod es el modo compartido de la clase 065

Un pod es uno o varios contenedores que comparten cosas concretas, y saber cuáles evita la mitad de las confusiones:

```
comparten
  espacio de red     una sola dirección IP; se hablan por 127.0.0.1
                     y NO pueden escuchar en el mismo puerto
  almacenamiento efímero  los volúmenes declarados en el pod
  ciclo de vida      se planifican juntos, se ubican en el mismo nodo
                     y se eliminan juntos

no comparten
  sistema de ficheros    cada uno el suyo, salvo los volúmenes que monte
  espacio de procesos    salvo que se active explícitamente
  límites de recursos    cada contenedor tiene los suyos
```

La primera línea es literalmente el modo compartido que la clase 065 explicó, y por eso aquello no era una curiosidad: era el modelo mental de esta parte.

De ahí salen dos patrones, y la diferencia entre ellos es cuándo se ejecutan:

```
contenedor de inicialización   se ejecuta hasta TERMINAR, en orden,
                               antes de que arranque ningún contenedor principal
contenedor auxiliar            acompaña al principal durante toda su vida
```

Para qué sirve cada uno, con los casos que de verdad aparecen:

```
inicialización
  esperar a que una dependencia acepte conexiones (clase 066)
  ajustar la propiedad de un volumen (el problema de uid de la clase 064)
  descargar configuración o certificados antes de arrancar

auxiliar
  representante de red que cifra o enruta el tráfico
  recolector de métricas o de registros
  actualizador de configuración que refresca un fichero montado
```

Y la trampa del primero, que es la que produce incidentes y ya apareció en la clase 071 con otro mecanismo:

```
un contenedor de inicialización se ejecuta UNA VEZ POR POD
  → con tres réplicas, tres veces
  → al escalar a diez, diez veces
  → en cada reinicio de un pod, otra vez
```

Por eso una migración de esquema no va en un contenedor de inicialización. Parece el sitio natural —«antes de que arranque la aplicación»— y en realidad se ejecuta una vez por réplica y en cada reinicio. Es la segunda aparición de la misma ley que la clase 071 enunció, ahora con un mecanismo de Kubernetes:

```
migración de esquema  →  un trabajo, ejecutado una vez, antes del despliegue
esperar a la base     →  contenedor de inicialización, o mejor, reintento
                         en la aplicación (clases 066, 068)
```

La segunda opción merece un matiz: esperar en un contenedor de inicialización resuelve el arranque y no la vida, exactamente como dependson con condición de salud en la clase 066. El reintento en la aplicación sigue siendo obligatorio, y con él, el contenedor de inicialización suele sobrar.

Y sobre los contenedores auxiliares, el problema histórico que conviene conocer porque explica una configuración que se ve en muchos sitios: si el auxiliar es un contenedor normal, no hay garantía de que arranque antes que el principal ni de que muera después. Eso produce dos fallos exactos:

```
al arrancar   el principal intenta salir por el representante que aún no está
              → errores de conexión en cada creación de pod
al parar      el representante muere primero
              → las peticiones en curso del principal fallan (clase 068)
```

La forma correcta actual es declararlo como contenedor de inicialización con política de reinicio permanente, lo que le da orden garantizado: arranca antes que los principales y se termina después.

2. Cómo avanza y cómo vuelve atrás un despliegue

Un despliegue no gestiona pods: gestiona conjuntos de réplicas, y ahí está la explicación de la vuelta atrás.

```
$ kubectl get rs -l app=tienda
NAME               DESIRED   CURRENT   READY   AGE
tienda-7d4b9c8f5        3         3       3     2m     ← la nueva
tienda-6b1d4a2e7        0         0       0     6d     ← la anterior, conservada
```

El sufijo es una huella de la plantilla del pod. Cambiar cualquier campo de la plantilla produce una huella distinta y, con ella, un conjunto nuevo. Y volver atrás es volver a escalar el conjunto anterior, que sigue existiendo:

```
$ kubectl rollout undo deploy/tienda
$ kubectl rollout history deploy/tienda
```

De donde salen dos requisitos que se olvidan:

```
revisionHistoryLimit > 0     si es 0, no hay conjunto anterior que reescalar
la imagen por HUELLA         si la plantilla referencia una etiqueta y esa
                             etiqueta se movió, "volver atrás" descarga
                             el contenido nuevo (clase 061)
```

La segunda es la quinta aparición de la regla de la clase 061, ahora con la consecuencia más incómoda: una vuelta atrás que no vuelve a ninguna parte, con todo el mecanismo funcionando correctamente.

El avance se controla con dos parámetros y su combinación decide si hay capacidad de sobra o de menos durante el despliegue:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1          # cuántos pods DE MÁS se permiten
    maxUnavailable: 0    # cuántos pueden faltar
```

maxUnavailable: 0 es el valor correcto por defecto: se crean pods nuevos antes de retirar los viejos, así que la capacidad nunca baja. El coste es que hace falta sitio para uno más y que el despliegue es algo más lento.

Y un detalle que produce despliegues que parecen colgados: un pod cuenta como disponible cuando pasa su comprobación de disponibilidad, no cuando arranca. Si esa comprobación es exigente y la aplicación tarda, el despliegue avanza al ritmo del arranque real — que es lo correcto y hay que dimensionar el tiempo límite en consecuencia.

```
$ kubectl rollout status deploy/tienda --timeout=300s
Waiting for deployment "tienda" rollout to finish: 1 out of 3 new replicas have been updated...
```

Y el mecanismo que impide un despliegue infinito: si en un plazo no hay progreso, el despliegue se marca como fallido y se queda parado, con los pods viejos sirviendo:

```
progressDeadlineSeconds: 600
```

Eso es lo que se quiere: un despliegue que no puede avanzar no debe seguir intentándolo mientras el conjunto anterior mantiene el servicio. Y la canalización tiene que leer ese estado, porque apply habrá devuelto éxito hace rato — la ley de la clase 073.

Sobre la inmutabilidad: la mayoría de los campos de un pod no se pueden modificar. Cambiar la imagen, las variables o los límites significa crear otro pod. Eso es coherente con todo el modelo y explica por qué cualquier cambio produce

reemplazo, y por qué el despliegue existe para hacerlo de forma controlada.

3. Trabajos: reintentos, plazos y lo que nadie retira

Un trabajo ejecuta pods hasta que terminan con éxito, y tiene cuatro parámetros que hay que poner siempre porque sus valores por defecto producen problemas:

```
apiVersion: batch/v1
kind: Job
metadata: {name: facturar-dia}
spec:
  completions: 1
  parallelism: 1
  backoffLimit: 3                # reintentos antes de darse por vencido
  activeDeadlineSeconds: 3600    # plazo total, pase lo que pase
  ttlSecondsAfterFinished: 86400  # se elimina solo un día después
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: facturar
          image: registro/tienda@sha256:9f2c…
          command: ["/bin/tienda", "facturar-dia"]
```

Qué pasa sin cada uno:

```
sin backoffLimit          usa el valor por defecto y reintenta seis veces;
                          con un fallo permanente, seis pods fallidos que quedan
sin activeDeadlineSeconds un proceso colgado no termina NUNCA
                          y bloquea la ejecución siguiente si hay política
                          de concurrencia estricta
sin ttlSecondsAfterFinished  los objetos terminados NO se eliminan solos
                          → se acumulan indefinidamente
restartPolicy: Always     no es válido en un trabajo: reiniciaría para siempre
```

El tercero merece detenerse, porque es la conexión directa con la clase 073. Un trabajo programado que se ejecuta cada cinco minutos genera 288 objetos al día. Sin limpieza automática:

```
288 objetos/día × 3 trabajos programados × 90 días ≈ 78.000 objetos
cada uno con su pod, sus sucesos y su historial de estado
→ presión sobre el almacén de estado, listados lentos y, en el límite,
  el clúster en solo lectura de la clase 073
```

La comprobación es directa y conviene hacerla en cualquier clúster con unos meses de vida:

```
$ kubectl get jobs -A --no-headers | wc -l
$ kubectl get pods -A --field-selector status.phase=Succeeded --no-headers | wc -l
```

Y los trabajos programados añaden tres parámetros propios, dos de los cuales ya aparecieron en la clase 071:

```
spec:
  schedule: "0 2 * * *"
  timeZone: "Europe/Madrid"
  concurrencyPolicy: Forbid          # si la anterior sigue, no arranca otra
  startingDeadlineSeconds: 300       # si no pudo arrancar en 5 min, se salta
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 3
```

La política de concurrencia es la corrección del incidente de la clase 071: sin ella, una ejecución que dure más que el intervalo se solapa consigo misma.

El plazo de arranque resuelve un fallo que solo aparece después de un incidente y es espectacular: si el plano de control estuvo caído varias horas, al volver el controlador intenta recuperar todas las ejecuciones perdidas. Un trabajo cada cinco minutos durante seis horas de caída son 72 ejecuciones lanzadas a la vez. Con el plazo puesto, las que no pudieron arrancar a tiempo simplemente se saltan, que casi siempre es lo correcto.

Y la zona horaria merece declararse explícitamente: sin ella, el horario se interpreta en la del controlador, que puede no ser la del negocio. Un cierre diario «a las 2 de la madrugada» que se ejecuta a las 3 o a la 1 según la estación es un error difícil de atribuir.

Y para lo que debe ejecutarse en todos los nodos —un recolector de registros, un agente de red— existe el conjunto de demonios, que planifica un pod por nodo y añade automáticamente los nodos nuevos. Su particularidad operativa es que actualizarlo toca todos los nodos, así que su despliegue progresivo tiene que estar bien configurado o afecta al

clúster entero a la vez.

4. Propiedad, borrado en cascada y objetos huérfanos

Cada objeto creado por un controlador lleva una referencia a su propietario, y de ahí sale el borrado en cascada:

```
$ kubectl get rs tienda-7d4b9c8f5 -o jsonpath='{.metadata.ownerReferences}' | jq -r '.[].kind'
Deployment
$ kubectl get pod tienda-7d4b9c8f5-x2k4p -o jsonpath='{.metadata.ownerReferences}' | jq -r '.[].kind'
ReplicaSet
```

Borrar el despliegue borra sus conjuntos de réplicas y estos sus pods. Y hay una variante que produce sorpresas:

```
$ kubectl delete deploy tienda --cascade=orphan
```

Eso elimina el despliegue y deja los pods vivos, sin dueño. Nadie los reemplaza si mueren, nadie los actualiza, y no aparecen al listar despliegues. Es útil en migraciones controladas y es una fuente de recursos fantasma cuando se usa sin querer.

La comprobación de huérfanos merece estar en la revisión periódica de un clúster:

```
$ kubectl get pods -A -o json \
  | jq -r '.items[] | select(.metadata.ownerReferences == null)
           | "\(.metadata.namespace)/\(.metadata.name)"'
```

Cualquier pod sin propietario es un pod que nadie gestiona: no se recrea, no se actualiza y no lo cubre ningún presupuesto de interrupción de la clase 079.

Y los finalizadores, que la clase 073 introdujo, aparecen aquí en su forma habitual: un objeto que no se borra porque un controlador debía limpiar algo primero.

```
$ kubectl get pvc datos-tienda -o jsonpath='{.metadata.finalizers}'
["kubernetes.io/pvc-protection"]
```

Ese en concreto es una protección deliberada: impide borrar una reclamación de volumen mientras un pod la esté usando. Quitarlo a mano borra el objeto y deja el volumen real sin nadie que lo gestione — que es exactamente el tipo de recurso huérfano que aparece en la factura seis meses después.

Y la lista de comprobación de esta clase, que es lo que se lleva al proyecto de la 084:

- ninguna migración de esquema en contenedores de inicialización
- los auxiliares declarados con orden garantizado de arranque y parada
- imagen por huella en la plantilla, para que la vuelta atrás signifique algo
- revisionHistoryLimit mayor que cero
- maxUnavailable en cero salvo justificación
- progressDeadlineSeconds acorde al arranque real, y leído por la canalización
- todo trabajo con límite de reintentos, plazo y limpieza automática
- todo trabajo programado con política de concurrencia, plazo de arranque y zona horaria declarada
- ningún pod sin propietario

Nueve puntos, de los cuales cuatro corrigen valores por defecto que producen problemas y dos vienen directamente de incidentes de las clases 061 y 071.

5. Elegir el objeto correcto

Con seis tipos de carga disponibles, elegir mal produce trabajo que no encaja con el mecanismo. La tabla que evita la discusión:

Necesidad	Objeto	Por qué
Servicio sin estado, N réplicas intercambiables	Deployment	Despliegue progresivo y vuelta atrás
Identidad estable y almacenamiento propio por réplica	StatefulSet	Clase 077
Un pod en cada nodo	DaemonSet	Se ajusta solo al añadir nodos
Trabajo que termina	Job	Reintentos, paralelismo y plazo
Trabajo que termina, con horario	CronJob	Concurrencia y plazo de arranque
Un pod suelto	Pod	Casi nunca: nadie lo recrea

La última fila merece la advertencia: un pod creado directamente no lo gestiona ningún controlador. Si su nodo cae, desaparece y nadie lo sustituye. Solo tiene sentido para depuración puntual —y para eso está el contenedor efímero de la clase 070.

Y dos decisiones que se toman mal con frecuencia:

Un despliegue con una sola réplica no es alta disponibilidad. Durante cualquier actualización, cualquier expulsión y cualquier caída de nodo hay una ventana sin servicio. Si el servicio importa, el mínimo es dos réplicas repartidas, y eso exige que la aplicación tolere ejecutarse dos veces — que es la propiedad que las clases 064 y 071 llamaron estado adherido.

Un trabajo no es un servicio con horario. Un proceso de larga duración que «se reinicia si muere» no debe ser un trabajo con reintentos infinitos: es un despliegue con una réplica, y sus reinicios los gestiona el kubelet con espera creciente.

Y el cierre que enlaza con el resto de la parte: los objetos de esta clase describen qué debe existir. Nada de lo visto aquí dice cómo se llega a ello desde fuera, dónde se guardan los datos ni quién puede hablar con quién — que son las tres clases siguientes y, no por casualidad, tres de las cuatro fugas que la clase 072 predijo. La hipótesis va por buen camino: Kubernetes está resolviendo la reubicación y el despliegue progresivo, y las fugas siguen ahí esperando su nombre propio.

Ejemplo trabajado

CloudShop despliega en Kubernetes la aplicación de la parte 05. Los cinco incidentes del primer trimestre son de los objetos de carga, y tres de ellos son leyes conocidas reapareciendo con un mecanismo nuevo.

Incidente 1 — la migración de esquema se ejecutó tres veces.

El equipo puso la migración en un contenedor de inicialización, que parecía el sitio natural.

```
$ kubectl logs tienda-7d4b9-x2k4p -c migrar | head -2
Aplicando migración 2026_08_add_column…
ERROR: column "descuento" of relation "pedidos" already exists
```

Con tres réplicas, tres ejecuciones simultáneas. Dos fallaron, el despliegue se quedó a medias y la base quedó con una migración aplicada parcialmente.

```
migración                          contenedor de inicialización   trabajo previo
                                                                  al despliegue
ejecuciones por despliegue                      3                      1
espera a la base de datos            en el mismo contenedor     reintento en la
                                                                  aplicación (068)
contenedores de inicialización                  1                      0
```

Segunda aparición de la ley de la clase 071 —lo que se ejecuta por réplica se ejecuta N veces— con un mecanismo distinto y el mismo daño.

Incidente 2 — errores en cada creación de pod, y otra vez al parar.

El representante de red que cifra el tráfico entre servicios estaba declarado como contenedor normal.

```
al arrancar   la aplicación intentaba conectar antes de que el representante
              escuchara → ~15 errores por pod creado
al parar      el representante moría primero → las peticiones en curso
              de la aplicación fallaban

declaración del auxiliar       contenedor normal   inicialización con reinicio
                                                   permanente: orden garantizado
errores por creación de pod             ~15                0
errores por parada de pod               ~8                 0
errores por despliegue completo         ~70                0
```

Es el mismo problema que la clase 068 resolvió dentro de un proceso, ahora entre contenedores del mismo pod: quién arranca antes y quién muere después.

Incidente 3 — 78.000 objetos terminados y el almacén al límite.

```
$ kubectl get jobs -A --no-headers | wc -l
26412
$ kubectl get pods -A --field-selector status.phase=Succeeded --no-headers | wc -l
51833
```

Tres trabajos programados sin limpieza automática, funcionando desde hacía tres meses. Los listados tardaban minutos y el almacén de estado se acercaba a la cuota — el mismo camino que produjo el incidente 3 de la clase 073.

```
limpieza automática                       ninguna         86.400 s (1 día)
historial conservado por trabajo programado  ilimitado      3 correctos, 3 fallidos
objetos de trabajos en el clúster           78.245            412
tiempo de un listado de todos los pods      2 min 40 s        1,8 s
tamaño del almacén de estado                1,4 GiB         210 MiB
```

Incidente 4 — 72 ejecuciones simultáneas después de un mantenimiento.

Tras seis horas de plano de control detenido por una actualización, al volver:

```
$ kubectl get jobs -l cronjob=sincronizar-stock --no-headers | wc -l
72
```

El controlador intentó recuperar todas las ejecuciones perdidas de golpe. Las 72 arrancaron a la vez contra el mismo proveedor externo, que las bloqueó — el mismo desenlace que el incidente de ritmo de la clase 056, con otro origen.

```
plazo de arranque                         sin declarar        300 s
política de concurrencia                    Allow             Forbid
zona horaria                              sin declarar    Europe/Madrid
ejecuciones tras una caída de 6 h             72               1
```

Incidente 5 — la vuelta atrás que no volvió a ninguna parte.

Un despliegue defectuoso se revierte y el problema persiste.

```
$ kubectl rollout undo deploy/tienda
deployment.apps/tienda rolled back
$ kubectl get pods -l app=tienda -o jsonpath='{..imageID}' | tr ' ' '\n' | sort -u
registro/tienda@sha256:c74e0182…      ← la misma imagen defectuosa
```

La plantilla referenciaba tienda:v8, y la canalización había reescrito esa etiqueta. Volver atrás reescaló el conjunto anterior, cuya plantilla apuntaba a la misma etiqueta, que ahora contenía el código nuevo.

```
referencia en la plantilla                etiqueta            huella
revisionHistoryLimit                          0                 10
vuelta atrás efectiva                        no                 sí, 34 s medidos
etiquetas inmutables en el registro          no                 sí
```

Quinta aparición de la regla de la clase 061, con la consecuencia más incómoda posible: todo el mecanismo funcionó correctamente y no sirvió de nada.

Resumen del primer trimestre:

```
ejecuciones de la migración por despliegue        3              1
errores por despliegue completo                 ~70              0
objetos de trabajos en el clúster              78.245           412
tiempo de un listado completo de pods         2 min 40 s       1,8 s
ejecuciones tras una caída de 6 h                72              1
vuelta atrás efectiva                            no          sí, 34 s
pods sin propietario                              4              0
```

La lección que esta clase traslada al resto de la parte 06: cuatro de los cinco incidentes venían de valores por defecto —sin limpieza, sin plazo de arranque, sin historial de revisiones, contenedor auxiliar sin orden— y ninguno era un fallo del sistema. Y tres eran leyes ya conocidas del programa reapareciendo con mecanismos nuevos. La lista de nueve comprobaciones de esta clase habría evitado los cinco, y ninguna de las nueve exige entender nada que las clases anteriores no hubieran enseñado ya.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/074-pods-replicasets-deployments-y-jobs/lab.py
```

El laboratorio selecciona el motor de práctica kubernetes 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 workload-kubernetes en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es manifiestos declarativos con estado observado. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye workload-kubernetes 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 migración de esquema se ejecuta varias veces y deja la base a medias	Está en un contenedor de inicialización, que se ejecuta una vez por pod	Ejecútala como un trabajo previo al despliegue; el contenedor de inicialización es para esperar o preparar, no para actuar una sola vez.
Errores al crear y al parar cada pod con un representante de red	El contenedor auxiliar no tiene orden garantizado respecto del principal	Decláralo como contenedor de inicialización con reinicio permanente: arranca antes y termina después.
Los listados tardan minutos y el almacén de estado crece sin parar	Los trabajos terminados no se eliminan solos	Pon limpieza automática en todos los trabajos y limita el historial de los programados.
Tras una caída del plano de control se lanzan decenas de ejecuciones a la vez	El controlador recupera las ejecuciones perdidas y no hay plazo de arranque	Declara plazo de arranque y política de concurrencia estricta; casi siempre saltarse lo perdido es lo correcto.
La vuelta atrás se completa y el problema persiste	La plantilla referencia una etiqueta que se reescribió, o no hay historial de revisiones	Referencia por huella, mantén historial mayor que cero y activa etiquetas inmutables en el registro.
Existen pods que nadie recrea ni actualiza	Se crearon sueltos o se borró su propietario dejándolos huérfanos	Usa siempre un controlador y revisa periódicamente los pods sin referencia de propietario.

■ 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 comparten los contenedores de un pod y con qué mecanismo de la clase 065 se corresponde?

2. ¿Por qué una migración de esquema no debe ir en un contenedor de inicialización?
3. ¿Cómo funciona exactamente una vuelta atrás y qué dos cosas pueden dejarla sin efecto?
4. ¿Qué cuatro parámetros hay que poner siempre en un trabajo y qué produce la ausencia de cada uno?
5. ¿Qué ocurre con los trabajos programados tras varias horas de plano de control detenido, y cómo se evita?

Referencias

- Kubernetes (2025). Pods — qué comparten los contenedores y ciclo de vida conjunto. <https://kubernetes.io/docs/concepts/workloads/pods/>
- Kubernetes (2025). Sidecar containers — orden garantizado de arranque y terminación. <https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/>
- Kubernetes (2025). Deployments — conjuntos de réplicas, estrategia, historial y vuelta atrás. <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>
- Kubernetes (2025). Jobs and automatic cleanup — reintentos, plazo y eliminación tras finalizar. <https://kubernetes.io/docs/concepts/workloads/controllers/job/>
- Kubernetes (2025). CronJob — concurrencia, plazo de arranque, zona horaria e historial. <https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 06 en PDF · Manual integral

Anterior	Índice	Siguiente
← 073 · API server, etcd, scheduler, controllers y kubelet	Parte 06 · Programa	075 · Services, DNS, Ingress y Gateway API →

Evaluación — 074 Pods, ReplicaSets, Deployments y Jobs

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 workload-kubernetes 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.

075 — Services, DNS, Ingress y Gateway API

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: network · Estado: EXECUTABLECORE

Propósito

Alcanzar un pod desde otro pod y desde fuera del clúster, que es la primera de las cuatro fugas de la clase 072 puesta a prueba: Kubernetes no resuelve el problema de red, le pone nombre propio. Un servicio es una abstracción sobre direcciones que cambian, y todo lo demás —resolución de nombres, entrada desde fuera, reparto— se construye encima con las mismas propiedades y las mismas trampas de la clase 065, más una nueva que solo aparece aquí: la lista de destinos se actualiza con retraso, y ese retraso es donde se pierden peticiones.

Resultados de aprendizaje

Al finalizar podrás:

1. Explicar qué hace un servicio y por qué su lista de destinos es el objeto que de verdad importa.
2. Elegir el tipo de servicio adecuado y saber qué crea cada uno fuera del clúster.
3. Diagnosticar la resolución de nombres interna y el coste oculto de su configuración por defecto.
4. Exponer un servicio desde fuera distinguiendo entrada clásica de la interfaz moderna.
5. Localizar la ventana en la que un despliegue pierde peticiones por retraso en la lista de destinos.

Conceptos centrales

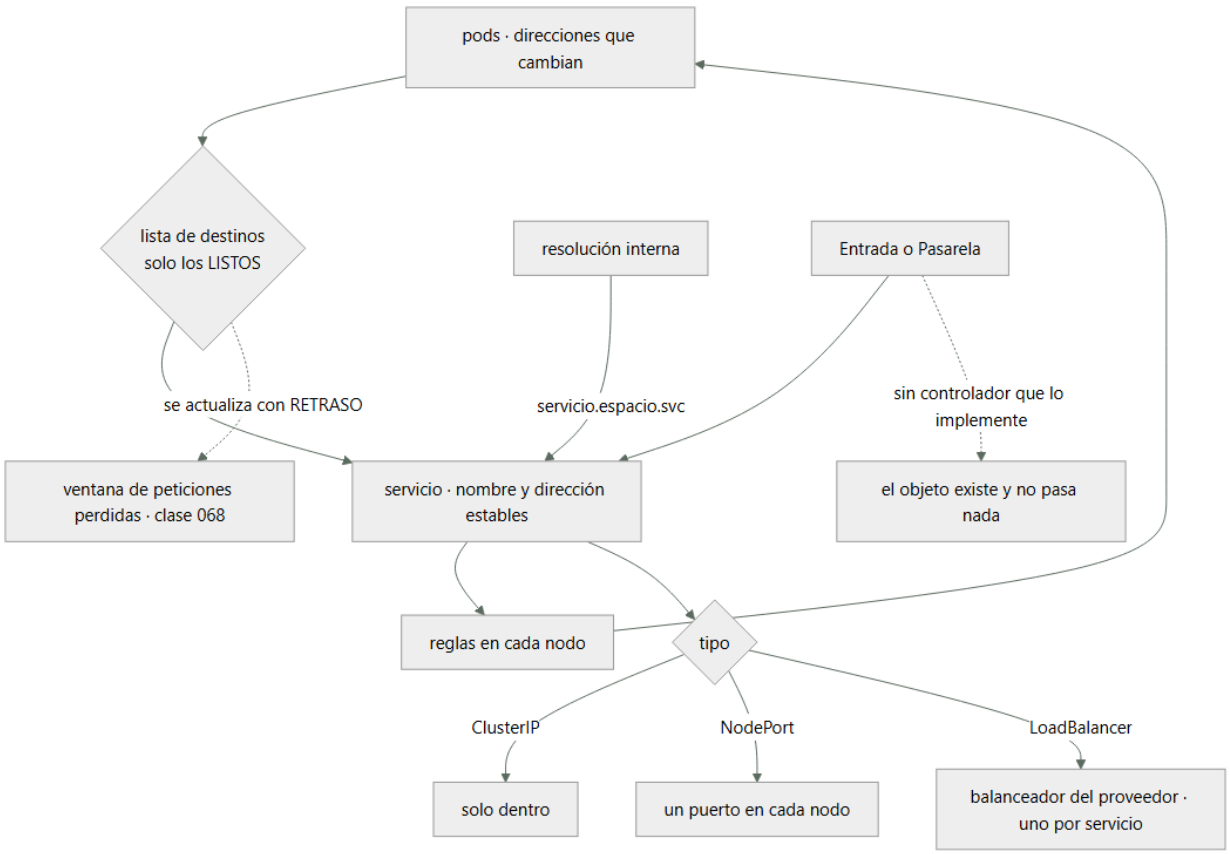
Concepto	Comprensión verificable
servicio	Nombre y dirección virtual estables delante de un conjunto de pods que cambian. No es un proceso: es una regla que el nodo aplica.
lista de destinos	Objeto que enumera las direcciones que están listas para recibir. Es lo que de verdad decide adónde va el tráfico, y se actualiza con retraso.
servicio sin dirección	Servicio que no asigna dirección virtual y devuelve directamente las de los pods. Es lo que necesitan las cargas con identidad de la clase 077.
entrada	Objeto que describe cómo llega el tráfico HTTP desde fuera. No hace nada por sí solo: hace falta un controlador que lo implemente.
interfaz de pasarela	Sustituta de la entrada, con los papeles separados: quien opera la infraestructura declara la pasarela y quien opera la aplicación declara sus rutas.
opción de puntos	Configuración del resolutor que hace que un nombre externo se intente primero con varios sufijos. Multiplica las consultas de DNS de todo el clúster.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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 servicio no es un proceso: es una regla

Un pod tiene una dirección y esa dirección desaparece con él. El servicio resuelve eso con un nombre y una dirección virtual estables:

```
apiVersion: v1
kind: Service
metadata: {name: api}
spec:
  selector: {app: api}
  ports: [{port: 80, targetPort: 8080}]
```

Y conviene entender qué ocurre por debajo, porque explica su comportamiento:

```
1. un controlador observa los pods que cumplen el selector
2. escribe en la LISTA DE DESTINOS los que están LISTOS
3. en cada nodo, un componente traduce esa lista a reglas del núcleo
4. el tráfico hacia la dirección virtual se reparte entre esos destinos
```

De ahí, tres consecuencias que hay que tener claras:

La dirección virtual no responde a nada. No hay ningún proceso escuchando en ella; es una dirección que solo existe como regla. Un intento de diagnosticarla con herramientas que esperan un destino real da resultados desconcertantes:

```
$ kubectl run t --rm -it --image=nicolaka/netshoot -- ping 10.96.14.7
# sin respuesta: no hay nada que responda a eso, y el servicio funciona
$ kubectl run t --rm -it --image=nicolaka/netshoot -- curl -s 10.96.14.7:80/readyz
ok                                                                    ✓
```

El objeto que importa es la lista de destinos, no el servicio. Casi todos los problemas de «el servicio no responde» se resuelven mirándola:

```
$ kubectl get endpointslices -l kubernetes.io/service-name=api \
    -o jsonpath='{range .items[*].endpoints[*]}{.addresses[0]} {.conditions.ready}{"\n"}{end}'
10.60.3.14  true
10.60.1.9   false      ← existe y NO está lista: no recibe tráfico
```

Una lista vacía tiene tres causas posibles y el diagnóstico es inmediato:

```
el selector no coincide con ninguna etiqueta de los pods
los pods existen y ninguno pasa su comprobación de disponibilidad
el puerto de destino no coincide con el que el contenedor declara
```

El reparto no es equilibrado en el sentido que se espera. Se reparte por conexión, no por petición. Un cliente con una conexión persistente envía todas sus peticiones al mismo destino mientras la mantenga abierta. Es exactamente el problema del reparto de la clase 066, y aparece igual:

```
cliente HTTP con conexión reutilizada, 3 réplicas
  → una réplica recibe todo el tráfico de ese cliente
  → al escalar, las nuevas no reciben nada hasta que alguien reconecte
```

La corrección está en el cliente —cerrar y rehacer conexiones periódicamente— o en una capa que reparta por petición, que es lo que hacen la entrada y las mallas de servicio. Conviene saberlo antes de concluir que «el reparto no funciona».

2. Los tipos, y qué crea cada uno fuera del clúster

```
ClusterIP      dirección virtual interna. El valor por defecto y el correcto
               para casi todo.
NodePort       además, un puerto ALTO en TODOS los nodos.
LoadBalancer   además, pide al proveedor un balanceador externo.
ExternalName   no reparte nada: devuelve un nombre. Es un alias de DNS.
```

Y dos cosas que conviene saber antes de elegir:

Cada servicio de tipo balanceador crea un recurso del proveedor, con su dirección pública, su coste por hora y su cuota. Veinte servicios expuestos así son veinte balanceadores, veinte direcciones y veinte facturas. Es la razón por la que existe la entrada: un único punto de entrada que reparte por nombre y por ruta.

```
20 servicios con balanceador propio     20 direcciones · ~20 × precio/hora
20 servicios detrás de una entrada       1 dirección  ·  1 × precio/hora
```

El puerto en todos los nodos rara vez es lo que se quiere. Abre un puerto alto en cada nodo del clúster, lo que complica el cortafuegos y ata a los clientes a direcciones de nodo que cambian. Casi siempre es un paso intermedio de la implementación del balanceador, no una elección de diseño.

Y hay un tipo que resuelve un caso concreto de la clase 077 y conviene situar aquí: el servicio sin dirección virtual.

```
spec:
  clusterIP: None
  selector: {app: bd}
```

Con eso, la resolución del nombre devuelve las direcciones de los pods en vez de una virtual. Sirve cuando el cliente necesita saber quién es quién —una base de datos con réplicas, un clúster que se coordina entre sus miembros—, y es lo que da a cada réplica un nombre estable en la clase 077.

Y una precisión sobre la preservación de la dirección de origen, que produce sorpresas al auditar:

```
por defecto, el tráfico que entra por un balanceador puede reenviarse
entre nodos, y en ese salto se sustituye la dirección de origen
→ la aplicación ve la dirección de un nodo, no la del cliente
```

La corrección es declarar que el tráfico solo se atienda en el nodo donde llega:

```
spec:
  externalTrafficPolicy: Local
```

Con eso, la aplicación ve la dirección real del cliente y la comprobación del balanceador solo marca como sanos los nodos que tienen pods — lo que además reparte mejor. El coste es que si un nodo tiene dos pods y otro ninguno, el reparto entre nodos deja de ser uniforme, así que conviene combinarlo con reglas de distribución de la clase 078.

3. La resolución interna y su coste por defecto

Cada servicio obtiene un nombre resoluble, con una forma jerárquica:

```
api                          desde el mismo espacio de nombres
api.tienda                   desde otro espacio
api.tienda.svc               forma completa dentro del clúster
api.tienda.svc.cluster.local nombre absoluto
```

Y hay un detalle de configuración que afecta al rendimiento de todo el clúster y que casi nadie revisa:

```
$ kubectl exec app-1 -- cat /etc/resolv.conf
search tienda.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
```

La última línea significa: cualquier nombre con menos de cinco puntos se prueba primero con todos los sufijos de búsqueda. Para un nombre externo:

```
consulta de api.proveedor.com (2 puntos, menos de 5)
  1. api.proveedor.com.tienda.svc.cluster.local   → no existe
  2. api.proveedor.com.svc.cluster.local          → no existe
  3. api.proveedor.com.cluster.local              → no existe
  4. api.proveedor.com                            → ¡por fin!
```

Cuatro consultas —ocho si el cliente pregunta también por direcciones de la versión 6 del protocolo— por cada resolución de un nombre externo. En un clúster con servicios que llaman mucho hacia fuera, eso multiplica la carga del servicio de nombres y añade latencia a cada conexión nueva.

Dos correcciones, y la primera es gratis:

```
1. usar el nombre absoluto, con punto final: api.proveedor.com.
   → una sola consulta
2. bajar la opción de puntos en los pods que llaman mucho hacia fuera
```

```
spec:
  dnsConfig:
    options: [{name: ndots, value: "2"}]
```

Y una tercera medida que es la más efectiva en clústeres grandes: una caché de resolución en cada nodo, que responde localmente y evita atravesar la red para cada consulta. Reduce la latencia de resolución y elimina una clase entera de fallos intermitentes bajo carga.

Y vuelve, por tercera vez en el programa, el problema de la caché en el cliente:

```
clases 039 y 051   el nombre resolvía al sitio equivocado
clase 065          el cliente cacheó la dirección y el contenedor se recreó
aquí               lo mismo, con pods que se recrean constantemente
```

Con pods que cambian en cada despliegue, un cliente que resuelve una vez y guarda la dirección habla con un pod que ya no existe. La corrección es la de siempre —resolver por nombre en cada conexión nueva, caducidad corta y reintento— y aquí es más importante que nunca, porque las direcciones cambian por diseño y con frecuencia.

4. Entrar desde fuera: entrada y pasarela

La entrada describe cómo llega el tráfico HTTP desde fuera:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tienda
  annotations:
    nginx.ingress.kubernetes.io/proxy-body-size: "25m"
spec:
  ingressClassName: nginx
  tls: [{hosts: [tienda.example], secretName: tienda-tls}]
  rules:
    - host: tienda.example
      http:
        paths:
          - path: /api
            pathType: Prefix
            backend: {service: {name: api, port: {number: 80}}}
```

Y la primera cosa que hay que saber: el objeto no hace nada por sí solo. Es una declaración que un controlador tiene que implementar. Sin controlador instalado, la entrada se crea, apply devuelve éxito y no ocurre nada — la ley de la clase 073 en su forma más literal.

```
$ kubectl get ingress tienda
NAME     CLASS   HOSTS            ADDRESS   PORTS
tienda   nginx   tienda.example             80, 443
#                                 ↑ vacío: nadie la ha implementado
```

La segunda es su límite conocido: la especificación cubre poco —nombre, ruta y cifrado— y todo lo demás se expresa con anotaciones específicas del controlador. Eso significa que una entrada escrita para un controlador no vale para otro sin reescribir sus anotaciones, que es justo lo contrario de la portabilidad que se busca.

La interfaz de pasarela existe para corregir ambas cosas, y su aportación principal no es técnica sino organizativa: separa los papeles.

```
# lo declara quien opera la infraestructura
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata: {name: entrada-publica, namespace: red}
spec:
  gatewayClassName: proveedor
  listeners:
    - name: https
      protocol: HTTPS
      port: 443
      tls: {certificateRefs: [{name: comodin-tls}]}
      allowedRoutes: {namespaces: {from: Selector,
        selector: {matchLabels: {expone: "si"}}}}
---
# lo declara cada equipo, en SU espacio de nombres
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata: {name: tienda, namespace: tienda}
spec:
  parentRefs: [{name: entrada-publica, namespace: red}]
  hostnames: ["tienda.example"]
  rules:
    - matches: [{path: {type: PathPrefix, value: /api}}]
      backendRefs: [{name: api, port: 80, weight: 90},
                    {name: api-canario, port: 80, weight: 10}]
```

Tres cosas que la entrada no daba y que aquí son parte de la especificación:

```
reparto por peso        el canario de la clase 052 y de Cloud Run, sin anotaciones
control de quién expone qué   la pasarela decide qué espacios pueden colgar rutas
separación de papeles   red opera la pasarela y los certificados;
                        cada equipo opera sus rutas sin tocar la infraestructura
```

La tercera es el mismo compromiso que la VPC compartida de la clase 051 resolvió: quien gobierna la red no tiene que ser un cuello de botella para quien despliega, siempre que el reparto de permisos esté bien hecho.

Y una precisión operativa sobre los certificados: en la mayoría de los clústeres los emite y renueva un controlador que observa objetos y habla con una autoridad. Es un controlador más, con la misma propiedad que todos: si deja de funcionar, los certificados no se renuevan y nadie avisa hasta que caducan. La alerta correcta no es sobre el controlador sino sobre lo que ve el cliente:

```
vigilar la caducidad del certificado SERVIDO, no el objeto del clúster
```

Sexta aparición de la misma lección, ahora con la variante que la clase 058 ya había señalado en Azure: el vencimiento se vigila donde lo ve el usuario.

5. La ventana donde se pierden peticiones

La clase 068 identificó la ventana entre la señal de parada y la retirada del tráfico. En Kubernetes esa ventana tiene un mecanismo concreto y merece verse entero, porque es la causa de los errores de despliegue que quedan después de haber corregido la aplicación.

```
t+0     el pod se marca para eliminar
t+0     el kubelet envía la señal de parada al contenedor
t+0     el controlador de destinos empieza a quitarlo de la lista
t+~0,3  la lista actualizada llega al servidor de API
t+~0,5  cada nodo recibe la lista y reescribe sus reglas
t+~1-3  la entrada y los clientes con conexiones abiertas se enteran
```

Es decir: la señal llega antes que la retirada, y el margen depende del tamaño del clúster. Con muchos nodos y muchos servicios, la propagación de reglas tarda más.

De ahí que la corrección de la clase 068 —esperar antes de cerrar el escuchador— sea aquí obligatoria y no una mejora:

```
lifecycle:
  preStop:
    exec: {command: ["/bin/sleep", "5"]}
terminationGracePeriodSeconds: 45
```

Y tiene una segunda mitad específica de esta clase: los clientes con conexiones persistentes no consultan la lista. Ya tienen una conexión abierta a una dirección concreta, y la seguirán usando aunque el pod haya salido de la lista. Solo la cierra:

```
el pod, respondiendo con la cabecera que indica cierre de conexión
o el propio cierre del escuchador al terminar el plazo
```

Por eso el orden importa: primero salir de la lista y seguir sirviendo, después dejar de reutilizar conexiones, y solo al final cerrar.

Y el mismo mecanismo explica un problema al arrancar: un pod entra en la lista cuando pasa su comprobación de disponibilidad, y las reglas tardan en propagarse. Si el despliegue retira pods viejos al ritmo al que los nuevos se declaran listos, puede haber un instante con menos capacidad efectiva de la que las cifras indican. Con maxUnavailable: 0 y un margen de capacidad, deja de importar.

La comprobación, que es la misma de la clase 068 y ahora se ejecuta contra el clúster:

```
$ hey -z 3m -c 50 -q 20 https://tienda.example/api/pedidos > carga.txt &
$ sleep 60 && kubectl set image deploy/api api=registro/api@sha256:nueva…
$ kubectl rollout status deploy/api --timeout=300s
$ grep -E '\[503\]|\[502\]|errors' carga.txt
  [502] 0 responses
  [503] 0 responses
  errors 0                                                              ✓
```

Y una fuente de errores que solo aparece en el clúster y no en Compose: el controlador de entrada tiene su propia lista de destinos, que puede actualizarse a un ritmo distinto del de las reglas del nodo. Con un despliegue rápido de muchos pods, el controlador puede seguir enviando a direcciones que ya no existen durante uno o dos segundos. Los controladores modernos lo mitigan leyendo la lista directamente y con reintentos; comprobarlo forma parte de la verificación del despliegue, no de la confianza en la herramienta.

Ejemplo trabajado

CloudShop expone su aplicación en el clúster. La conectividad interna funciona en media hora y los cinco incidentes siguientes reparten la culpa entre valores por defecto, un controlador ausente y una ventana de propagación que nadie había medido.

Incidente 1 — el servicio no responde y los pods están sanos.

```
$ kubectl get endpointslices -l kubernetes.io/service-name=api
NAME        ADDRESSTYPE   ENDPOINTS   AGE
api-x7d2f   IPv4          <unset>     4m
```

Lista vacía. Y la causa, en el selector:

```
$ kubectl get svc api -o jsonpath='{.spec.selector}'
{"app":"api"}
$ kubectl get pods -l app=api --no-headers | wc -l
0
$ kubectl get pods --show-labels | head -2
api-7d4b9-x2k4p   1/1  Running  app.kubernetes.io/name=api
```

Las etiquetas de la plantilla usaban la forma con prefijo y el selector no. Media hora de diagnóstico que la lista de destinos habría resuelto en treinta segundos.

```
primera comprobación ante "no responde"   logs del pod    lista de destinos
convención de etiquetas                mezclada         una sola, documentada
comprobación en la canalización           ninguna     falla si la lista está vacía
```

Incidente 2 — la entrada existe y no llega tráfico.

```
$ kubectl get ingress -A
NAMESPACE  NAME     CLASS   HOSTS            ADDRESS   PORTS
tienda     tienda   nginx   tienda.example             80, 443
```

Sin dirección. No había ningún controlador de entrada instalado en el clúster: el objeto se había creado correctamente y nadie lo implementaba.

```
controlador de entrada                    ninguno          instalado, 3 réplicas
comprobación de que la clase existe        ninguna     validación en la admisión
tiempo hasta detectarlo                    2 días            —
```

Dos días, porque apply había devuelto éxito y el objeto figuraba creado. La ley de la clase 073 en su forma más pura.

Incidente 3 — el servicio de nombres al límite y latencia en todas las conexiones nuevas.

```
$ kubectl -n kube-system logs -l k8s-app=kube-dns --tail=1 | head
[INFO] plugin/ready: Still waiting on: "kubernetes"
$ kubectl -n kube-system top pods -l k8s-app=kube-dns
NAME           CPU(cores)   MEMORY(bytes)
coredns-6f9b   940m         180Mi
```

El análisis del tráfico de consultas mostró que el 78 % eran búsquedas fallidas con sufijos, generadas por dos servicios que llamaban constantemente a una API externa.

```
nombres externos en la configuración   api.proveedor.com  api.proveedor.com.
opción de puntos en esos pods                 5                 2
caché de resolución por nodo                  no                sí
consultas por segundo al servicio de nombres  4.100             620
latencia de conexión nueva (p95)             34 ms             4 ms
```

Un punto final en dos cadenas de configuración eliminó tres cuartas partes de la carga.

Incidente 4 — la auditoría no podía identificar el origen de las peticiones.

Todos los registros de acceso mostraban direcciones de nodos, no de clientes.

```
política de tráfico externo              Cluster            Local
dirección de origen en los registros    la del nodo     la del cliente
comprobación del balanceador          todos los nodos   solo los que tienen pods
reparto entre nodos                     uniforme       proporcional a los pods
                                                       (compensado con reglas
                                                        de distribución · 078)
```

Incidente 5 — errores en cada despliegue, después de haber corregido la aplicación.

La aplicación ya tenía el apagado ordenado de la clase 068 y aun así aparecían errores.

```
errores por despliegue                 entre 20 y 60
todos en una ventana de ~1,5 s tras marcar cada pod para eliminar
```

La espera previa al cierre era de 2 segundos, calculada en la parte 05 sobre una sola máquina. En un clúster de 24 nodos, la propagación de reglas tardaba más:

```
# medición: tiempo desde que el pod sale de la lista hasta que el último nodo
# deja de enviarle tráfico
$ ./medir-propagacion.sh
p50 0,9 s · p95 2,8 s · p99 4,1 s
```

```
espera previa al cierre                   2 s              6 s
plazo de gracia                           30 s             45 s
errores por despliegue                   20-60             0
medición de la propagación               ninguna      trimestral, y al crecer
                                                      el clúster
```

La última fila es la lección: la espera correcta depende del tamaño del clúster, así que un valor calculado una vez deja de valer cuando la plataforma crece.

Resumen de la exposición:

```
tiempo de diagnóstico de "el servicio no responde"  30 min      < 1 min
objetos de entrada sin controlador                    1            0
consultas por segundo al servicio de nombres      4.100          620
latencia p95 de conexión nueva                     34 ms          4 ms
dirección de origen en los registros             del nodo     del cliente
errores por despliegue                             20-60          0
balanceadores del proveedor                           9           1
coste mensual de balanceadores                    ~198 USD      ~22 USD
```

La lección que esta clase traslada al resto de la parte 06: la hipótesis de la clase 072 se confirma en su primera prueba. Kubernetes no resolvió el problema de red: le puso nombres —servicio, lista de destinos, entrada, pasarela— y devolvió a la aplicación exactamente las mismas obligaciones de la parte 05: resolver por nombre en cada conexión, no cachear direcciones, esperar antes de cerrar y medir la ventana. Lo único genuinamente nuevo es que ahora la

ventana depende del tamaño del clúster, así que la cifra que se calculó en una máquina hay que volver a medirla aquí.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/075-services-dns-ingress-y-gateway-api/lab.py
```

El laboratorio selecciona el motor de práctica network 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 exposicion-kubernetes 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 topología con rutas, puertos y controles. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

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

■ Criterio de aceptación

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

Errores frecuentes

Síntoma	Causa probable	Corrección
Un servicio no responde y sus pods están sanos	La lista de destinos está vacía: selector que no coincide, comprobación de disponibilidad que falla o puerto mal declarado	Mira siempre la lista de destinos primero; resuelve en segundos lo que por los registros cuesta media hora.
Un objeto de entrada existe y no llega tráfico	No hay ningún controlador que implemente esa clase de entrada	Comprueba que el objeto tiene dirección asignada y valida en la admisión que la clase declarada existe.
El servicio de nombres del clúster está saturado y las conexiones nuevas tardan	La opción de puntos por defecto prueba varios sufijos antes de cada nombre externo	Usa nombres absolutos con punto final, baja la opción en los pods que llaman hacia fuera y añade caché por nodo.
Los registros muestran direcciones de nodos en lugar de las de los clientes	El tráfico externo se reenvía entre nodos y en ese salto se sustituye el origen	Declara la política de tráfico externo como local, y compensa el reparto con reglas de distribución de pods.
Siguen apareciendo errores en cada despliegue pese al apagado ordenado	La espera previa al cierre se calculó para una máquina y la propagación de reglas en un clúster grande tarda más	Mide la propagación real y ajusta la espera; vuelve a medirla cuando el clúster crezca.
Un cliente sigue hablando con un pod que ya no existe	Tiene una conexión persistente abierta y no consulta la lista de destinos	Deja de reutilizar conexiones durante el apagado y limita la vida de las conexiones en el cliente.

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 objeto hay que mirar primero cuando un servicio no responde, y qué tres causas explica?
2. ¿Por qué veinte servicios de tipo balanceador cuestan más que veinte rutas detrás de una entrada?
3. ¿Cuántas consultas genera resolver un nombre externo con la configuración por defecto, y cómo se reduce a una?
4. Describe la ventana entre la señal de parada y la retirada del tráfico en un clúster, y de qué depende su duración.
5. ¿Qué aporta la interfaz de pasarela sobre la entrada, y por qué su aportación principal es organizativa?

Referencias

- Kubernetes (2025). Service — tipos, selectores, política de tráfico externo y servicios sin dirección. <https://kubernetes.io/docs/concepts/services-networking/service/>
- Kubernetes (2025). EndpointSlices — la lista de destinos y su propagación. <https://kubernetes.io/docs/concepts/services-networking/endpoint-slices/>
- Kubernetes (2025). DNS for Services and Pods — nombres, sufijos de búsqueda y opción de puntos. <https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/>
- Kubernetes (2025). Gateway API — separación de papeles, reparto por peso y control de exposición. <https://gateway-api.sigs.k8s.io/>
- Kubernetes (2025). Pod termination and endpoint removal — orden de la señal y de la retirada de destinos. <https://kubernetes.io/docs/concepts/services-networking/service/#deleting-and-terminating-endpoints>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 06 en PDF · Manual integral

Evaluación — 075 Services, DNS, Ingress y Gateway API

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 exposicion-kubernetes 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.

076 — ConfigMaps, Secrets y configuración externa

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: configuration · Estado: EXECUTABLECORE

Propósito

Inyectar configuración y secretos en Kubernetes, donde dos comportamientos producen la mayoría de los incidentes y ninguno es evidente: un cambio de configuración no reinicia nada, así que la nueva versión convive con la antigua indefinidamente; y un secreto no está cifrado, solo codificado, de modo que quien pueda leerlo en el almacén o crear un pod en su espacio de nombres lo tiene. La clase 058 ya estableció cuándo se resuelve un secreto según cómo se consume; aquí se paga por tercera vez, con un mecanismo nuevo.

Resultados de aprendizaje

Al finalizar podrás:

1. Decidir entre variable de entorno y fichero montado sabiendo cuándo se actualiza cada uno.
2. Forzar un despliegue cuando cambia la configuración, en vez de confiar en que se propague.
3. Explicar qué protege y qué no protege un secreto de Kubernetes.
4. Traer secretos del gestor externo de las clases 035, 046 y 058 en lugar de duplicarlos.
5. Acotar quién puede leer secretos, contando los caminos indirectos.

Conceptos centrales

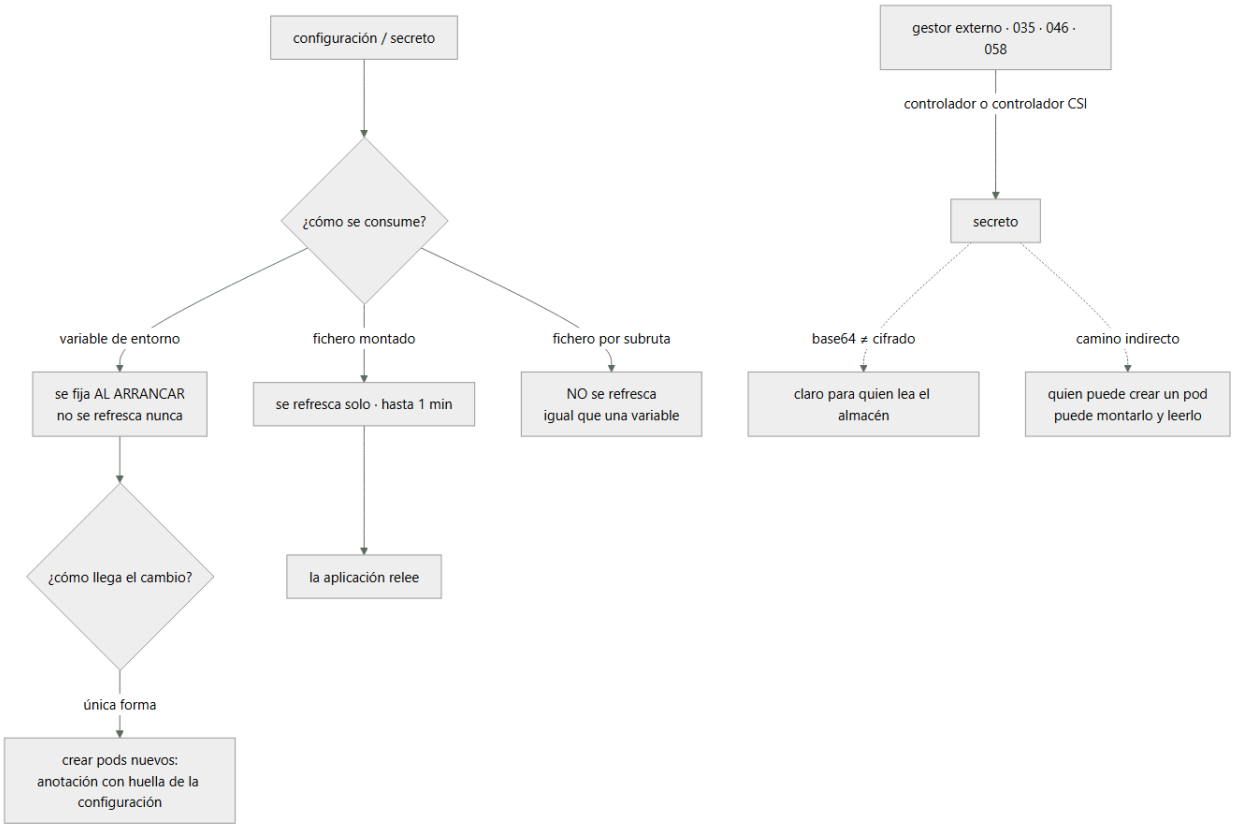
Concepto	Comprensión verificable
objeto de configuración	Pares clave-valor que se consumen como variables o como ficheros. Su tamaño total está limitado por el almacén de estado, así que no sirve para ficheros grandes.
secreto	Igual que el anterior, con el contenido codificado en base64, que no es cifrado. Sin cifrado en reposo del almacén, está en claro para quien pueda leerlo.
actualización de un montaje	Un fichero montado se refresca solo, con un retraso de hasta un minuto. Una variable de entorno no se refresca nunca: se fija al arrancar el contenedor.
montaje por subruta	Forma de montar una sola clave en una ruta concreta. Es cómoda y no recibe actualizaciones, lo que convierte un fichero montado en el equivalente de una variable.
objeto inmutable	Configuración o secreto marcado como no modificable. Reduce carga en el plano de control y obliga a crear uno nuevo para cambiar, que es lo que se quiere.
camino indirecto a un secreto	Quien puede crear un pod en un espacio de nombres puede montar cualquier secreto de ese espacio y leerlo. El permiso de lectura directa no es el único que concede acceso.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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. Cambiar la configuración no cambia nada

Este es el comportamiento que más sorprende y el que produce configuraciones divergentes que duran semanas.

```
$ kubectl edit configmap tienda-config      # se cambia LOG_LEVEL a debug
configmap/tienda-config edited
$ kubectl get pods -l app=tienda
# los mismos tres pods, con la configuración antigua, indefinidamente
```

Nada reinicia los pods. El objeto cambió y los contenedores siguen con lo que leyeron al arrancar. Y lo peor no es eso: lo peor es lo que ocurre la próxima vez que un pod se recrea por cualquier motivo —una expulsión, un cambio de nodo, un escalado—, porque entonces ese pod arranca con la configuración nueva y los demás siguen con la vieja.

```
resultado: réplicas del mismo servicio con configuraciones distintas,
           sin que nadie lo haya decidido y sin ninguna señal
```

Y el comportamiento depende de cómo se consume, que es la distinción de la clase 058 apareciendo por tercera vez:

```
variable de entorno   se fija al arrancar el contenedor · NO se refresca nunca
fichero montado       el kubelet lo refresca solo, con hasta un minuto de retraso
fichero por subruta   NO se refresca: se comporta como una variable
```

La tercera línea es una trampa fina: montar una sola clave en una ruta concreta es lo más cómodo y lo más habitual, y desactiva el refresco sin avisar.

```
# NO se actualiza
volumeMounts:
  - name: config
    mountPath: /app/config.yaml
    subPath: config.yaml

# SÍ se actualiza
volumeMounts:
  - name: config
    mountPath: /app/config          # el directorio entero
```

Y aunque el fichero se refresque, la aplicación tiene que releerlo. Un proceso que carga su configuración al arrancar no se entera de que el fichero cambió. Así que hay dos formas de que un cambio surta efecto, y conviene elegir una

explícitamente:

```
A · la aplicación vigila el fichero y relee
    lo mejor para lo que debe cambiar en caliente: nivel de registro,
    interruptores de funcionalidad, límites de ritmo

B · se crean pods nuevos
    lo correcto para todo lo demás, y lo que hace que el cambio sea
    un despliegue con vuelta atrás
```

La opción B necesita un mecanismo, porque cambiar la configuración no toca la plantilla del pod y por tanto no genera un despliegue (clase 074). El patrón estándar es una anotación con la huella del contenido:

```
spec:
  template:
    metadata:
      annotations:
        cloudshop.example/config-hash: "a1b2c3d4e5f6…"
```

La canalización calcula esa huella del contenido de la configuración y la escribe en la plantilla. Cambiar la configuración cambia la huella, la huella cambia la plantilla, y eso sí produce un despliegue progresivo con su historial y su vuelta atrás.

Y los objetos inmutables cierran el caso desde el otro lado:

```
apiVersion: v1
kind: ConfigMap
metadata: {name: tienda-config-v8}
immutable: true
data: {LOG_LEVEL: info}
```

Con ellos, la configuración se versiona igual que la imagen: para cambiarla se crea v9 y se apunta la plantilla ahí, lo que es a la vez un despliegue y un registro de qué configuración tenía cada versión. Además reduce la carga del plano de control, porque el kubelet deja de vigilar cambios que no pueden ocurrir.

2. Un secreto no está cifrado

El nombre invita a suponer una protección que no existe por defecto:

```
$ kubectl get secret bd-credenciales -o jsonpath='{.data.password}' | base64 -d
Pr0d-2026-…
```

Base64 es una codificación, no un cifrado. Lo que un secreto sí aporta frente a un objeto de configuración normal:

```
no se muestra en claro al listar ni en la salida por defecto
tiene su propio tipo de recurso, así que se le pueden dar permisos aparte
el kubelet lo monta en memoria, no en disco
se puede cifrar en reposo en el almacén, si se configura
```

Y lo que no aporta, que es lo que hay que asumir:

```
no está cifrado salvo que se active el cifrado en reposo del almacén (073)
no protege de quien pueda leer una copia de seguridad del almacén
no protege de quien pueda CREAR UN POD en ese espacio de nombres
```

La tercera es el camino indirecto que casi nunca se cuenta al auditar permisos:

```
# alguien con permiso para crear pods, sin permiso de leer secretos
spec:
  containers:
    - name: leer
      image: busybox
      command: ["sh", "-c", "cat /s/password"]
      volumeMounts: [{name: s, mountPath: /s}]
  volumes:
    - name: s
      secret: {secretName: bd-credenciales}
```

Montar un secreto no exige permiso de lectura sobre él: lo monta el kubelet en nombre del pod. Quien puede crear pods en un espacio de nombres puede leer todos sus secretos. Es la misma estructura del problema de suplantación de la clase 050 —el permiso efectivo incluye lo que se puede alcanzar indirectamente— y tiene la misma consecuencia para las auditorías: contar solo los permisos directos da un resultado falso.

De ahí, tres medidas concretas:

```
1. cifrado en reposo del almacén, con clave gestionada externamente
2. los secretos de producción no comparten espacio de nombres con
   cargas de menor confianza; el espacio de nombres es la frontera real
3. auditar quién puede crear pods, no solo quién puede leer secretos
```

Y la cuarta, que es la que de verdad resuelve: no guardar el secreto en el clúster. Las clases 035, 046 y 058 dejaron gestores de secretos en las tres nubes, con rotación, auditoría y control de acceso propios. Duplicar su contenido en el clúster multiplica los sitios donde está.

Dos mecanismos lo evitan:

```
controlador que sincroniza   observa un objeto que describe qué secreto quiere
                             y crea el secreto de Kubernetes desde el gestor
                             → el secreto SÍ acaba en el clúster, pero rota solo

controlador de almacenamiento de secretos
                             monta el valor directamente en el pod
                             → NO se crea ningún objeto en el clúster
```

El segundo es mejor y exige más: el valor nunca pasa por el almacén de estado. Y ambos usan la identidad de la carga —cuenta de servicio federada de las clases 038, 050 y 069— para autenticarse contra el gestor, sin ninguna credencial estática. Es el contrato de identidad del programa aplicado por sexta vez.

Y una advertencia sobre la rotación que la clase 058 ya midió: si el secreto se consume como variable de entorno, rotarlo en el gestor no cambia nada hasta que los pods se recreen. Con fichero montado y relectura, sí. Tercera vez que esta distinción decide el resultado.

3. Límites, y qué no cabe aquí

Los objetos de configuración viven en el almacén de estado, con las consecuencias de la clase 073:

```
tamaño máximo por objeto     ~1 MiB
se cargan en memoria del kubelet y del servidor de API
cada cambio genera una revisión en el almacén
```

De ahí una regla que evita problemas de escala: la configuración es texto pequeño. Lo que no cabe:

```
ficheros de datos, diccionarios, modelos, catálogos    → almacenamiento
                                                         de objetos o volumen
certificados de autoridad grandes                      → caben, pero conviene
                                                         revisar el total
plantillas y activos estáticos                         → dentro de la imagen
```

Y un patrón que produce el fallo más silencioso de esta clase: montar un objeto de configuración que no existe. Sin más opciones, el pod se queda esperando indefinidamente:

```
$ kubectl describe pod tienda-7d4b9-x2k4p | grep -A2 Events
  Warning  FailedMount  MountVolume.SetUp failed for volume "config":
    configmap "tienda-config-v9" not found
```

El pod no arranca, el despliegue no avanza, y el objeto anterior sigue sirviendo — que es el comportamiento correcto y hay que saber leerlo. Con la marca de opcional, en cambio, el pod arranca sin esa configuración, que casi nunca es lo que se quiere:

```
volumes:
  - name: config
    configMap: {name: tienda-config-v9, optional: true}   # ← arranca sin ella
```

Y dos prácticas que convierten la configuración en algo revisable:

Una fuente por entorno, versionada en el repositorio. El objeto del clúster se genera desde ahí, nunca se edita a mano. Un kubectl edit sobre una configuración es un cambio sin revisión, sin historial y que el siguiente despliegue deshace — y esa combinación produce incidentes cuya causa es imposible de encontrar.

Validación del contenido antes de aplicarlo. Un fichero de configuración mal formado no lo detecta Kubernetes: lo detecta la aplicación al arrancar, con lo que el fallo aparece en el despliegue.

```
$ kubectl create configmap tienda-config-v9 --from-file=config.yaml --dry-run=client -o yaml \
  | yq '.data["config.yaml"]' | yq -e 'has("puerto") and has("tiempoEspera")' >/dev/null \
  && echo "configuración válida"
```

Es la misma idea que la validación sobre el plan de la clase 059: comprobar antes de aplicar cuesta segundos y comprobar después cuesta un despliegue.

4. Quién puede leer qué

El control de acceso a la configuración y a los secretos se apoya en el espacio de nombres y en los permisos, y merece una regla explícita porque los caminos indirectos la complican.

La regla base:

```
el espacio de nombres es la frontera de confianza real
  → los secretos de producción no conviven con cargas de menor confianza
  → cada equipo en el suyo, con permisos acotados (clase 080)
```

Y el permiso que hay que vigilar de verdad no es el de leer secretos:

```
# quién puede leer secretos directamente
$ kubectl auth can-i list secrets --as=usuario -n produccion

# quién puede leerlos INDIRECTAMENTE, montándolos
$ kubectl auth can-i create pods --as=usuario -n produccion

# y quién puede leerlos a través de una cuenta de servicio (clase 050)
$ kubectl auth can-i create serviceaccounts/token --as=usuario -n produccion
```

Las tres preguntas conceden acceso efectivo. Una revisión que solo haga la primera da un resultado tranquilizador y falso.

Y hay un cuarto camino que aparece al integrar herramientas: quien puede leer el registro de auditoría o los sucesos puede ver valores si algún componente los registra. Una aplicación que imprime su configuración al arrancar publica sus secretos en los registros, que están en un sistema con otros permisos y otra retención — la ley del sistema de solo añadir de la clase 072, en su quinta aparición.

```
$ kubectl logs deploy/tienda | grep -Ei 'password|token|secret' | head
```

Esa comprobación debería estar en la canalización, sobre los registros del entorno de pruebas.

Y la lista de comprobación de la clase, que alimenta el proyecto de la 084:

- configuración versionada en el repositorio; nada editado a mano en el clúster
- objetos inmutables y versionados, o anotación con huella que fuerce despliegue
- montaje de directorio, no por subruta, para lo que deba refrescarse
- la aplicación relee lo que deba cambiar en caliente; lo demás, despliegue
- secretos desde el gestor externo, preferiblemente sin pasar por el clúster
- cifrado en reposo del almacén activado
- ningún secreto compartido entre espacios de nombres de distinta confianza
- auditoría de acceso que cuente crear pods y emitir testigos, no solo leer
- la aplicación no imprime su configuración al arrancar
- validación del contenido antes de aplicar

Diez puntos, de los cuales cuatro corrigen comportamientos por defecto y tres vienen de leyes ya establecidas en partes anteriores.

5. El contrato de configuración, ya en tres plataformas

Con las partes 02 a 06 recorridas, la configuración es uno de los contratos que mejor se ha conservado, y merece consolidarlo porque es lo que hace barato el cambio de plataforma:

```
lo que se repitió en todas
  configuración inyectada al ejecutar, nunca dentro del artefacto
  secretos fuera del artefacto y fuera del repositorio
  identidad de la carga para acceder al gestor, sin credenciales estáticas
  distinción entre lo que se lee al arrancar y lo que se relee
  rotación que no exige ventana de corte

lo que cambió de nombre
  variables y ficheros montados      →  igual en todas
  gestor de secretos                 →  uno por nube
  mecanismo de recarga               →  reinicio, revisión o refresco de fichero
```

La cuarta línea del primer bloque es la que ha producido un incidente en cada plataforma: la clase 058 con las variables de entorno que se resuelven al arrancar, la 066 con la configuración que no se relee, y esta con el montaje por subruta. Tres mecanismos distintos para la misma pregunta: cuándo llega el valor nuevo al proceso.

Y una consecuencia práctica que conviene dejar escrita, porque simplifica el diseño de cualquier aplicación del programa:

```
clasifica cada valor de configuración en dos grupos:

  A · puede cambiar en caliente
      nivel de registro, interruptores, límites de ritmo, umbrales
      → fichero montado, releído por la aplicación

  B · define la identidad de la versión
      cadenas de conexión, puntos de entrada, tamaños de grupo,
      credenciales, versiones de esquema
      → objeto inmutable versionado + despliegue con vuelta atrás
```

La clasificación hay que hacerla una vez por servicio y evita las dos preguntas que se repiten en cada cambio: «¿hace falta desplegar?» y «¿por qué unas réplicas tienen esto y otras no?».

Y el cierre que conecta con la clase siguiente: todo lo de aquí trata datos pequeños que caben en el plano de control. Lo que no cabe —los datos de verdad— es la segunda fuga de la clase 072, y la clase 077 comprueba si Kubernetes la resuelve o le pone nombre.

Ejemplo trabajado

CloudShop mueve su configuración al clúster. El primer mes produce cuatro incidentes, y el más caro es el que menos parece un incidente: dos réplicas del mismo servicio funcionando con configuraciones distintas durante once días.

Incidente 1 — las réplicas divergentes.

Un cambio de tiempo de espera se aplicó al objeto de configuración y nadie desplegó, porque «la configuración se lee del volumen».

```
$ for p in $(kubectl get pods -l app=api -o name); do
    echo "$p $(kubectl exec $p -- cat /app/config.yaml | yq '.tiempoEspera')"
  done
pod/api-7d4b9-x2k4p  30
pod/api-7d4b9-m9c1s  30
pod/api-7d4b9-q4t8w  5      ← recreado tras el cambio
```

El tercero se había recreado por una expulsión y arrancó con el valor nuevo. Durante once días, un tercio del tráfico tuvo un tiempo de espera seis veces menor, lo que explicaba una tasa de error intermitente que nadie había podido atribuir.

```
montaje                                por subruta      directorio completo
relectura en la aplicación                 no        sí, para los valores de A
anotación con huella de configuración     ninguna     en la plantilla del pod
objetos de configuración                 mutables      inmutables y versionados
réplicas con configuración divergente       3 de 3          0
```

Incidente 2 — quien podía crear pods podía leer todos los secretos.

Una auditoría de permisos daba un resultado tranquilizador: solo dos personas podían leer secretos en producción. La comprobación de los caminos indirectos dio otro:

```
$ for u in $(cat usuarios.txt); do
    kubectl auth can-i create pods --as=$u -n produccion >/dev/null 2>&1 \
      && echo "$u puede leer TODOS los secretos de produccion"
  done | wc -l
14
```

Catorce personas, no dos.

```
permiso de lectura directa de secretos      2 personas       2 personas
permiso de crear pods en producción        14 personas       3 (despliegue
                                                             por canalización)
espacios de nombres por confianza             1                 3
cifrado en reposo del almacén                no                 sí
auditoría que cuenta caminos indirectos      no             trimestral
```

Incidente 3 — un secreto rotado que no llegó a la mitad de la flota.

La rotación en el gestor externo funcionó; los pods siguieron con la credencial antigua.

```
$ kubectl get deploy api -o jsonpath='{.spec.template.spec.containers[0].env}' | jq -r '.[].name'
BD_PASSWORD
```

Como variable de entorno. Tercera aparición en el programa de exactamente el mismo problema —la clase 058 lo midió en Azure y la 066 en Compose—, ahora con el mecanismo de Kubernetes.

```
consumo del secreto                  variable de entorno  fichero montado
                                                          desde el controlador
                                                          de almacenamiento
el secreto pasa por el almacén de estado     sí               no
relectura ante fallo de autenticación        no               sí
errores durante la rotación siguiente      1.847              0
```

Incidente 4 — la aplicación publicaba su configuración al arrancar.

```
$ kubectl logs deploy/informes --tail=200 | grep -c 'BD_PASSWORD='
1
```

Una línea por arranque, con el valor completo, enviada al sistema de registros con su propia retención y sus propios permisos — la ley del sistema de solo añadir de la clase 072, quinta aparición.

```
volcado de configuración al arrancar    completo       solo claves, sin valores
credencial expuesta en los registros        sí          rotada el mismo día
comprobación en la canalización          ninguna      falla si aparece un patrón
                                                       de credencial en los registros
retención de los registros afectados      90 días       purgados
```

Resumen:

```
réplicas con configuración divergente          3 de 3           0
personas con acceso efectivo a secretos          14              3
secretos almacenados en el clúster               11              0
errores durante una rotación                   1.847             0
credenciales en los registros                     1              0
cambios de configuración con despliegue y vuelta atrás  0 de 9  9 de 9
```

La lección que esta clase traslada al resto de la parte 06: los cuatro incidentes son el mismo malentendido visto desde cuatro ángulos — creer que cambiar el objeto cambia lo que el proceso está usando. No lo cambia con variables, no lo cambia con subrutas, no lo cambia si la aplicación no relee y no lo cambia en el gestor externo si el consumo es por entorno. La corrección no es un ajuste sino una clasificación: qué valores pueden cambiar en caliente y cuáles definen la identidad de la versión, y tratarlos de forma distinta.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/076-configmaps-secrets-y-configuracion-externa/
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-kubernetes 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-kubernetes 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
Réplicas del mismo servicio con configuraciones distintas	Cambiar el objeto no reinicia nada, y los pods que se recrean arrancan con el valor nuevo	Objetos inmutables versionados o anotación con huella en la plantilla, para que un cambio de configuración sea un despliegue.
Un fichero montado no se actualiza nunca	Está montado por subruta, que desactiva el refresco	Monta el directorio completo y haz que la aplicación relea, o acepta que el valor exige despliegue.
Una auditoría dice que dos personas leen secretos y en realidad son catorce	Quien puede crear pods puede montar cualquier secreto de ese espacio de nombres	Cuenta los caminos indirectos —crear pods y emitir testigos— y separa por espacios de nombres según confianza.
Una rotación en el gestor externo no llega a los pods	El secreto se consume como variable de entorno, que se fija al arrancar	Consúmelo como fichero montado con relectura ante fallo de autenticación; es la tercera vez que esta distinción decide el resultado.
Un pod se queda esperando y el despliegue no avanza	Monta un objeto de configuración que no existe	Lee el suceso de montaje; y no marques como opcional lo que la aplicación necesita, porque arrancaría sin ello.
Aparecen credenciales en el sistema de registros	La aplicación imprime su configuración al arrancar	Registra solo las claves, rota lo expuesto y añade una comprobación de patrones de credencial a la canalización.

■ 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 ocurre exactamente al cambiar un objeto de configuración, y por qué produce réplicas divergentes?
2. ¿Cuándo se actualiza un valor consumido como variable, como directorio montado y como subruta?
3. ¿Qué aporta y qué no aporta un secreto de Kubernetes frente a un objeto de configuración normal?
4. ¿Qué tres permisos conceden acceso efectivo a un secreto, y cuál se suele olvidar en una auditoría?
5. Clasifica cinco valores de configuración de un servicio en los grupos de cambio en caliente y de identidad de versión.

Referencias

- Kubernetes (2025). ConfigMaps — consumo como variables y como volúmenes, y actualización. <https://kubernetes.io/docs/concepts/configuration/configmap/>

- Kubernetes (2025). Secrets — codificación, riesgos y buenas prácticas. <https://kubernetes.io/docs/concepts/configuration/secret/>
- Kubernetes (2025). Encrypting confidential data at rest — cifrado del almacén de estado. <https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/>
- Kubernetes (2025). Immutable ConfigMaps and Secrets — versionado y efecto en el plano de control. <https://kubernetes.io/docs/concepts/configuration/configmap/#configmap-immutable>
- Kubernetes SIG Storage (2025). Secrets Store CSI Driver — montar secretos del gestor externo sin crearlos en el clúster. <https://secrets-store-csi-driver.sigs.k8s.io/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 06 en PDF · Manual integral

Anterior	Índice	Siguiente
← 075 · Services, DNS, Ingress y Gateway API	Parte 06 · Programa	077 · Volumes, PersistentVolumes, CSI y StatefulSets →

Evaluación — 076 ConfigMaps, Secrets y configuración externa

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-kubernetes 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.

077 — Volumes, PersistentVolumes, CSI y StatefulSets

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: storage · Estado: EXECUTABLECORE

Propósito

Persistir datos en Kubernetes, que es la segunda fuga de la clase 072 y la que peor se resuelve. Kubernetes no elimina ninguna de las restricciones de la clase 064 —un volumen de bloque sigue teniendo un solo escritor y sigue viviendo en una zona— y añade dos propias: el volumen ata el pod a una zona antes de que el planificador elija nodo, y la garantía de identidad de una carga con estado no es una garantía de exclusión, así que durante una partición de red pueden existir dos instancias que se creen únicas.

Resultados de aprendizaje

Al finalizar podrás:

1. Encadenar los objetos de almacenamiento y saber cuál falla cuando un pod no arranca.
2. Elegir modo de acceso y política de vinculación sabiendo cómo afectan a la planificación.
3. Explicar qué garantiza un conjunto con estado y qué no garantiza durante una partición.
4. Anticipar el ciclo de vida del dato: qué sobrevive al borrado del pod, del objeto y del clúster.
5. Operar ampliaciones, instantáneas y restauraciones con una prueba que las demuestre.

Conceptos centrales

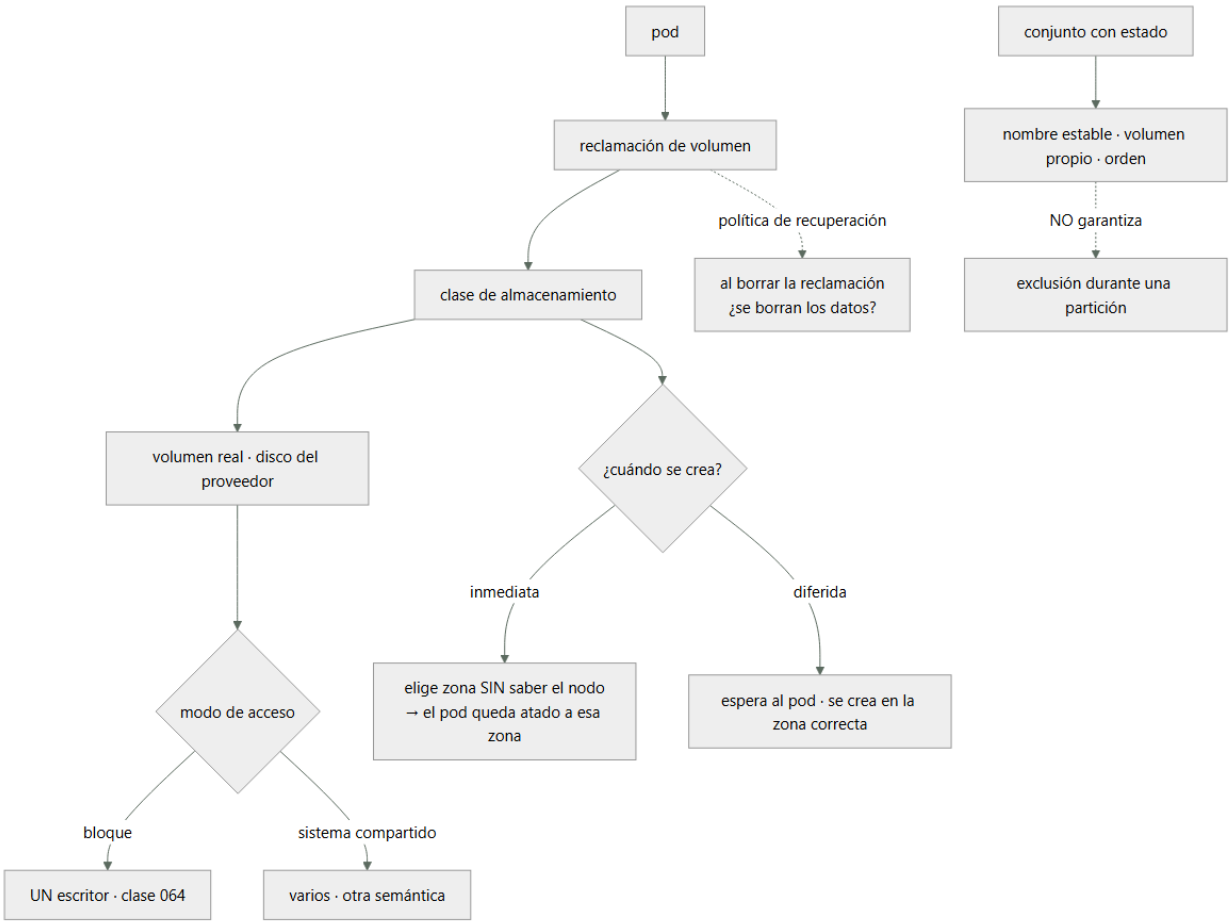
Concepto	Comprensión verificable
reclamación de volumen	Petición de almacenamiento hecha por la aplicación. Se vincula a un volumen real, y esa vinculación es permanente: no se reasigna a otro.
clase de almacenamiento	Plantilla que dice qué tipo de disco se aprovisiona y con qué política. Decide también cuándo se crea el volumen, que es lo que afecta a la planificación.
vinculación diferida	El volumen no se crea hasta que hay un pod que lo necesita, de modo que se crea en la zona del nodo elegido. Sin ella, el volumen elige zona primero y el pod queda atado a ella.
modo de acceso	Cuántos nodos pueden montarlo y cómo. El de bloque sigue siendo un escritor, exactamente como en la clase 064.
conjunto con estado	Da a cada réplica un nombre estable, un volumen propio y un orden. No garantiza que solo exista una instancia de cada índice durante una partición.
política de recuperación	Qué le ocurre al volumen real cuando se borra la reclamación: se elimina o se conserva. El valor por defecto de muchas clases borra los datos.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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 objetos encadenados, y cuál falla en cada caso

El almacenamiento en Kubernetes es una cadena de cuatro objetos, y saber cuál es cuál convierte «el pod no arranca» en un diagnóstico de treinta segundos:

```
pod                        pide un volumen por nombre
  reclamación de volumen   dice cuánto y de qué clase
    clase de almacenamiento  dice qué se aprovisiona y con qué política
      volumen real         el disco del proveedor
```

Y cada eslabón falla de una forma reconocible:

```
$ kubectl get pvc datos-bd
NAME       STATUS    VOLUME   CAPACITY   STORAGECLASS   AGE
datos-bd   Pending                       premium        6m
```

```
reclamación en Pending
  la clase no existe, o el aprovisionador no está, o
  es vinculación diferida y todavía no hay pod (correcto)

reclamación vinculada y pod en Pending
  el volumen está en una zona donde no hay nodo con sitio

pod en ContainerCreating
  el volumen existe y no se monta: permisos, controlador,
  o ya está conectado a otro nodo
```

El tercer caso da un mensaje muy concreto que conviene reconocer:

```
Multi-Attach error for volume "pvc-9f2c…":
  Volume is already exclusively attached to one node
```

Eso no es un fallo: es la restricción de la clase 064 en su forma de Kubernetes. Un volumen de bloque se conecta a un nodo cada vez, y punto. Aparece siempre en dos situaciones:

```
un despliegue con dos réplicas compartiendo una reclamación
  → la segunda nunca arranca
un despliegue progresivo con `maxUnavailable: 0`
  → el nuevo pod espera un volumen que el viejo aún no ha soltado
```

La segunda es la que rompe la recomendación general de la clase 074. Para una carga con volumen de bloque hay que invertir la estrategia: retirar antes de crear, y aceptar la ventana.

```
strategy:
  type: Recreate        # para cargas con volumen de bloque
```

Y los modos de acceso son la misma tabla de la clase 064 con otros nombres:

```
un nodo, lectura y escritura        disco de bloque · lo habitual
un pod, lectura y escritura         más estricto: ni siquiera dos pods del
                                    mismo nodo
muchos nodos, lectura y escritura   sistema de ficheros compartido
muchos nodos, solo lectura          contenido estático compartido
```

Y una advertencia que evita una decepción: declarar un modo no lo concede. Si el controlador de almacenamiento solo admite un escritor, pedir muchos hace que la reclamación se quede sin vincular o que el pod falle al montar. La comprobación es directa:

```
$ kubectl get sc premium -o jsonpath='{.provisioner}'
$ kubectl get csidrivers
```

2. El volumen elige zona antes que el planificador

Este es el comportamiento propio de Kubernetes que más desconcierta, y tiene una corrección de una línea.

Con aprovisionamiento inmediato, el volumen se crea en cuanto existe la reclamación, sin saber en qué nodo va a correr el pod. El aprovisionador elige una zona, y a partir de ahí el pod solo puede planificarse en esa zona:

```
t+0   se crea la reclamación
t+1   el volumen se crea en la zona B
t+2   se crea el pod
t+3   el planificador solo puede usar nodos de la zona B
      si no hay sitio allí → Pending indefinido, con nodos libres en A y C
```

El suceso lo dice, y hay que saber leerlo:

```
0/9 nodes are available: 6 node(s) had volume node affinity conflict,
  3 Insufficient cpu.
```

«Conflicto de afinidad de nodo del volumen» significa exactamente esto: el volumen ya eligió zona.

La corrección se declara en la clase de almacenamiento:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata: {name: premium}
provisioner: disk.csi.proveedor.com
volumeBindingMode: WaitForFirstConsumer     # ← diferida
allowVolumeExpansion: true
reclaimPolicy: Retain
parameters: {type: ssd}
```

Con vinculación diferida, el volumen no se crea hasta que hay un pod que lo necesita, así que el planificador elige nodo primero y el volumen se crea en la zona correcta. Debería ser el valor por defecto de cualquier clase que se use con discos zonales.

Y hay una consecuencia que sobrevive a la corrección y define el diseño: una vez creado, el volumen ata el pod a su zona para siempre. Si esa zona cae, el pod no se puede reubicar:

```
caída de zona con volumen zonal
  el pod no puede arrancar en otra zona
  la recuperación pasa por restaurar desde una instantánea en otra zona
  → el RTO es el de la restauración, no el de la reprogramación
```

Es exactamente lo que la clase 052 midió con el disco regional: 47 minutos frente a 4. Las opciones son las mismas y conviene decidirlas antes:

```
disco replicado entre zonas   más caro, permite reubicar; solo para el volumen
                              cuya disponibilidad define el RTO
réplicas en zonas distintas   la aplicación replica, cada réplica con su volumen
                              zonal; es lo que hace un conjunto con estado
```

```
instantáneas y restauración   el RTO es el tiempo de restaurar; barato y lento
```

La segunda es la que Kubernetes soporta mejor y la que lleva al objeto de la sección siguiente.

3. Identidad estable no es exclusión

Un conjunto con estado da tres cosas que un despliegue no da:

```
nombre estable        bd-0, bd-1, bd-2 · sobreviven al reinicio y al cambio de nodo
volumen propio        cada índice con su reclamación, creada por plantilla
orden                 se crean y se actualizan en orden, y se retiran al revés
```

Con un servicio sin dirección virtual (clase 075), cada réplica obtiene además un nombre resoluble, que es lo que permite a un clúster de base de datos coordinarse:

```
apiVersion: apps/v1
kind: StatefulSet
metadata: {name: bd}
spec:
  serviceName: bd-headless
  replicas: 3
  podManagementPolicy: OrderedReady
  volumeClaimTemplates:
    - metadata: {name: datos}
      spec:
        accessModes: [ReadWriteOnce]
        storageClassName: premium
        resources: {requests: {storage: 100Gi}}
```

```
bd-0.bd-headless.datos.svc.cluster.local
bd-1.bd-headless…
```

Y ahora la parte que hay que entender bien, porque es la fuente de corrupciones de datos y casi nunca se dice con claridad:

> El nombre estable no garantiza que solo exista una instancia con ese nombre.

Durante una partición de red, el nodo que aloja bd-0 puede seguir vivo con su contenedor ejecutándose, mientras el plano de control lo da por perdido. Y ahí hay dos comportamientos posibles según el tipo de volumen:

```
volumen de bloque      el mecanismo de conexión exclusiva IMPIDE que otro nodo
                       lo monte, así que el sustituto no arranca
                       → seguridad de datos, a costa de disponibilidad
sistema compartido     los dos pueden montar y escribir
                       → dos instancias escribiendo el mismo dato
```

Por eso Kubernetes no expulsa automáticamente los pods de un nodo incomunicado si tienen volúmenes: prefiere quedarse sin servicio a arriesgar dos escritores. Y por eso forzar la eliminación es una operación peligrosa que hay que conocer:

```
$ kubectl delete pod bd-0 --force --grace-period=0
```

Eso borra el objeto sin confirmar que el contenedor murió. Si el nodo estaba solo incomunicado, ahora hay dos bd-0: el viejo, escribiendo en su volumen, y el nuevo, que arranca en cuanto el volumen se libera. Con un sistema de ficheros compartido, eso es corrupción garantizada.

```
regla   forzar la eliminación de un pod con estado solo después de
        CONFIRMAR que el nodo está apagado, no solo incomunicado
```

Y dos parámetros que ajustan el comportamiento del conjunto:

```
podManagementPolicy
  OrderedReady   crea y actualiza de uno en uno, esperando disponibilidad
                 correcto para bases de datos que se unen a un clúster
  Parallel       todos a la vez; correcto cuando las réplicas son independientes

persistentVolumeClaimRetentionPolicy
  qué ocurre con los volúmenes al reducir réplicas o borrar el conjunto
  el valor por defecto los CONSERVA, lo que es seguro y acumula coste
```

La segunda merece revisarse: reducir de cinco réplicas a tres deja dos volúmenes conservados, con su coste, que nadie recuerda. Y si después se vuelve a escalar a cinco, se reutilizan con sus datos antiguos, que puede ser lo que se quiere o exactamente lo contrario.

4. El ciclo de vida del dato

Tres borrados distintos con tres consecuencias distintas, y confundirlos cuesta datos:

```
borrar el POD                el volumen sobrevive; el pod nuevo lo vuelve a montar
borrar la RECLAMACIÓN        depende de la política de recuperación
borrar el CLÚSTER            depende de la política; con eliminación automática,
                             se van todos los discos del proveedor
```

La política vive en la clase de almacenamiento y tiene un valor por defecto peligroso:

```
$ kubectl get sc -o custom-columns=NOMBRE:.metadata.name,POLITICA:.reclaimPolicy
NOMBRE     POLITICA
standard   Delete          ← borrar la reclamación BORRA el disco
premium    Retain
```

Con Delete, un kubectl delete pvc elimina el disco y sus datos. Y con eliminación en cascada (clase 074), borrar el espacio de nombres se lleva las reclamaciones y con ellas los discos.

```
regla   producción con Retain, siempre
        el coste es que quedan volúmenes liberados que hay que limpiar a mano,
        y ese coste es infinitamente menor que el de la alternativa
```

Y la protección adicional, que ya apareció en la clase 074: el finalizador que impide borrar una reclamación en uso. Quitarlo a mano es la forma habitual de perder datos «desatascando» algo.

Las instantáneas son el mecanismo de copia, y tienen la misma advertencia que la clase 064:

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata: {name: bd-2026-08-03}
spec:
  volumeSnapshotClassName: premium-snap
  source: {persistentVolumeClaimName: datos-bd-0}
```

```
una instantánea de un volumen con un motor en marcha NO es consistente
  → hay que usar el mecanismo del motor, o congelar la escritura
  → o aceptar que la restauración puede necesitar recuperación
```

Y la restauración crea una reclamación nueva a partir de la instantánea, lo que la hace segura de ensayar:

```
spec:
  dataSource: {name: bd-2026-08-03, kind: VolumeSnapshot,
               apiGroup: snapshot.storage.k8s.io}
```

Cuarta vez que este programa insiste en lo mismo, y con la misma exigencia: la restauración se prueba con un recuento y se registra su duración, porque esa duración es el tiempo de recuperación real y en las clases 042, 048 y 064 resultó ser bastante mayor que el del plan.

Y la ampliación de un volumen, que se puede hacer en caliente si la clase lo permite:

```
$ kubectl patch pvc datos-bd-0 -p '{"spec":{"resources":{"requests":{"storage":"200Gi"}}}}'
```

Con dos límites que conviene saber: no se puede reducir, igual que en la clase 054, y algunos controladores necesitan reiniciar el pod para que el sistema de ficheros se amplíe. Es otra decisión de un solo sentido con coste mensual permanente.

5. La pregunta que hay que hacer antes de crear un volumen

Con las restricciones ya enumeradas, conviene cerrar como cerró la clase 064: la mejor gestión de un volumen es no necesitarlo.

La tabla de decisión, que ya se puede escribir con seis partes de experiencia:

El dato es…	Dónde va	Por qué
Contenido subido por usuarios	Almacenamiento de objetos	Sin restricciones de zona ni de escritor
Sesión o caché	Servicio gestionado o se reconstruye	No merece un volumen
Datos relacionales	Servicio gestionado de la nube	Conmutación, copias y parches resueltos
Índice reconstruible	Volumen efímero o se reconstruye al arrancar	Se pierde y no pasa nada
Cola o registro de eventos	Servicio gestionado	Clases 033, 044, 056

El dato es…	Dónde va	Por qué
Dato que exige el motor en el clúster	Conjunto con estado	Y con todo lo de esta clase

La última fila existe y es minoritaria. Los motivos legítimos son concretos:

```
requisito de residencia que ningún servicio gestionado satisface
versión o extensión que el servicio gestionado no ofrece
coste, con la cuenta hecha e incluyendo el tiempo de operación
portabilidad exigida entre nubes, con la decisión escrita
```

Y el coste que hay que poner en esa cuenta y casi nunca se pone:

```
parches y actualizaciones de versión mayor
conmutación probada, no supuesta
copias, restauraciones ensayadas y su duración medida
vigilancia específica del motor
el conocimiento para operarlo, y su sustituto cuando esa persona no esté
```

Cuando esa lista se compara con la factura del servicio gestionado, la conclusión cambia en la mayoría de los casos. Y cuando no cambia, la decisión queda documentada con sus motivos, que es lo que este programa pide de cualquier decisión irreversible.

Y la lista de comprobación de la clase:

- vinculación diferida en toda clase con discos zonales
- política de recuperación de conservación en producción
- estrategia de recreación en despliegues con volumen de bloque
- conjuntos con estado con servicio sin dirección virtual y política de gestión acorde a si las réplicas se coordinan
- política de retención de reclamaciones decidida, no heredada
- instantáneas con el mecanismo del motor, y restauración probada con recuento
- duración de la restauración registrada como tiempo de recuperación real
- ningún `--force --grace-period=0` sobre cargas con estado sin confirmar que el nodo está apagado
- para cada volumen, la respuesta escrita a por qué no está en un servicio gestionado

Ejemplo trabajado

CloudShop lleva al clúster el único motor que no está gestionado: un índice de búsqueda con tres réplicas. Los cuatro incidentes del trimestre son todos de almacenamiento, y el último obliga a revisar la decisión de tenerlo ahí.

Incidente 1 — pods en Pending con nodos libres.

```
$ kubectl describe pod busqueda-0 | grep -A2 Events
  Warning  FailedScheduling  0/9 nodes are available:
    6 node(s) had volume node affinity conflict, 3 Insufficient memory.
```

La clase de almacenamiento usaba aprovisionamiento inmediato, así que los tres volúmenes se habían creado en la misma zona antes de que existiera ningún pod.

```
modo de vinculación                    inmediato        diferido
zonas de los tres volúmenes           B, B, B          A, B, C
tiempo en Pending                     indefinido          12 s
reparto de réplicas por zona           1 zona            3 zonas
```

Los volúmenes existentes hubo que recrearlos: la vinculación no se cambia sobre la marcha.

Incidente 2 — el despliegue progresivo que nunca terminaba.

Antes de pasar a conjunto con estado, el índice era un despliegue con un volumen compartido.

```
Multi-Attach error for volume "pvc-9f2c…":
  Volume is already exclusively attached to one node
```

Con maxUnavailable: 0, el pod nuevo esperaba un volumen que el viejo no soltaba, y el viejo no se retiraba hasta que el nuevo estuviera listo.

```
objeto                                Deployment       StatefulSet
volumen                            uno compartido    uno por réplica
estrategia                        RollingUpdate        OrderedReady
despliegue                           se bloqueaba      2 min 40 s
réplicas simultáneas posibles              1                3
```

Incidente 3 — se perdieron los datos de preproducción al limpiar un espacio de nombres.

```
$ kubectl delete namespace busqueda-pre
namespace "busqueda-pre" deleted
$ kubectl get pv | grep busqueda-pre
(vacío)
```

El borrado en cascada se llevó las reclamaciones, y la clase de almacenamiento tenía política de eliminación, así que se llevó también los discos del proveedor.

```
política de recuperación                  Delete          Retain
volúmenes liberados tras un borrado    desaparecen     quedan, con etiqueta
                                                        y limpieza mensual
instantánea previa a operaciones destructivas  ninguna   obligatoria
datos perdidos                          9 días de índice      —
```

El coste real fueron nueve días de reindexación. En producción la política ya era de conservación; en preproducción se había heredado la clase por defecto.

Incidente 4 — dos instancias escribiendo el mismo índice.

Un nodo perdió la red. busqueda-1 figuraba como perdido y no se sustituía, porque su volumen seguía conectado a ese nodo. Alguien forzó la eliminación para «desbloquear»:

```
$ kubectl delete pod busqueda-1 --force --grace-period=0
```

El nodo estaba incomunicado, no apagado. Durante once minutos hubo dos procesos escribiendo, y el índice quedó incoherente: consultas que devolvían documentos borrados y omitían documentos existentes.

```
procedimiento ante nodo incomunicado    forzar borrado   confirmar apagado
                                                          del nodo primero
tiempo de detección de la incoherencia    3 días          comprobación diaria
                                                          de coherencia del índice
reconstrucción necesaria                  sí, 6 h              —
documentado y ensayado                     no                  sí
```

La protección que Kubernetes ofrecía —no expulsar pods con volumen de un nodo incomunicado— funcionó correctamente, y se anuló a mano por no entender qué estaba protegiendo.

Y la revisión de la decisión.

Con los cuatro incidentes sobre la mesa, se rehizo la cuenta que no se había hecho al principio:

```
coste de infraestructura                  190 USD/mes       340 USD/mes
tiempo de operación (medido)             ~14 h/trimestre     ~1 h/trimestre
incidentes en el trimestre                      4                 0
restauración probada                           no          incluida y probada
conmutación entre zonas                  manual, 47 min    automática
parches de versión mayor                  del equipo        del proveedor
```

Ciento cincuenta dólares más al mes frente a catorce horas de trabajo trimestral y cuatro incidentes. La decisión se cambió, y lo relevante es lo que dice el registro: la decisión original nunca se había tomado; se había heredado de cuando el índice corría en una máquina.

Resumen:

```
zonas de las réplicas                          1             3
despliegue del índice                      bloqueado      2 min 40 s
política de recuperación en no producción    Delete         Retain
datos perdidos por borrado en cascada       9 días           0
incidentes por dos escritores                  1              0
volúmenes gestionados por el equipo            3              0
restauración probada con recuento              no        sí, mensual
```

La lección que esta clase traslada al resto de la parte 06: la hipótesis de la clase 072 se confirma por segunda vez. Kubernetes no resolvió el almacenamiento: renombró las restricciones de la clase 064 —un escritor, una zona, un ciclo de vida propio— y añadió dos suyas: el volumen elige zona antes que el planificador, y la identidad estable no es exclusión. Y el resultado más útil del trimestre no fue ninguna corrección técnica, sino haber hecho por fin la cuenta que decidía si ese dato tenía que estar ahí.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/077-volumes-persistentvolumes-csi-y-statefulsets/lab.py
```

El laboratorio selecciona el motor de práctica storage 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 estado-kubernetes 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 política de durabilidad, acceso, retención y costo. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

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

■ Criterio de aceptación

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

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
Un pod queda en Pending con nodos libres y el suceso menciona afinidad de volumen	El volumen se aprovisionó de forma inmediata y eligió zona antes de que existiera el pod	Usa vinculación diferida en toda clase con discos zonales; los volúmenes ya creados hay que recrearlos.
Un despliegue progresivo se bloquea con un error de conexión múltiple	Un volumen de bloque solo se conecta a un nodo, y el pod nuevo espera a que el viejo lo suelte	Estrategia de recreación para cargas con volumen de bloque, o un volumen por réplica con conjunto con estado.
Borrar un espacio de nombres elimina los discos y sus datos	La clase de almacenamiento tenía política de eliminación y el borrado es en cascada	Política de conservación en todos los entornos con datos, e instantánea previa a cualquier operación destructiva.
Dos instancias con el mismo nombre escriben el mismo dato	Se forzó la eliminación de un pod cuyo nodo estaba incomunicado pero no apagado	Confirma que el nodo está apagado antes de forzar; la protección que impide la sustitución está impidiendo dos escritores.
Al reducir réplicas quedan volúmenes que nadie usa y siguen costando	La política de retención de reclamaciones conserva por defecto	Decide esa política explícitamente y revisa periódicamente los volúmenes sin reclamación asociada.
Una restauración tarda mucho más de lo que dice el plan	Nunca se había ensayado ni cronometrado	Restaura mensualmente en una reclamación nueva, verifica con un recuento y registra la duración como tiempo de recuperación real.

■ 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. Encadena los cuatro objetos del almacenamiento y di qué falla cuando la reclamación está en Pending y cuando el pod lo está.
2. ¿Por qué la vinculación diferida evita que un pod quede atado a una zona sin nodos libres?
3. ¿Qué garantiza y qué no garantiza un conjunto con estado durante una partición de red?
4. ¿Qué ocurre con los datos al borrar el pod, la reclamación y el espacio de nombres?
5. ¿Qué costes hay que incluir al comparar un motor en el clúster con su servicio gestionado?

Referencias

• Kubernetes (2025). Persistent Volumes — reclamaciones, políticas de recuperación y modos de acceso. <https://kubernetes.io/docs/concepts/storage/persistent-volumes/>

• Kubernetes (2025). Storage Classes and volume binding mode — aprovisionamiento inmediato frente a diferido. <https://kubernetes.io/docs/concepts/storage/storage-classes/>

• Kubernetes (2025). StatefulSets — identidad, orden, plantillas de reclamación y retención. <https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>

• Kubernetes (2025). Force delete StatefulSet pods — riesgos de forzar y confirmación previa. <https://kubernetes.io/docs/tasks/run-application/force-delete-stateful-set-pod/>

• Kubernetes (2025). Volume snapshots — instantáneas, restauración y consistencia. <https://kubernetes.io/docs/concepts/storage/volume-snapshots/>

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

Descargar: Parte 06 en PDF · Manual integral

Anterior	Índice	Siguiente
← 076 · ConfigMaps, Secrets y configuración externa	Parte 06 · Programa	078 · Requests, limits, scheduling y autoscaling →

Evaluación — 077 Volumes, PersistentVolumes, CSI y StatefulSets

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 estado-kubernetes 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.

078 — Requests, limits, scheduling y autoscaling

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: capacity · Estado: EXECUTABLECORE

Propósito

Decidir cuánto pide cada carga, dónde se coloca y cómo crece, que son tres problemas distintos con tres mecanismos distintos y una relación que produce sorpresas. La clase 063 explicó qué hace un límite; aquí se añade lo que solo existe en Kubernetes: la relación entre lo solicitado y lo limitado decide a quién se sacrifica primero cuando el nodo se queda sin recursos, y una regla de colocación demasiado estricta puede impedir que el sistema crezca aunque haya capacidad de sobra.

Resultados de aprendizaje

Al finalizar podrás:

1. Clasificar una carga por su relación entre solicitud y límite, y saber qué implica en una expulsión.
2. Distinguir expulsión por presión del nodo de desalojo por prioridad, y quién decide cada una.
3. Escribir reglas de colocación que repartan sin bloquear el crecimiento.
4. Configurar escalado por métrica evitando que el propio límite impida que la métrica suba.
5. Anticipar por qué un nodo vacío no se retira y qué lo impide.

Conceptos centrales

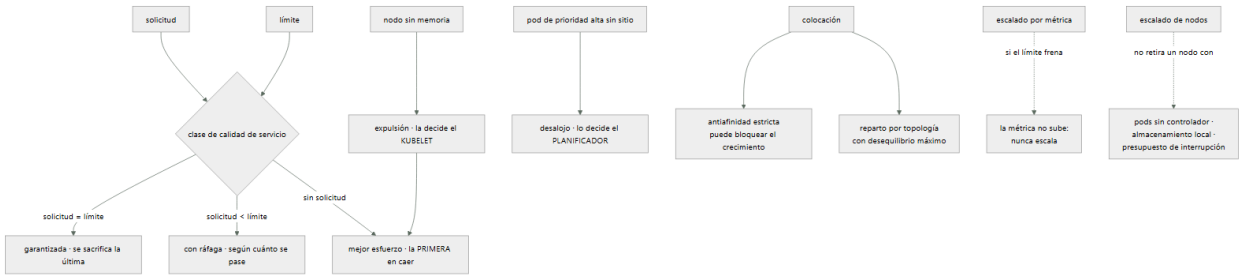
Concepto	Comprensión verificable
clase de calidad de servicio	Se deduce de la relación entre solicitud y límite. Decide el orden de sacrificio cuando el nodo se queda sin recursos, y no se declara: se calcula.
expulsión por presión	La decide el kubelet cuando al nodo le falta memoria o disco. Sacrifica primero a quien no pidió nada y a quien más se pasó de lo que pidió.
desalojo por prioridad	Lo decide el planificador: un pod de prioridad alta que no cabe echa a pods de prioridad menor. Es una decisión global, no del nodo.
regla de reparto por topología	Distribuye réplicas entre zonas o nodos con un desequilibrio máximo. Sustituye a la antiafinidad estricta, que bloquea el crecimiento.
escalado por métrica	Ajusta el número de réplicas hacia un objetivo. Si el límite de CPU frena el proceso, la métrica no sube y el escalado nunca se dispara.
sobrecompromiso	La suma de los límites de un nodo supera su capacidad. Es normal y deseable; lo que no puede superarla es la suma de las solicitudes.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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 clase de servicio no se declara: se calcula

De la relación entre lo que un pod pide y lo que se le permite sale una clasificación que nadie escribe y que decide qué ocurre en el peor momento:

```
todos los contenedores con solicitud = límite, en CPU y memoria
  → garantizada

al menos uno con solicitud, y solicitud < límite en algo
  → con ráfaga

ningún contenedor con solicitud ni límite
  → mejor esfuerzo
```

```
$ kubectl get pod tienda-7d4b9-x2k4p -o jsonpath='{.status.qosClass}'
Burstable
```

Y para qué sirve: cuando a un nodo le falta memoria, el kubelet expulsa pods antes de que el núcleo empiece a matar procesos al azar, y el orden es este:

```
1. mejor esfuerzo            los primeros, siempre
2. con ráfaga que se ha pasado de su solicitud, ordenados por cuánto se pasaron
3. garantizada               los últimos, y solo si no queda otra
```

De ahí sale la regla operativa que este programa ya insinuó en la clase 063 y aquí se concreta:

```
lo crítico          solicitud = límite en memoria: clase garantizada
lo importante       solicitud realista y límite con margen
lo prescindible     puede quedarse en mejor esfuerzo, a propósito
nada                sin solicitud por descuido
```

La última línea es la habitual: un pod sin solicitudes no es «flexible», es el primero en caer, y además engaña al planificador, que cree que no consume nada y coloca demasiadas cosas en ese nodo.

Y conviene separar dos mecanismos que se confunden porque ambos hacen desaparecer pods:

```
expulsión por presión   la decide el KUBELET, mira su nodo,
                        actúa cuando falta memoria o disco
                        criterio: clase de servicio y exceso sobre la solicitud

desalojo por prioridad  lo decide el PLANIFICADOR, mira todo el clúster,
                        actúa cuando un pod importante no cabe en ningún sitio
                        criterio: prioridad declarada
```

Las clases de prioridad son una herramienta útil y peligrosa:

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata: {name: critico}
value: 1000000
preemptionPolicy: PreemptLowerPriority
globalDefault: false
```

Y la trampa es el valor por defecto global: si una clase alta se marca como predeterminada, todo lo que no declare prioridad la hereda, con lo que un trabajo por lotes puede desalojar pods de producción. La regla es tener pocas clases, ninguna predeterminada, y que el trabajo por lotes tenga prioridad negativa:

```
crítico de plataforma      alta
producción                 media
por lotes y desarrollo     negativa: nunca desaloja, y es lo primero que se va
```

2. Colocar sin bloquear el crecimiento

Cuatro mecanismos deciden dónde va un pod, y su rigidez es lo que hay que calibrar:

```
selector de nodo       el más simple: etiqueta exacta
afinidad               igual, con expresiones y con variante PREFERIDA
marcas y tolerancias   el nodo rechaza salvo que el pod lo tolere
                       — para nodos especiales: con acelerador, con licencia
reparto por topología  distribuye entre zonas o nodos con desequilibrio máximo
```

Y el error clásico está en la antiafinidad estricta, que parece la forma correcta de repartir réplicas:

```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:      # ← estricta
      - labelSelector: {matchLabels: {app: api}}
        topologyKey: kubernetes.io/hostname
```

Eso dice «nunca dos réplicas de api en el mismo nodo». Con tres nodos y tres réplicas funciona. Al escalar a cuatro:

```
la cuarta réplica queda en Pending PARA SIEMPRE
0/3 nodes are available: 3 node(s) didn't satisfy pod anti-affinity rules
```

Y el escalado de nodos no lo arregla necesariamente, porque el nodo nuevo tarda minutos y mientras tanto el servicio no crece. Lo peor es que el fallo aparece exactamente cuando hay más carga, que es cuando se escala.

La forma moderna expresa la intención sin el bloqueo:

```
topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule        # entre zonas: exigente
    labelSelector: {matchLabels: {app: api}}
  - maxSkew: 1
    topologyKey: kubernetes.io/hostname
    whenUnsatisfiable: ScheduleAnyway       # entre nodos: preferencia
    labelSelector: {matchLabels: {app: api}}
```

La combinación dice lo que de verdad se quiere: reparte entre zonas de forma estricta, porque una zona entera es un fallo real; reparte entre nodos si puedes, y si no, colócalo igual. Con eso, escalar nunca se bloquea y el reparto sigue siendo bueno.

Y hay dos detalles que producen desequilibrios inesperados:

```
las reglas se evalúan al PLANIFICAR, no después
  → un reparto correcto se desequilibra tras expulsiones y reinicios,
    y nadie lo recoloca
nodeAffinityPolicy y nodeTaintsPolicy
  → deciden si los nodos que el pod no puede usar cuentan para el reparto
```

El primero explica por qué un servicio que se desplegó bien acaba con cuatro réplicas en un nodo y una en otro después de unas semanas. Reequilibrar exige una herramienta aparte o un despliegue.

Y las marcas y tolerancias merecen una nota porque su uso más importante no es el reparto sino la operación: un nodo que se va a retirar se marca primero, con lo que deja de recibir pods nuevos sin expulsar los que tiene. Es la base de cualquier mantenimiento ordenado, y la clase 079 lo completa con el presupuesto de interrupción.

3. Tres escalados que pueden estorbarse

Hay tres ejes y conviene no mezclarlos:

```
réplicas    escalado horizontal por métrica
tamaño      escalado vertical: ajusta solicitudes y límites
nodos       escalado del clúster: añade y retira máquinas
```

El horizontal calcula un objetivo con una fórmula simple:

```
réplicas deseadas = ceil( réplicas actuales × métrica actual / métrica objetivo )
```

Y tiene un fallo característico que une esta clase con la 063: si el límite de CPU frena el proceso, el uso nunca alcanza el objetivo.

```
objetivo: 70 % de la CPU solicitada
solicitud 500 m, límite 500 m
la carga sube, el proceso se frena al llegar al límite
el uso se queda en ~500 m = 100 % de la solicitud… pero medido sobre el LÍMITE
```

```
y con el paralelismo mal ajustado (clase 063), ni siquiera llega
→ la métrica no sube lo suficiente y el escalado no se dispara
```

La corrección tiene dos partes, y la primera es de la clase 063: ajustar el paralelismo del tiempo de ejecución a la cuota. La segunda es dar margen entre solicitud y límite para que el uso pueda subir por encima del objetivo.

Y los parámetros que evitan la oscilación, que es el otro fallo habitual:

```
behavior:
  scaleUp:
    stabilizationWindowSeconds: 0
    policies: [{type: Percent, value: 100, periodSeconds: 30}]
  scaleDown:
    stabilizationWindowSeconds: 300      # espera 5 min antes de reducir
    policies: [{type: Percent, value: 25, periodSeconds: 60}]
```

La asimetría es deliberada y es la misma conclusión de la clase 029: subir rápido y bajar despacio. Reducir deprisa deja al servicio sin margen justo cuando la carga vuelve.

El vertical ajusta las solicitudes según el uso observado, y su conflicto hay que conocerlo:

```
horizontal y vertical sobre la MISMA métrica se pelean
  el vertical sube la solicitud → el uso relativo baja → el horizontal reduce
  → oscilación
```

La combinación válida es horizontal por una métrica de negocio —peticiones por segundo, longitud de cola— y vertical para memoria, o simplemente usar el vertical en modo recomendación y aplicar sus cifras a mano, que es lo más común y lo más predecible.

El de nodos es reactivo y lento, y conviene saber sus tiempos:

```
añadir   se dispara cuando hay pods en Pending que cabrían en un nodo nuevo
         → detección + arranque de máquina + registro: de 1 a 4 minutos
retirar  cuando un nodo está infrautilizado durante un plazo
         → y solo si TODOS sus pods se pueden mover
```

Y la segunda condición es la que explica los nodos casi vacíos que nunca se retiran. Un nodo no se retira si tiene:

```
pods sin controlador que los recree                    (clase 074)
pods con almacenamiento local efímero                  (clase 064)
pods que un presupuesto de interrupción protege        (clase 079)
pods del propio sistema sin alternativa
pods con reglas de colocación que no se cumplen en otro sitio
```

```
$ kubectl -n kube-system logs deploy/cluster-autoscaler | grep -i 'scale down' | tail -3
node nodo-14 cannot be removed: pod informes-8f2 has local storage
```

Esa línea es el diagnóstico completo, y hay que buscarla antes de concluir que el escalado de nodos no funciona.

Y una alternativa que conviene conocer: en vez de grupos de nodos predefinidos, existen aprovisionadores que eligen el tipo de máquina en función de los pods pendientes. Reducen el desperdicio de forma notable y trasladan la decisión de tamaño de máquina desde una configuración estática a una respuesta a la demanda real.

4. Solicitar bien: la aritmética que casi nadie hace

La mayoría de los clústeres desperdician mucho, y no por falta de herramientas sino porque las solicitudes se ponen a ojo y nunca se revisan.

Las dos cifras que hay que mirar y casi nunca están juntas:

```
# lo que se ha reservado
$ kubectl describe node nodo-14 | sed -n '/Allocated resources/,/^$/p'
  Resource  Requests      Limits
  cpu       6400m (80%)   14200m (177%)
  memory    24Gi (75%)    41Gi (128%)

# lo que se usa de verdad
$ kubectl top node nodo-14
NAME      CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
nodo-14   1840m        23%    9100Mi          28%
```

Ochenta por ciento reservado y veintitrés usado. El nodo está lleno para el planificador y vacío de verdad, y no se puede colocar nada más en él.

La corrección es medir por carga y ajustar, con el mismo método de las clases 052 y 070:

```
$ kubectl top pods -n tienda --sort-by=cpu
# y, mejor, el percentil sobre dos semanas desde el sistema de métricas

solicitud de CPU     ≈ percentil 50-70 del uso real
límite de CPU        con margen, o ninguno (clase 068), vigilando la limitación
solicitud de memoria ≈ percentil 95-99 del uso real
límite de memoria    = solicitud, o poco más: la memoria no se comprime
```

La asimetría entre CPU y memoria es deliberada y viene de la clase 063: la CPU se frena y la memoria mata. Se puede solicitar CPU por debajo del pico porque el peor caso es lentitud; no se puede solicitar memoria por debajo del pico porque el peor caso es una terminación.

Y el sobrecompromiso que se ve en la salida de arriba —límites al 177 %— es normal y deseable: no todas las cargas alcanzan su límite a la vez. Lo que no puede superar el 100 % es la suma de las solicitudes, porque eso es lo que el planificador reparte.

Dos herramientas de gobierno que hacen sostenible esto en un clúster compartido:

```
# valores por defecto y topes por espacio de nombres
apiVersion: v1
kind: LimitRange
metadata: {name: por-defecto, namespace: tienda}
spec:
  limits:
    - type: Container
      default: {cpu: 500m, memory: 512Mi}
      defaultRequest: {cpu: 100m, memory: 256Mi}
      max: {cpu: "4", memory: 8Gi}
---
# techo total del espacio de nombres
apiVersion: v1
kind: ResourceQuota
metadata: {name: total, namespace: tienda}
spec:
  hard:
    requests.cpu: "40"
    requests.memory: 80Gi
    persistentvolumeclaims: "20"
```

El primero elimina la clase de mejor esfuerzo por descuido: cualquier contenedor sin solicitudes recibe las predeterminadas. El segundo impide que un equipo consuma la capacidad de todos, y tiene un efecto secundario que conviene anticipar: con una cuota activa, un pod sin solicitudes se rechaza, lo que obliga a declararlas y es exactamente lo que se busca.

Y el ahorro que produce esta revisión suele ser grande, porque el desperdicio se acumula: cada servicio pide de más «por si acaso» y nadie suma.

5. Ver el conjunto antes de tocar nada

Con tres escalados, cinco mecanismos de colocación y dos de gobierno, conviene tener un orden de diagnóstico para cuando algo no se coloca o no crece.

```
síntoma: un pod no arranca
  1. ¿el planificador dice por qué?      describe → sucesos, con el recuento
                                          de nodos descartados por motivo
  2. ¿es capacidad o es una regla?        "Insufficient" frente a
                                          "didn't satisfy" o "untolerated taint"
  3. si es capacidad, ¿está el escalado
     de nodos funcionando?                sus registros dicen qué evalúa

síntoma: el servicio no escala
  1. ¿el objeto de escalado ve la métrica?  describe hpa → "unknown" es
                                            el fallo más común
  2. ¿la métrica puede subir?               ¿la limita el propio límite?
  3. ¿hay tope de réplicas o cuota?

síntoma: hay nodos casi vacíos que no se retiran
  1. los registros del escalado de nodos dicen qué pod lo impide
  2. casi siempre: almacenamiento local, sin controlador,
     o presupuesto de interrupción
```

La primera comprobación del segundo bloque merece detalle porque es la que más tiempo consume:

```
$ kubectl describe hpa api
Metrics:
  resource cpu on pods (as a percentage of request):  <unknown> / 70%
Conditions:
  AbleToScale     True
  ScalingActive   False   FailedGetResourceMetric
```

<unknown> significa que no hay fuente de métricas, o que el pod no declara solicitudes —sin solicitud no hay porcentaje sobre el que calcular—. Las dos causas son frecuentes y ninguna produce un error visible en el despliegue: el objeto existe, parece configurado y no escala nunca. Sexta aparición de la familia de fallos de la clase 060.

Y tres métricas que deberían estar en el panel de cualquier clúster y rara vez están completas:

```
solicitudes reservadas frente a uso real, por nodo y por espacio de nombres
  → detecta el desperdicio antes de que alguien pida más nodos
pods en Pending por motivo
  → distingue falta de capacidad de reglas de colocación imposibles
nodos que el escalado no puede retirar, con el motivo
  → explica la factura que no baja
```

La tercera es la que convierte una discusión recurrente —«el clúster no reduce»— en un dato con nombre.

Y un cierre que conecta con la clase siguiente: todo lo de aquí decide dónde y cuántos. Lo que decide cuándo se puede mover —cuántas réplicas pueden estar fuera a la vez durante un mantenimiento o una actualización— es el presupuesto de interrupción, y es la pieza que hace que un nodo se pueda vaciar sin cortar el servicio. Sin él, todo lo anterior funciona y cualquier operación rutinaria sigue siendo arriesgada.

Ejemplo trabajado

CloudShop lleva seis meses en el clúster. La factura crece, el escalado automático no reduce nada y un incidente de memoria tumba el servicio equivocado. Los cinco hallazgos comparten causa: nadie había hecho la aritmética.

Hallazgo 1 — el nodo lleno que estaba vacío.

```
solicitudes reservadas en el clúster       78 %
uso real medio                             21 %
nodos                                      24
```

La revisión por servicio, con dos semanas de percentiles:

```
api                CPU           2000m       340m       500m
                   memoria       4Gi         1,1Gi      1,5Gi
catalogo           CPU           1000m       180m       300m
                   memoria       2Gi         720Mi      1Gi
informes           CPU           4000m      3100m      3500m
                   memoria       8Gi         6,4Gi      7Gi     ← este sí

solicitudes reservadas                        78 %            34 %
nodos                                          24              11
costo mensual de cómputo                    3.180 USD       1.460 USD
rango de límites por espacio de nombres     ninguno         definido
cuota por espacio de nombres                ninguna         definida
```

Trece nodos menos sin tocar la aplicación. El desperdicio se había acumulado servicio a servicio, cada uno pidiendo de más «por si acaso», y nadie sumaba.

Hallazgo 2 — la expulsión que se llevó al servicio equivocado.

Un nodo se quedó sin memoria y el kubelet expulsó pods. Cayó el servicio de pagos y sobrevivió un trabajo por lotes.

```
$ kubectl get pod pagos-7d4 -o jsonpath='{.status.qosClass}'
BestEffort
$ kubectl get pod lote-9c2 -o jsonpath='{.status.qosClass}'
Guaranteed
```

El servicio crítico no declaraba solicitudes —así que era el primero en la lista de sacrificio— y el trabajo por lotes las declaraba iguales al límite, con lo que era el último.

```
pagos                                  mejor esfuerzo    garantizada
trabajo por lotes                      garantizada       con ráfaga,
                                                         prioridad negativa
pods en mejor esfuerzo en producción        7                 0
```

```
rango de límites que impide pods sin
  solicitudes                              no había          activo
```

Hallazgo 3 — el escalado que nunca se disparó.

```
$ kubectl describe hpa api | grep -A1 Metrics
Metrics:
  resource cpu on pods (as a percentage of request):  <unknown> / 70%
```

Dos causas a la vez: la fuente de métricas no estaba desplegada en ese espacio de nombres, y los pods de api no declaraban solicitud de CPU, con lo que no había base para el porcentaje. El objeto existía desde hacía cuatro meses y no había escalado nunca.

```
fuente de métricas                       parcial          en todo el clúster
solicitudes declaradas                      no                sí
objetivo de escalado                     CPU 70 %      peticiones por segundo
ventana de estabilización al reducir     por defecto        300 s
veces que escaló en 4 meses                   0            41 en el primero
```

El cambio de métrica fue el importante: con el límite de CPU frenando el proceso (clase 063), el uso nunca llegaba al objetivo aunque la latencia se disparara.

Hallazgo 4 — la cuarta réplica que nunca arrancaba.

```
0/11 nodes are available: 11 node(s) didn't satisfy pod anti-affinity rules.
```

Antiafinidad estricta por nodo con once nodos y once réplicas ya colocadas. Al escalar a doce, la última quedaba pendiente para siempre — y ocurría en los picos, que es cuando el escalado se dispara.

```
regla                          antiafinidad estricta   reparto por topología
zonas                              sin regla          desequilibrio máximo 1,
                                                       estricto
nodos                          nunca dos por nodo     preferencia, no bloqueo
réplicas bloqueadas en los picos       1-3                  0
```

Hallazgo 5 — nueve nodos al 12 % que no se retiraban.

```
$ kubectl -n kube-system logs deploy/cluster-autoscaler | grep 'cannot be removed' \
  | sed 's/.*pod //' | awk '{print $2, $3, $4}' | sort | uniq -c
      6 has local storage
      3 not replicated
```

Seis nodos retenidos por pods con almacenamiento local efímero —un directorio temporal declarado como volumen del nodo— y tres por pods creados sueltos, sin controlador (clase 074).

```
pods con almacenamiento local del nodo         6            0 (montaje en memoria)
pods sin controlador                           3            0
nodos que el escalado no podía retirar         9            0
nodos en horas valle                          11            4
costo mensual tras esta corrección        1.460 USD        980 USD
```

Resumen:

```
solicitudes reservadas                          78 %           34 %
nodos en horas valle                             24              4
pods en mejor esfuerzo en producción              7              0
escalados automáticos en 4 meses                  0             41/mes
réplicas bloqueadas por reglas de colocación    1-3              0
nodos que el escalado no podía retirar            9              0
costo mensual de cómputo                   3.180 USD        980 USD
```

La lección que esta clase traslada al resto de la parte 06: los cinco hallazgos existían desde el primer día y ninguno producía un error. El clúster funcionaba, el objeto de escalado existía, las reglas de colocación estaban puestas y la factura crecía. Lo único que faltaba era sumar: solicitado frente a usado, réplicas frente a nodos disponibles, y qué pod impide retirar cada máquina. Las tres cuentas caben en tres órdenes, y ninguna estaba en ningún panel.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/078-requests-limits-scheduling-y-autoscaling/lab.
py
```

El laboratorio selecciona el motor de práctica capacity 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 capacidad-kubernetes en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es requests, límites y una decisión de escalado medida. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye capacidad-kubernetes 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 expulsión por falta de memoria se lleva el servicio crítico y respeta a un trabajo por lotes	El crítico no declaraba solicitudes —clase de mejor esfuerzo— y el lote sí	Solicitud igual a límite en memoria para lo crítico, prioridad negativa para el lote y un rango de límites que impida pods sin solicitudes.
El escalado por métrica no se dispara nunca	No hay fuente de métricas o los pods no declaran solicitudes, así que la métrica es desconocida	Comprueba el estado del objeto de escalado; declara solicitudes y escala por una métrica de negocio si el límite de CPU frena el proceso.
Al escalar, una réplica queda en Pending para siempre	Antiafinidad estricta por nodo con tantas réplicas como nodos	Usa reparto por topología: estricto entre zonas y como preferencia entre nodos.
Hay nodos casi vacíos que el escalado no retira	Contienen pods con almacenamiento local, sin controlador o protegidos por un presupuesto de interrupción	Lee los registros del escalado, que nombran el pod concreto, y corrige la causa en la carga.
El clúster está lleno para el planificador y vacío en uso real	Las solicitudes se pusieron a ojo y nunca se revisaron	Ajusta con percentiles medidos: CPU sobre el percentil medio, memoria sobre el alto, y pon cuotas por espacio de nombres.
Un trabajo por lotes desaloja pods de producción	Una clase de prioridad alta está marcada como predeterminada global	Ninguna clase predeterminada, pocas clases, y prioridad negativa para lo prescindible.

■ 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. ¿Cómo se calcula la clase de calidad de servicio y en qué orden se sacrifican las cargas?
2. ¿Qué diferencia hay entre expulsión por presión y desalojo por prioridad, y quién decide cada una?
3. ¿Por qué la antiafinidad estricta bloquea el crecimiento y qué la sustituye?
4. ¿Por qué el escalado por CPU puede no dispararse nunca aunque el servicio esté saturado?
5. Enumera tres motivos por los que un nodo casi vacío no se retira, y cómo se averigua cuál es.

Referencias

• Kubernetes (2025). Quality of Service classes — cómo se calculan y su papel en la expulsión. <https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/>

• Kubernetes (2025). Node-pressure eviction — señales del kubelet y orden de sacrificio. <https://kubernetes.io/docs/concepts/scheduling-eviction/node-pressure-eviction/>

• Kubernetes (2025). Pod topology spread constraints — desequilibrio máximo y comportamiento al no poder cumplirse. <https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/>

• Kubernetes (2025). Horizontal Pod Autoscaler — fórmula, comportamiento y ventanas de estabilización. <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

• Kubernetes autoscaler (2025). Cluster Autoscaler FAQ — qué impide retirar un nodo. <https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md>

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

Descargar: Parte 06 en PDF · Manual integral

Anterior	Índice	Siguiente
← 077 · Volumes, PersistentVolumes, CSI y StatefulSets	Parte 06 · Programa	079 · Probes, rollouts, rollback y PodDisruptionBudget →

Evaluación — 078 Requests, limits, scheduling y autoscaling

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 capacidad-kubernetes 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.

079 — Probes, rollouts, rollback y PodDisruptionBudget

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: reliability · Estado: EXECUTABLECORE

Propósito

Hacer que las operaciones rutinarias —desplegar, vaciar un nodo, actualizar el clúster— no corten el servicio, que es lo que separa un clúster que funciona de uno que se puede operar. La clase 068 dejó el contrato de la aplicación; aquí se añade el objeto que Kubernetes aporta y que nadie configura bien a la primera: el presupuesto de interrupción, que protege de las operaciones voluntarias y de nada más, y que mal puesto impide actualizar el clúster durante semanas sin que nadie relacione una cosa con la otra.

Resultados de aprendizaje

Al finalizar podrás:

1. Traducir las tres comprobaciones de la clase 068 a su forma en Kubernetes y calcular sus plazos.
2. Distinguir interrupción voluntaria de involuntaria y saber qué protege un presupuesto.
3. Fijar un presupuesto que permita vaciar nodos sin cortar el servicio.
4. Vaciar un nodo de forma ordenada y diagnosticar qué lo bloquea.
5. Reconocer qué no deshace una vuelta atrás y cómo se compensa.

Conceptos centrales

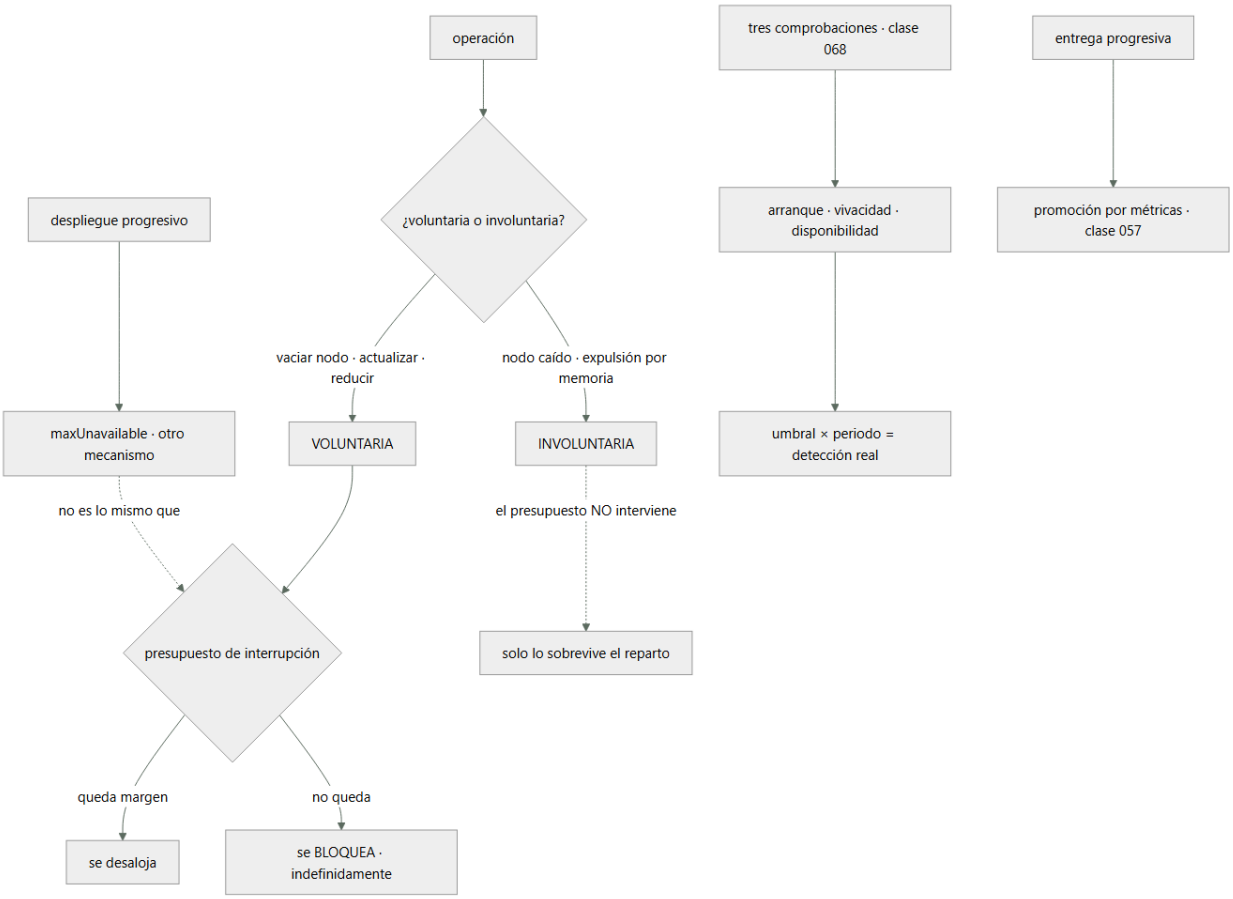
Concepto	Comprensión verificable
interrupción voluntaria	La que provoca alguien: vaciar un nodo, actualizar, reducir el clúster. Es la única que un presupuesto de interrupción puede frenar.
interrupción involuntaria	La que ocurre sola: un nodo que se cae, una expulsión por falta de memoria. Ningún presupuesto la evita, y es la que hay que sobrevivir con réplicas repartidas.
presupuesto de interrupción	Cuántas réplicas pueden estar fuera a la vez por una operación voluntaria. Mal puesto, bloquea toda operación de mantenimiento de forma indefinida.
vaciado de nodo	Marcar el nodo para que no reciba pods y desalojar los que tiene, respetando los presupuestos. Es la base de cualquier mantenimiento.
umbral de fallo	Número de comprobaciones fallidas antes de actuar. Multiplicado por el periodo da el tiempo real de detección, que casi nunca es el que se cree.
entrega progresiva	Aumentar el tráfico de una versión nueva según métricas y no según el reloj, con vuelta atrás automática si el presupuesto de error se consume.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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

■ Flujo de razonamiento



Desarrollo

1. Las tres comprobaciones, con su aritmética

La clase 068 estableció qué pregunta responde cada comprobación. En Kubernetes se declaran así, y lo que hay que calcular es el tiempo real de cada decisión:

```
startupProbe:
  httpGet: {path: /healthz, port: 8080}
  periodSeconds: 5
  failureThreshold: 30        # tolera 150 s de arranque
livenessProbe:
  httpGet: {path: /healthz, port: 8080}
  periodSeconds: 10
  timeoutSeconds: 3
  failureThreshold: 3         # detecta un atasco en 30 s
readinessProbe:
  httpGet: {path: /readyz, port: 8080}
  periodSeconds: 5
  timeoutSeconds: 2
  failureThreshold: 2         # sale de rotación en 10 s
  successThreshold: 1
```

```
tiempo real de detección = periodo × umbral de fallo
y hay que sumarle el tiempo de espera si la comprobación se cuelga
```

Dos errores de aritmética que producen incidentes:

```
tiempo de espera mayor que el periodo
  una comprobación cada 5 s con espera de 10 s se solapa consigo misma
  → bajo carga, fallos que no corresponden a nada

umbral de vivacidad demasiado bajo
  2 fallos con periodo de 5 s reinicia a los 10 s
  → una pausa de recolección de basura de 12 s reinicia el proceso
```

El segundo es especialmente traicionero porque empeora bajo carga, que es cuando menos conviene reiniciar. La regla práctica: la vivacidad debe tolerar la peor pausa conocida del tiempo de ejecución, con margen.

Los cuatro tipos de sonda, y cuándo usa cada uno:

```
HTTP        lo habitual; el código 200-399 es sano
TCP         solo comprueba que el puerto acepta: sirve para lo que no habla HTTP
ejecución   ejecuta una orden dentro; cuesta más y no existe en imágenes
            sin intérprete (clase 062)
gRPC        nativa, sin necesidad de un binario auxiliar
```

La tercera merece una advertencia doble: consume recursos en cada ejecución y no funciona en una imagen mínima. Es una de las pocas fricciones reales del endurecimiento de la parte 05, y la salida es la sonda nativa del protocolo o un punto HTTP mínimo en la aplicación.

Y el apagado ordenado de la clase 068, en su forma de Kubernetes:

```
lifecycle:
  preStop:
    exec: {command: ["/bin/sleep", "6"]}
terminationGracePeriodSeconds: 45
```

Con la cifra que la clase 075 midió: la espera depende del tamaño del clúster, y hay que volver a medirla cuando crece. Aquí la única aportación nueva es que el gancho previo a la parada se ejecuta antes de la señal, así que su duración se resta del plazo de gracia:

```
plazo de gracia = duración del gancho previo + tiempo de apagado del proceso
45 s = 6 s de espera + hasta 39 s para terminar peticiones y cerrar
```

Si el gancho tarda más que el plazo, el proceso recibe la señal de matar sin haber recibido nunca la de parada.

2. Voluntaria e involuntaria: qué protege el presupuesto

Esta distinción es la que hace útil el objeto y la que casi nadie hace:

```
VOLUNTARIA        la provoca alguien
  vaciar un nodo para mantenimiento
  actualizar la versión del clúster
  reducir el número de nodos
  → el presupuesto de interrupción PUEDE frenarla

INVOLUNTARIA      ocurre sola
  el nodo se apaga o pierde la red
  expulsión por falta de memoria (clase 078)
  el proceso muere
  → el presupuesto NO interviene en absoluto
```

De ahí se sigue lo que el objeto no hace, y conviene decirlo porque se espera de él:

```
no impide que un nodo se caiga
no garantiza disponibilidad
no sustituye a repartir réplicas entre zonas
no controla el despliegue progresivo, que usa otro mecanismo
```

La última confunde con frecuencia: durante un despliegue, quien decide cuántas réplicas pueden faltar es maxUnavailable de la estrategia (clase 074), no el presupuesto. Son dos controles distintos para dos situaciones distintas, y hay que configurar ambos de forma coherente.

El objeto se declara así:

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata: {name: api}
spec:
  minAvailable: 2                 # o maxUnavailable: 1
  selector: {matchLabels: {app: api}}
```

Y tiene dos formas de estar mal puesto, con consecuencias opuestas.

Demasiado estricto: bloquea el mantenimiento para siempre.

```
réplicas: 2 · minAvailable: 2
  → no se puede desalojar ninguna réplica, nunca
  → vaciar un nodo se queda esperando indefinidamente
  → la actualización del clúster no avanza
```

```
   → el escalado de nodos no puede reducir (clase 078)
```

Y lo peor es que no produce ningún error visible: la operación simplemente no termina, y quien la lanzó la interrumpe suponiendo que va lenta.

```
$ kubectl get pdb api
NAME   MIN AVAILABLE   ALLOWED DISRUPTIONS   AGE
api    2               0                     94d
```

La columna de interrupciones permitidas en cero durante noventa y cuatro días es el diagnóstico completo. Debería ser una alerta.

Demasiado laxo o sin efecto: no protege nada.

```
selector: {matchLabels: {app: api}}      # y los pods llevan app.kubernetes.io/name
```

Un selector que no coincide con ningún pod crea un presupuesto que protege un conjunto vacío. El objeto existe, figura en el inventario y no hace nada — séptima aparición de la familia de fallos de la clase 060.

```
$ kubectl get pdb api -o jsonpath='{.status.currentHealthy} de {.status.expectedPods}'
0 de 0                                                                    ✗
```

La regla que evita los dos extremos:

```
réplicas ≥ 3   →  maxUnavailable: 1     permite vaciar de uno en uno
réplicas = 2   →  maxUnavailable: 1     acepta operar con una durante el vaciado
réplicas = 1   →  NO poner presupuesto  y asumir que el mantenimiento corta;
                  si no se puede asumir, hacen falta más réplicas
```

Expresarlo como máximo indisponible en vez de como mínimo disponible tiene una ventaja: sigue siendo correcto al escalar. Un mínimo de 2 con 10 réplicas permite vaciar 8 a la vez, que casi nunca es lo que se quiere.

3. Vaciar un nodo, y qué lo bloquea

El mantenimiento de un nodo tiene una secuencia fija:

```
$ kubectl cordon nodo-14                      # deja de recibir pods nuevos
$ kubectl drain nodo-14 \
    --ignore-daemonsets \
    --delete-emptydir-data \
    --timeout=600s
# … mantenimiento …
$ kubectl uncordon nodo-14
```

Y las dos opciones del vaciado son avisos disfrazados:

```
--ignore-daemonsets      los pods de conjunto de demonios no se desalojan
                         porque volverían a crearse en el mismo nodo
--delete-emptydir-data   hay pods con datos en almacenamiento efímero del nodo
                         y esos datos SE VAN A PERDER
```

La segunda merece pararse: la orden pide confirmación explícita porque va a destruir datos. Si aparece, alguien tiene datos donde la clase 064 dijo que no debía tenerlos.

Y el vaciado se bloquea por cuatro motivos, que son los mismos que impiden reducir el clúster (clase 078):

```
un presupuesto sin margen              lo más frecuente
pods sin controlador que los recree     nadie los recrearía en otro sitio
pods con almacenamiento local           los datos se perderían
ninguna capacidad libre en otros nodos  el pod desalojado no cabe en ningún sitio
```

El diagnóstico es directo y conviene conocerlo porque el mensaje es claro:

```
$ kubectl drain nodo-14 --dry-run=server
error: cannot delete Pods with local storage (use --delete-emptydir-data): tienda/informes-8f2
error: cannot delete Pods not managed by ReplicationController… : tienda/depuracion-manual
```

El --dry-run=server antes de cualquier mantenimiento es la comprobación que evita descubrir el bloqueo a mitad de una ventana.

Y la actualización del clúster es exactamente esto repetido por cada nodo, con dos matices que definen su duración:

```
nodos que se actualizan a la vez     configurable; con uno cada vez y 40 nodos,
                                     una actualización tarda horas
un presupuesto sin margen bloquea    el nodo 7 de 40, y la operación se detiene
                                     ahí, con el clúster a medias
```

La segunda situación es la peor de esta clase: un clúster con la mitad de los nodos en una versión y la mitad en otra, parado indefinidamente, porque un presupuesto de un servicio no deja avanzar. Y como el bloqueo no produce error, se descubre cuando alguien mira.

La comprobación preventiva, que debería ejecutarse antes de cualquier actualización:

```
$ kubectl get pdb -A -o custom-columns=\
NS:.metadata.namespace,NOMBRE:.metadata.name,PERMITIDAS:.status.disruptionsAllowed,\
SANOS:.status.currentHealthy,ESPERADOS:.status.expectedPods \
  | awk 'NR==1 || $3==0 || $5==0'
```

Esa orden lista dos cosas a la vez: los presupuestos que bloquean —cero interrupciones permitidas— y los que no protegen nada —cero pods esperados—. Las dos son incidentes latentes.

Y una precaución sobre los nodos gestionados por el proveedor: en las plataformas de la clase 083, la actualización de nodos la inicia el proveedor y respeta los presupuestos igual. Un presupuesto bloqueante puede dejar una actualización de seguridad del proveedor sin aplicar durante semanas, con el nodo ejecutando una versión con vulnerabilidades conocidas. Es la misma consecuencia con otro origen, y suele descubrirse en una auditoría.

4. Lo que una vuelta atrás no deshace

La clase 074 dejó el mecanismo: volver atrás es reescalar el conjunto de réplicas anterior, y funciona en segundos si la imagen está referenciada por huella. Lo que hay que añadir es qué queda fuera de ese mecanismo.

```
la vuelta atrás DESHACE
  la versión del código y su configuración de plantilla

la vuelta atrás NO deshace
  las migraciones de esquema ya aplicadas
  los mensajes ya publicados con el formato nuevo
  los ficheros ya escritos con la estructura nueva
  los datos ya modificados por la versión nueva
```

De ahí que la disciplina de la clase 071 —cambios compatibles en ambos sentidos— no sea una recomendación sino el requisito que hace posible la vuelta atrás:

```
expandir     añadir la columna nueva, que la versión vieja ignora
desplegar    la versión nueva escribe en ambas
esperar      hasta que la vuelta atrás ya no sea plausible
contraer     retirar lo viejo
```

Saltarse el paso de espera es lo que convierte una vuelta atrás de treinta segundos en un incidente de datos. Y la pregunta que hay que responder antes de cada despliegue es concreta: «si vuelvo atrás dentro de una hora, ¿qué queda roto?». Si la respuesta no es «nada», el despliegue no es reversible y hay que tratarlo como tal.

Y para reducir el riesgo de los despliegues que sí son reversibles, la entrega progresiva conecta con la clase 057:

```
despliegue por tiempo      10 % · esperar 10 min · 50 % · esperar · 100 %
                           el reloj no sabe si algo va mal

despliegue por métricas    10 % · comparar tasa de error y latencia con
                           la versión estable · promover o revertir
                           → la decisión la toma un dato
```

El mecanismo se apoya en el reparto por peso de la clase 075 y en el presupuesto de error de la clase 057, y su valor no es la automatización sino el criterio escrito:

```
promover si     tasa de error del canario ≤ la de la estable
                percentil 95 dentro del 10 % de la estable
                sin excepciones nuevas en el agrupador de errores
revertir si     cualquiera de los tres falla
```

Es el mismo criterio que la clase 071 exigía escribir antes de empezar una migración, aplicado a cada despliegue. Y tiene el mismo beneficio: durante un despliegue todo el mundo tiene prisa, y un criterio decidido de antemano es lo único que resiste esa prisa.

Y dos operaciones del despliegue que conviene conocer:

```
$ kubectl rollout pause deploy/api      # congela el avance sin revertir
$ kubectl rollout resume deploy/api
```

Pausar es útil para observar un despliegue a medias con tráfico real en ambas versiones — un canario improvisado— y tiene un riesgo: un despliegue pausado y olvidado deja el servicio con dos versiones indefinidamente. Merece una alerta sobre despliegues que llevan demasiado tiempo sin completarse.

5. La lista que hace operable un clúster

Reuniendo lo de esta clase con las 068, 074 y 075, la lista por servicio queda así:

- tres comprobaciones separadas, con la de vivacidad sin dependencias
- umbral por periodo calculado, y tiempo de espera menor que el periodo
- la vivacidad tolera la peor pausa conocida del tiempo de ejecución
- gancho previo a la parada mayor que la propagación medida (clase 075)
- plazo de gracia = gancho + apagado del proceso, con margen
- maxUnavailable en cero para servicios sin volumen de bloque
- presupuesto de interrupción con máximo indisponible, no mínimo disponible
- el selector del presupuesto coincide de verdad con los pods
- interrupciones permitidas mayores que cero, vigilado
- despliegue reversible, o declarado como no reversible
- criterios de promoción y de reversión escritos antes

Y las tres comprobaciones a nivel de clúster que conviene automatizar:

```
# 1. presupuestos que bloquean o que no protegen
$ kubectl get pdb -A -o json | jq -r '.items[]
  | select(.status.disruptionsAllowed == 0 or .status.expectedPods == 0)
  | "\(.metadata.namespace)/\(.metadata.name) permitidas=\(.status.disruptionsAllowed) esperados=\(.
status.expectedPods)"'

# 2. servicios sin presupuesto con más de una réplica
$ kubectl get deploy -A -o json | jq -r '.items[]
  | select(.spec.replicas > 1) | "\(.metadata.namespace)/\(.metadata.name)"' \
  > con-replicas.txt

# 3. despliegues que llevan demasiado sin completarse
$ kubectl get deploy -A -o json | jq -r '.items[]
  | select(.status.conditions[]? | select(.type=="Progressing" and .status=="False"))
  | "\(.metadata.namespace)/\(.metadata.name)"'
```

Y el ensayo que demuestra que todo lo anterior funciona, que es el equivalente al despliegue con carga de la clase 068:

```
# vaciar un nodo con carga en curso
$ hey -z 5m -c 50 -q 20 https://tienda.example/api/pedidos > carga.txt &
$ kubectl drain nodo-14 --ignore-daemonsets --timeout=600s
$ kubectl uncordon nodo-14
$ grep -E '\[502\]|\[503\]|errors' carga.txt
```

Cero errores al vaciar un nodo bajo carga es el criterio, y hay que medirlo como se mide el despliegue. Un clúster donde vaciar un nodo produce errores es un clúster donde cada actualización de seguridad tiene un coste de disponibilidad, y eso lleva a retrasar actualizaciones — que es como se acumulan las vulnerabilidades de la clase 067.

Ejemplo trabajado

CloudShop lleva cuatro meses sin actualizar el clúster. La actualización pendiente incluye una corrección de seguridad, y cada intento se queda a medias. Los cuatro hallazgos explican por qué, y el último cambia cómo se despliega.

Hallazgo 1 — la actualización se detenía en el nodo 7.

```
$ kubectl drain nodo-07 --ignore-daemonsets --dry-run=server
Cannot evict pod as it would violate the pod's disruption budget: tienda/pagos
```

```
$ kubectl get pdb -A -o custom-columns=NS:.metadata.namespace,N:.metadata.name,\
PERMITIDAS:.status.disruptionsAllowed | awk '$3==0'
tienda    pagos      0
tienda    informes   0
plataforma  registro 0
```

Tres presupuestos con cero interrupciones permitidas. El de pagos llevaba 94 días así:

```
réplicas: 2 · minAvailable: 2 → nunca se puede desalojar ninguna
```

```
expresión del presupuesto              minAvailable: 2   maxUnavailable: 1
réplicas de pagos                             2                 3
interrupciones permitidas                     0                 1
```

```
días sin poder actualizar el clúster        124                —
duración de la actualización completa   no terminaba      3 h 40 min
alerta sobre presupuestos bloqueantes     ninguna        permitidas = 0 → aviso
```

Ciento veinticuatro días con una corrección de seguridad sin aplicar, porque un objeto de dos líneas bloqueaba en silencio.

Hallazgo 2 — dos presupuestos que no protegían nada.

```
$ kubectl get pdb -A -o json | jq -r '.items[] | select(.status.expectedPods == 0)
  | "\(.metadata.namespace)/\(.metadata.name)"'
tienda/catalogo
tienda/busqueda
```

Los selectores usaban app: y las plantillas habían migrado a app.kubernetes.io/name: — el mismo desajuste de etiquetas que la clase 075 encontró en un servicio.

```
presupuestos con selector sin coincidencias    2                0
convención de etiquetas                     mezclada      una sola, validada
                                                           en la admisión
comprobación en la canalización             ninguna    falla si esperados = 0
```

Los dos servicios llevaban meses sin ninguna protección, con el objeto creado y visible en el inventario.

Hallazgo 3 — vaciar un nodo cortaba el servicio.

Con los presupuestos corregidos, el primer vaciado con carga en curso:

```
errores durante el vaciado de nodo-07     1.284
```

Dos causas a la vez:

```
$ kubectl get deploy api -o jsonpath='{.spec.template.spec.terminationGracePeriodSeconds}'
30
$ kubectl get deploy api -o jsonpath='{.spec.template.spec.containers[0].lifecycle}'
(vacío)
```

No había gancho previo a la parada —la corrección de la clase 075 se había aplicado solo a tienda, no a api— y el plazo de gracia era menor que la petición más larga.

```
gancho previo a la parada                  ninguno            6 s
plazo de gracia                             30 s              45 s
errores al vaciar un nodo bajo carga        1.284               0
ensayo de vaciado en la verificación       no había      obligatorio, trimestral
```

Hallazgo 4 — la vuelta atrás que no pudo volver.

Un despliegue con un error de cálculo se revirtió a los veinte minutos. La vuelta atrás tardó 31 segundos y el problema persistió:

```
la versión nueva había aplicado una migración que renombraba una columna
la versión anterior consultaba el nombre viejo
→ la vuelta atrás dejó el código antiguo contra el esquema nuevo
```

```
disciplina de cambios de esquema        renombrar       expandir y contraer
pregunta previa al despliegue           no se hacía   "si vuelvo atrás en 1 h,
                                                       ¿qué queda roto?"
despliegues declarados no reversibles         0        3 en el trimestre,
                                                       con procedimiento propio
tiempo de recuperación de ese incidente   2 h 10 min          —
```

Los tres despliegues declarados no reversibles son cambios de esquema que no se pueden hacer compatibles, y tienen su propio procedimiento: ventana anunciada, copia previa verificada y plan de reversión por restauración, no por reescalado.

Y el cambio de método que salió de todo esto.

```
promoción de un despliegue            por reloj        por métricas frente a
                                                        la versión estable
criterios escritos antes                 no                sí, tres
reversiones automáticas                   0            4 en el trimestre,
                                                        todas correctas
tiempo medio de detección de un
  despliegue malo                      18 min             2 min 40 s
```

Las cuatro reversiones automáticas son el dato interesante: cuatro despliegues que habrían llegado al 100 % del tráfico y se detuvieron en el 10 %.

Resumen:

```
días sin poder actualizar el clúster           124              0
presupuestos que bloquean                         3              0
presupuestos que no protegen nada                 2              0
errores al vaciar un nodo bajo carga          1.284              0
despliegues con criterio escrito                  0          todos
reversiones automáticas por métricas              0              4
tiempo de detección de un despliegue malo     18 min       2 min 40 s
```

La lección que esta clase traslada al resto de la parte 06: los dos hallazgos más caros —124 días sin actualizar y dos servicios sin protección— son el mismo tipo de fallo con signos opuestos, y ninguno producía un error. Un presupuesto demasiado estricto bloquea en silencio y uno mal dirigido protege en silencio a nadie. La comprobación que detecta ambos es una sola orden, y no estaba en ningún panel: interrupciones permitidas y pods esperados, las dos columnas, buscando ceros.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/079-probes-rollouts-rollback-y-
poddisruptionbudget/lab.py
```

El laboratorio selecciona el motor de práctica reliability 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 despliegue-resiliente 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 escenario de fallo con objetivo y recuperación medida. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye despliegue-resiliente 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 actualización del clúster se queda a medias y no da ningún error	Un presupuesto de interrupción sin margen bloquea el vaciado de un nodo	Expresa el presupuesto como máximo indisponible, ten al menos tres réplicas en lo crítico y alerta cuando las interrupciones permitidas sean cero.
Un presupuesto existe y el servicio se queda sin réplicas en un mantenimiento	El selector no coincide con ninguna etiqueta de los pods	Comprueba que los pods esperados son mayores que cero y unifica la convención de etiquetas con validación en la admisión.
Vaciar un nodo produce errores aunque el despliegue no los produzca	Falta el gancho previo a la parada o el plazo de gracia es menor que la petición más larga	Aplica la corrección de la clase 068 a todos los servicios y ensaya el vaciado con carga como parte de la verificación.
Una vuelta atrás se completa y el problema persiste	La versión nueva aplicó un cambio de esquema que la anterior no entiende	Expandir y contraer con espera intermedia; y si no es posible, declarar el despliegue como no reversible con su propio procedimiento.
Un proceso se reinicia bajo carga sin motivo aparente	El umbral de la comprobación de vivacidad no tolera la peor pausa del tiempo de ejecución	Calcula periodo por umbral con margen sobre esa pausa, y asegúrate de que el tiempo de espera cabe en el periodo.
Un servicio lleva días con dos versiones sirviendo a la vez	Un despliegue quedó pausado y nadie lo retomó	Alerta sobre despliegues que llevan demasiado tiempo sin completarse.

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 interrupciones frena un presupuesto y cuáles no, y qué implica eso para la disponibilidad real?
2. ¿Por qué expresar el presupuesto como máximo indisponible es mejor que como mínimo disponible?
3. ¿Qué cuatro motivos bloquean el vaciado de un nodo y con qué orden se comprueban antes de un mantenimiento?
4. ¿Qué no deshace una vuelta atrás, y qué disciplina la hace posible?
5. Calcula el tiempo real de detección de una vivacidad con periodo 10 s, umbral 3 y espera 3 s.

Referencias

- Kubernetes (2025). Disruptions — voluntarias e involuntarias, y alcance del presupuesto. <https://kubernetes.io/docs/concepts/workloads/pods/disruptions/>
- Kubernetes (2025). Specifying a Disruption Budget — expresión, selector y estado. <https://kubernetes.io/docs/tasks/run-application/configure-pdb/>
- Kubernetes (2025). Safely drain a node — secuencia, opciones y bloqueos. <https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node/>
- Kubernetes (2025). Configure probes — tipos, umbrales y tiempos de espera. <https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>
- Google (2018). The Site Reliability Workbook, cap. 16 — despliegue progresivo y criterios de promoción. <https://sre.google/workbook/canarying-releases/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Anterior	Índice	Siguiente
← 078 · Requests, limits, scheduling y autoscaling	Parte 06 · Programa	080 · Namespaces, RBAC, NetworkPolicy y admission →

Evaluación — 079 Probes, rollouts, rollback y PodDisruptionBudget

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 despliegue-resiliente 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.

080 — Namespaces, RBAC, NetworkPolicy y admission

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: security · Estado: EXECUTABLECORE

Propósito

Acotar quién puede hacer qué y quién puede hablar con quién dentro del clúster, que es la tercera fuga de la clase 072. Dos valores por defecto explican casi todos los problemas: todos los pods pueden alcanzar a todos los pods, en cualquier espacio de nombres, hasta que alguien escribe una política; y una política de red no hace nada si el complemento de red no la implementa, con el objeto creado y visible. La clase cierra además el patrón que este programa ha visto en cuatro plataformas: el permiso suma y lo que resta vive en otro sistema.

Resultados de aprendizaje

Al finalizar podrás:

1. Delimitar qué acota un espacio de nombres y qué no, y por qué no es una frontera de seguridad fuerte.
2. Conceder permisos contando los caminos indirectos, no solo los directos.
3. Escribir políticas de red sabiendo que seleccionar un pod lo cambia a denegar por defecto.
4. Comprobar que las políticas se están aplicando de verdad, con una prueba negativa.
5. Imponer el endurecimiento de la clase 069 con admisión en vez de con revisiones manuales.

Conceptos centrales

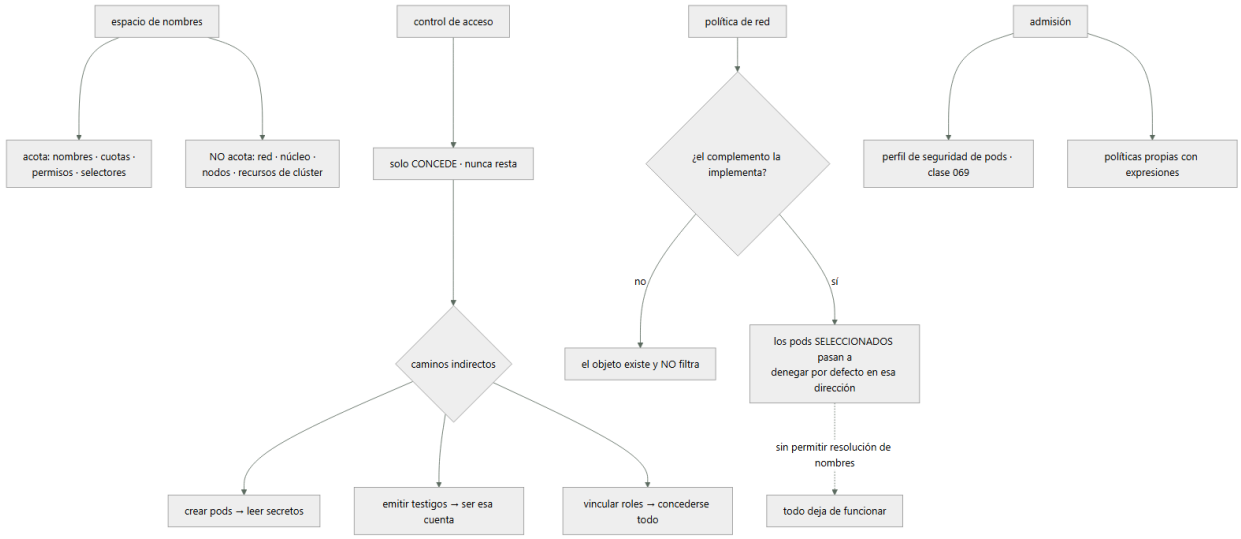
Concepto	Comprensión verificable
espacio de nombres	Ámbito para nombres, cuotas, permisos y selectores. No aísla la red ni el núcleo: es una frontera administrativa, no una de seguridad fuerte.
permiso aditivo	El control de acceso solo concede; no existe una regla que reste. Cuarta plataforma del programa con la misma propiedad, y la misma consecuencia: lo que acota vive en otro sistema.
camino indirecto	Permiso que concede acceso a algo distinto de lo que nombra: crear pods da acceso a los secretos del espacio; emitir testigos da la identidad de una cuenta de servicio.
denegar por defecto por selección	Una política de red no añade denegaciones al clúster: cambia a denegar por defecto los pods que selecciona, y solo en la dirección que declara.
complemento de red que no implementa	Si el complemento instalado no soporta políticas, el objeto se crea, se lista y no filtra nada. No hay ningún error.
admisión de seguridad de pods	Mecanismo integrado que rechaza o avisa sobre pods que no cumplen un perfil. Es el endurecimiento de la clase 069 impuesto por la plataforma.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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

■ Flujo de razonamiento



Desarrollo

1. Qué acota un espacio de nombres y qué no

El espacio de nombres se usa como si fuera una frontera de seguridad y no lo es. Lo que sí acota:

```
nombres de objetos          dos servicios pueden llamarse igual en espacios distintos
cuotas y rangos de límites  clase 078
permisos                    un rol vale dentro de su espacio
selectores                  las políticas de red y los presupuestos seleccionan dentro
secretos y configuración    y quien pueda crear pods ahí, los lee (clase 076)
```

Lo que no acota, y es donde está el malentendido:

```
la RED           por defecto, cualquier pod alcanza a cualquier pod
                 de cualquier espacio de nombres
el NÚCLEO        es uno solo para el nodo (clase 063)
los nodos        dos espacios pueden compartir la misma máquina
los recursos de clúster   nodos, volúmenes, definiciones de tipos, roles de clúster
```

La primera línea sorprende siempre y merece comprobarse una vez:

```
$ kubectl run t --rm -it -n desarrollo --image=nicolaka/netshoot -- \
    curl -s -m 3 http://bd.produccion.svc:5432 -o /dev/null -w '%{http_code}\n'
000                     # conecta: el puerto responde, no habla HTTP
```

Un pod de desarrollo alcanzando la base de datos de producción sin ninguna configuración especial. Es el equivalente de AllowVnetInBound de la clase 039 y de la ausencia de segmentación de la 051: la conectividad es el estado inicial y el aislamiento es trabajo explícito. Tercera plataforma con la misma propiedad.

De ahí se deduce cómo repartir espacios de nombres, que es por quién los administra y qué políticas necesitan, no por comodidad:

```
uno por equipo y entorno        permisos, cuotas y políticas distintas
plataforma aparte               componentes compartidos, con permisos propios
lo no confiable, en otro sitio  ver más abajo
```

Y la honestidad sobre el aislamiento entre inquilinos, que conviene decir con claridad:

```
espacios de nombres         suficiente entre equipos de la misma organización
                            que ejecutan código revisado
no suficiente               para código de terceros o de clientes:
                            comparten núcleo, nodos y plano de control
para eso                    clústeres separados, o aislamiento reforzado
                            (máquinas ligeras, núcleo en espacio de usuario,
                             clase 063)
```

Un clúster compartido por equipos de la misma empresa es un caso; ejecutar código arbitrario de clientes en espacios de nombres es otro, y el segundo exige lo de la tercera línea.

2. El permiso suma, y los tres caminos indirectos

El control de acceso de Kubernetes es puramente aditivo: hay reglas que conceden y no hay reglas que quiten. Cuarta plataforma del programa con la misma propiedad, después de AWS, Azure y Google Cloud, y ya se puede afirmar como regla general del oficio:

> El permiso suma, y lo que resta vive en otro sistema con otro error. Aquí ese otro sistema es la admisión.

Los objetos son cuatro y su combinación cubre todo:

```
rol             permisos DENTRO de un espacio de nombres
rol de clúster  permisos sobre recursos de clúster, o plantilla reutilizable
vinculación     asocia un rol a un sujeto, dentro de un espacio
vinculación de clúster   lo mismo, en el clúster ENTERO
```

Y el error más caro es usar la cuarta cuando bastaba la tercera:

```
$ kubectl get clusterrolebindings -o json | jq -r '.items[]
  | select(.roleRef.name=="cluster-admin")
  | "\(.metadata.name): \(.subjects[]?.kind)/\(.subjects[]?.name)"'
```

Cualquier resultado inesperado ahí es administración total del clúster, incluida la lectura de todos los secretos de todos los espacios.

Y los tres caminos indirectos, que una auditoría que solo mire verbos directos no encuentra:

```
1. crear pods en un espacio  →  leer todos sus secretos          (clase 076)
2. emitir testigos de una cuenta de servicio  →  actuar como ella
3. crear o vincular roles  →  concederse a uno mismo lo que quiera
```

El tercero tiene una protección integrada que conviene conocer porque a veces se desactiva: no se puede conceder un permiso que uno mismo no tiene, salvo que se tenga el verbo de escalada. Comprobar quién lo tiene es la pregunta correcta:

```
$ kubectl get clusterroles -o json | jq -r '.items[]
  | select(.rules[]? | (.verbs[]? == "escalate") or (.verbs[]? == "bind"))
  | .metadata.name'
```

Y las cuentas de servicio son la identidad de las cargas, con dos hechos que hay que fijar:

```
cada pod tiene una, y si no se declara usa la predeterminada del espacio
el testigo se proyecta con caducidad corta y con audiencia declarada
```

El primero produce un exceso de permisos silencioso: la cuenta predeterminada la comparten todos los pods del espacio, así que cualquier permiso que se le conceda lo tienen todos. La higiene es dar una cuenta propia a cada carga y desactivar el montaje automático donde no hace falta:

```
spec:
  serviceAccountName: api
  automountServiceAccountToken: false      # si la carga no habla con la API
```

La mayoría de las aplicaciones no hablan con la API de Kubernetes, así que la mayoría no necesita ningún testigo montado. Quitarlo elimina una credencial de dentro del contenedor, que es lo que un atacante busca primero.

Y la conexión con las partes 02 a 04: la cuenta de servicio del clúster puede federarse con la identidad de la nube, de modo que el pod obtiene credenciales del proveedor sin ninguna clave. Es el mismo contrato de las clases 026, 038, 050 y 069, aplicado por séptima vez:

```
metadata:
  annotations:
    # la anotación concreta depende del proveedor (clase 083)
    identidad-nube/rol: "arn:…:role/api-tienda"
```

Y con la misma condición crítica de siempre: la confianza del lado de la nube tiene que estar acotada al espacio de nombres y a la cuenta concretos. Sin acotar, cualquier pod del clúster puede pedir esa identidad.

3. Una política de red no deniega: cambia el modo del pod

Este es el concepto peor entendido de la clase y produce dos errores opuestos.

```
sin ninguna política                todos los pods se alcanzan entre sí
con una política que SELECCIONA un pod
  ese pod pasa a DENEGAR POR DEFECTO en la dirección que la política declare
  y solo se permite lo que la propia política (o otra) autorice
```

Es decir: la política no añade una prohibición; cambia el modo de los pods que selecciona. Y solo en su dirección: una política de entrada no afecta a la salida.

La consecuencia práctica es que el primer objeto que hay que escribir en un espacio de nombres es el que lo cierra:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata: {name: denegar-todo, namespace: tienda}
spec:
  podSelector: {}            # todos los pods del espacio
  policyTypes: [Ingress, Egress]
```

Sin reglas, eso deja el espacio cerrado en ambas direcciones. Y a partir de ahí se abre lo necesario:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata: {name: api-permitido, namespace: tienda}
spec:
  podSelector: {matchLabels: {app: api}}
  policyTypes: [Ingress, Egress]
  ingress:
    - from:
        - podSelector: {matchLabels: {app: web}}
        - namespaceSelector: {matchLabels: {kubernetes.io/metadata.name: red}}
          podSelector: {matchLabels: {app: entrada}}
      ports: [{protocol: TCP, port: 8080}]
  egress:
    - to: [{podSelector: {matchLabels: {app: bd}}}]
      ports: [{protocol: TCP, port: 5432}]
    - to:
        - namespaceSelector: {matchLabels: {kubernetes.io/metadata.name: kube-system}}
          podSelector: {matchLabels: {k8s-app: kube-dns}}
      ports: [{protocol: UDP, port: 53}, {protocol: TCP, port: 53}]
```

Esa última regla es la que se olvida siempre y produce el incidente más desconcertante: una política de salida sin permitir la resolución de nombres rompe absolutamente todo, incluidos los destinos que la propia política permite, porque el pod no puede resolver sus nombres.

```
síntoma   "permití la base de datos y sigue sin conectar"
causa     no puede resolver bd.tienda.svc
```

Y una precisión sobre los selectores que produce reglas más abiertas de lo previsto:

```
# DOS orígenes: cualquier pod del espacio red, O cualquier pod con app=web
from:
  - namespaceSelector: {matchLabels: {name: red}}
  - podSelector: {matchLabels: {app: web}}

# UN origen: pods con app=entrada DENTRO del espacio red
from:
  - namespaceSelector: {matchLabels: {name: red}}
    podSelector: {matchLabels: {app: entrada}}
```

La diferencia es un guion. La primera forma es mucho más permisiva de lo que la mayoría pretende escribir.

Y el fallo silencioso de esta clase, octava aparición de la familia de la clase 060:

```
si el complemento de red instalado NO implementa políticas,
el objeto se crea, se lista, aparece en el inventario
y NO FILTRA NADA
```

No hay error, no hay aviso, y el panel de cumplimiento dice que el espacio está segmentado. La única forma de saberlo es probarlo:

```
$ kubectl run atacante --rm -it -n desarrollo --image=nicolaka/netshoot -- \
    nc -zv bd.tienda.svc 5432
nc: connect to bd.tienda.svc port 5432 (tcp) failed: Connection timed out    ✓
```

Esa prueba negativa debería estar en el guion de verificación y ejecutarse en cada clúster, porque el complemento de red puede cambiar en una migración.

4. Imponer el endurecimiento en vez de revisarlo

La clase 069 dejó una lista de once puntos de endurecimiento. Revisarla a mano en cada pull request no escala; imponerla en la admisión sí.

El mecanismo integrado son los perfiles de seguridad de pods, con tres niveles:

```
privilegiado  sin restricciones
base          impide lo obviamente peligroso: modo privilegiado, espacios
              de nombres del anfitrión, capacidades peligrosas
restringido   además: sin privilegios, sin capacidades salvo una,
              sin elevación, perfil de llamadas al sistema declarado
```

Y se aplican por espacio de nombres, con tres modos que permiten adoptarlo sin romper nada:

```
apiVersion: v1
kind: Namespace
metadata:
  name: tienda
  labels:
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/enforce-version: latest
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/audit: restricted
```

```
audit    lo registra y lo permite      → para medir el tamaño del problema
warn     avisa a quien lo despliega    → para que los equipos lo vean
enforce  lo rechaza                    → cuando el inventario ya cumple
```

Es exactamente la secuencia que este programa ha aplicado en tres nubes: inventariar, corregir y después imponer. Empezar por el modo de rechazo bloquea despliegues legítimos y termina con la etiqueta quitada «temporalmente».

Y el perfil restringido exige justo lo de la clase 069, lo que cierra el círculo:

```
securityContext:
  runAsNonRoot: true
  runAsUser: 10001
  allowPrivilegeEscalation: false
  capabilities: {drop: [ALL]}
  seccompProfile: {type: RuntimeDefault}
  readOnlyRootFilesystem: true
```

Para lo que el perfil no cubre —reglas propias de la organización— está la validación en la admisión con expresiones, que evita tener que operar un servicio externo con los riesgos de la clase 073:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicy
metadata: {name: solo-huellas}
spec:
  matchConstraints:
    resourceRules:
      - apiGroups: ["apps"], apiVersions: ["v1"], operations: ["CREATE","UPDATE"],
        resources: ["deployments"]
  validations:
    - expression: >-
        object.spec.template.spec.containers.all(c, c.image.contains('@sha256:'))
      message: "las imágenes deben referenciarse por huella (clase 061)"
```

Eso impone por sí solo dos reglas de partes anteriores —despliegue por huella de la clase 061 y verificación de firma de la 067— sin depender de que nadie las recuerde. Y a diferencia de un complemento externo, no puede dejar el clúster bloqueado si un servicio se cae, porque se evalúa dentro del propio servidor de API.

Y el conjunto de comprobaciones que conviene automatizar sobre un clúster:

```
# 1. espacios sin perfil de seguridad
$ kubectl get ns -o json | jq -r '.items[]
  | select(.metadata.labels["pod-security.kubernetes.io/enforce"] == null)
  | .metadata.name'

# 2. espacios sin política de red que los cierre
$ for ns in $(kubectl get ns -o name | cut -d/ -f2); do
    [ "$(kubectl get netpol -n $ns --no-headers 2>/dev/null | wc -l)" = "0" ] \
      && echo "sin políticas: $ns"
  done

# 3. cuentas con administración total
$ kubectl get clusterrolebindings -o json | jq -r '.items[]
  | select(.roleRef.name=="cluster-admin") | .metadata.name'

# 4. pods con testigo montado que no hablan con la API
```

```
$ kubectl get pods -A -o json | jq -r '.items[]
  | select(.spec.automountServiceAccountToken != false)
  | "\(.metadata.namespace)/\(.metadata.name)"' | wc -l
```

5. La tercera fuga, calificada

La clase 072 predijo que Kubernetes renombraría la fuga de identidad sin resolverla. Con lo visto, la calificación es matizada y merece precisión.

```
lo que Kubernetes SÍ aporta
  un modelo de permisos propio, uniforme y auditable
  identidad de carga integrada, federable con la nube
  admisión: el sistema que RESTA, con perfiles listos para usar
  un objeto de segmentación de red declarativo

lo que NO resuelve y devuelve a quien lo opera
  la conectividad total por defecto: hay que cerrarla a mano
  los caminos indirectos de permiso: hay que contarlos igual
  la comprobación de que las políticas se aplican: hay que probarla
  el aislamiento fuerte entre inquilinos: no lo da
```

El segundo bloque es exactamente la misma lista que la parte 05 dejó abierta, con nombres nuevos. Y la novedad propia —el objeto de política de red que puede no hacer nada— añade una fuga en vez de cerrar una.

Y hay una diferencia respecto de las nubes que conviene señalar porque cambia el diseño: en las partes 02 a 04, la red y la identidad eran del proveedor y estaban siempre activas. Aquí, la política de red depende de un complemento que se elige al montar el clúster, con lo que la seguridad de red del clúster es una decisión de instalación, no de configuración. La clase 083 vuelve sobre esto porque cada plataforma gestionada trae complementos distintos.

Y la lista de comprobación de la clase, que alimenta el proyecto de la 084:

- un espacio de nombres por equipo y entorno, con cuota y rango de límites
- política que cierra cada espacio en ambas direcciones, y aperturas explícitas
- resolución de nombres permitida en toda política de salida
- prueba negativa que demuestra que las políticas se aplican de verdad
- ninguna vinculación de clúster con administración total fuera de la plataforma
- una cuenta de servicio por carga; testigo desmontado donde no se use
- federación con la nube acotada al espacio y a la cuenta concretos
- perfil de seguridad de pods en modo de rechazo, tras haber medido con auditoría
- políticas propias en la admisión para huella de imagen y firma
- auditoría de permisos que cuente los tres caminos indirectos

Diez puntos, de los cuales tres son pruebas y no configuraciones — que es la proporción que este programa ha ido defendiendo desde la clase 046.

Ejemplo trabajado

CloudShop audita su clúster antes de admitir a un segundo equipo. Los cinco hallazgos comparten una propiedad: todo estaba configurado y casi nada estaba comprobado.

Hallazgo 1 — las políticas de red no filtraban nada.

El clúster tenía 14 políticas de red escritas hacía ocho meses. La prueba:

```
$ kubectl run atacante --rm -it -n desarrollo --image=nicolaka/netshoot -- \
    nc -zv bd.produccion.svc 5432
Connection to bd.produccion.svc 5432 port [tcp/*] succeeded!
```

El complemento de red instalado no implementaba políticas. Los objetos existían, se listaban y el panel de cumplimiento los contaba como control activo.

```
complemento de red                    sin soporte de     con soporte
                                      políticas
políticas que filtraban de verdad        0 de 14          14 de 14
prueba negativa                         ninguna       en el guion, por espacio
tiempo que llevaban sin efecto          8 meses              —
```

Octava aparición de la familia de fallos de la clase 060, y la más incómoda de la parte: el control existía, estaba documentado y no hacía nada.

Hallazgo 2 — al activar las políticas, todo dejó de funcionar.

```
api   Error: getaddrinfo EAI_AGAIN bd.tienda.svc
```

Las políticas de salida no permitían la resolución de nombres, así que ningún pod podía resolver nada — ni siquiera los destinos que la política permitía explícitamente.

```
regla de resolución de nombres          ausente        en las 14 políticas
fallo tras activar                      total          ninguno
plantilla de política del equipo        no había       con la regla incluida
```

La corrección fue añadir la misma regla a las catorce, y la medida de fondo fue una plantilla que ya la trae.

Hallazgo 3 — cuarenta personas con administración total.

```
$ kubectl get clusterrolebindings -o json | jq -r '.items[]
  | select(.roleRef.name=="cluster-admin") | .metadata.name'
cluster-admin
desarrolladores-admin        ← 38 personas
soporte-admin                ←  4 personas
```

La vinculación de desarrolladores se había creado durante la puesta en marcha «temporalmente». Concedía lectura de todos los secretos de todos los espacios, entre otras cosas.

```
personas con administración total           42              2
roles por equipo                          ninguno       editor en su espacio
acceso a producción                     permanente    temporal, con solicitud
                                                        (clase 038)
caminos indirectos auditados                no        los tres, trimestral
```

Y al contar los caminos indirectos, el número efectivo antes de la corrección era mayor que 42, porque quien podía crear pods en un espacio leía sus secretos.

Hallazgo 4 — el testigo de la API montado en todo.

```
$ kubectl get pods -A -o json | jq -r '.items[]
  | select(.spec.automountServiceAccountToken != false) | .metadata.name' | wc -l
186
$ # de esos, los que realmente hablan con la API:
4
```

Ciento ochenta y dos contenedores con una credencial del clúster montada dentro sin usarla — exactamente lo que un atacante busca tras conseguir ejecución de código (clase 069).

```
pods con testigo montado                    186               4
cuentas de servicio propias por carga       3 de 21        21 de 21
uso de la cuenta predeterminada             18 cargas         0
permisos de la cuenta predeterminada    heredados de una
                                        vinculación amplia  ninguno
```

Hallazgo 5 — el endurecimiento revisado a mano no se cumplía.

La lista de once puntos de la clase 069 se revisaba en los pull requests. Medida con el perfil en modo auditoría durante una semana:

```
pods que incumplían el perfil restringido      31 de 74
incumplimientos más frecuentes
  sin runAsNonRoot                              22
  capacidades no retiradas                      19
  raíz escribible                               17
  sin perfil de llamadas al sistema             28
```

```
mecanismo                           revisión manual      admisión
modo                                      —          auditoría → aviso → rechazo
tiempo de adopción                        —          6 semanas
pods que incumplen                     31 de 74           0 de 74
excepciones                               —          2, con motivo y fecha
```

Las dos excepciones son un agente de recolección que necesita acceso a rutas del nodo, con su justificación escrita y revisión semestral.

Y la comprobación que se añadió al guion de verificación:

```
$ ./verificar-aislamiento.sh
✓ desarrollo → producción:5432        tiempo agotado
✓ tienda → kube-system:6443           tiempo agotado
✓ tienda/web → tienda/api:8080        conecta
✓ tienda/api → tienda/bd:5432         conecta
✓ tienda/api → internet:443           tiempo agotado
✓ resolución de nombres desde tienda   correcta
```

```
6/6 correctas
```

Resumen:

```
políticas de red efectivas                    0 de 14        14 de 14
personas con administración total               42              2
pods con testigo de la API montado             186              4
pods que incumplen el endurecimiento         31 de 74         0 de 74
cuentas de servicio propias por carga         3 de 21        21 de 21
pruebas negativas de aislamiento                 0              6
```

La lección que esta clase traslada al resto de la parte 06: los cinco hallazgos existían con la configuración escrita, revisada y documentada. Lo que faltaba era la comprobación, y en el caso más grave —catorce políticas de red sin efecto durante ocho meses— la comprobación es una sola orden desde un pod cualquiera. La tercera fuga de la clase 072 se confirma con un matiz: Kubernetes no solo devuelve las obligaciones de identidad y de red, sino que añade una propia — un objeto de seguridad que puede existir sin que nada lo implemente.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/080-namespaces-rbac-networkpolicy-y-admission/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 tenant-kubernetes 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 tenant-kubernetes 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
Las políticas de red existen y cualquier pod alcanza cualquier otro	El complemento de red instalado no implementa políticas, y el objeto se crea igualmente	Comprueba con una prueba negativa desde un pod real, y repítela en cada clúster y tras cada migración de complemento.
Al activar políticas de salida todo deja de funcionar	No se permitió la resolución de nombres, así que no se resuelven ni los destinos permitidos	Incluye siempre la regla hacia el servicio de nombres, y ten una plantilla de política que ya la traiga.
Una regla de red es mucho más abierta de lo previsto	Dos selectores como elementos separados de la lista son dos orígenes distintos, no una intersección	Combina espacio y pod en el mismo elemento cuando quieras intersección; la diferencia es un guion.
Una auditoría de permisos da un número tranquilizador y falso	Solo se contaron los verbos directos y no los tres caminos indirectos	Cuenta crear pods, emitir testigos y vincular roles; el permiso efectivo es la unión.
Casi todos los contenedores tienen una credencial del clúster dentro	El testigo de la cuenta de servicio se monta por defecto	Cuenta propia por carga y montaje desactivado donde la aplicación no hable con la API.
El endurecimiento se revisa en cada cambio y aun así no se cumple	La revisión manual no escala y las excepciones se cuelan	Impón el perfil en la admisión, adoptándolo por fases: auditoría, aviso y rechazo.

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 acota un espacio de nombres y qué no, y por qué no basta para inquilinos no confiables?
2. ¿Qué ocurre exactamente cuando una política de red selecciona un pod, y en qué direcciones?
3. ¿Por qué una política de salida rompe todo si no permite la resolución de nombres?
4. Enumera los tres caminos indirectos de permiso y qué concede cada uno.
5. ¿Qué secuencia de adopción evita que un perfil de seguridad de pods bloquee despliegues legítimos?

Referencias

- Kubernetes (2025). Using RBAC authorization — roles, vinculaciones, escalada y protecciones. <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>
- Kubernetes (2025). Network policies — semántica de selección, direcciones y requisitos del complemento. <https://kubernetes.io/docs/concepts/services-networking/network-policies/>
- Kubernetes (2025). Pod Security Admission — perfiles, modos y adopción por fases. <https://kubernetes.io/docs/concepts/security/pod-security-admission/>
- Kubernetes (2025). Validating Admission Policy — políticas con expresiones dentro del servidor de API. <https://kubernetes.io/docs/reference/access-authn-authz/validating-admission-policy/>
- Kubernetes (2025). Multi-tenancy — límites del espacio de nombres como frontera y alternativas. <https://kubernetes.io/docs/concepts/security/multi-tenancy/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 06 en PDF · Manual integral

Evaluación — 080 Namespaces, RBAC, NetworkPolicy y admission

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 tenant-kubernetes 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.

081 — Helm, Kustomize y gestión de paquetes

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: configuration · Estado: EXECUTABLECORE

Propósito

Gestionar los manifiestos de una aplicación en varios entornos sin duplicarlos ni convertirlos en un generador de texto ilegible. Hay dos filosofías —superponer capas sobre una base, o rellenar plantillas con valores— y la elección correcta depende de si el destinatario es tu propio equipo o alguien de fuera. La clase compara ambas con honestidad, señala las trampas concretas de cada una, y cierra con la afirmación que enlaza con la parte 08: generar manifiestos no es desplegar, y confundirlo es la ley de la clase 073 otra vez.

Resultados de aprendizaje

Al finalizar podrás:

1. Elegir entre superposición y plantillas según quién consume el resultado.
2. Fijar versiones e imágenes por huella en ambos enfoques, con el mismo criterio de las clases 061 y 062.
3. Anticipar los bloqueos de estado que deja una actualización fallida y cómo salir de ellos.
4. Forzar un despliegue al cambiar la configuración, con el mecanismo que cada herramienta ofrece.
5. Verificar el resultado antes de aplicarlo, y la convergencia después.

Conceptos centrales

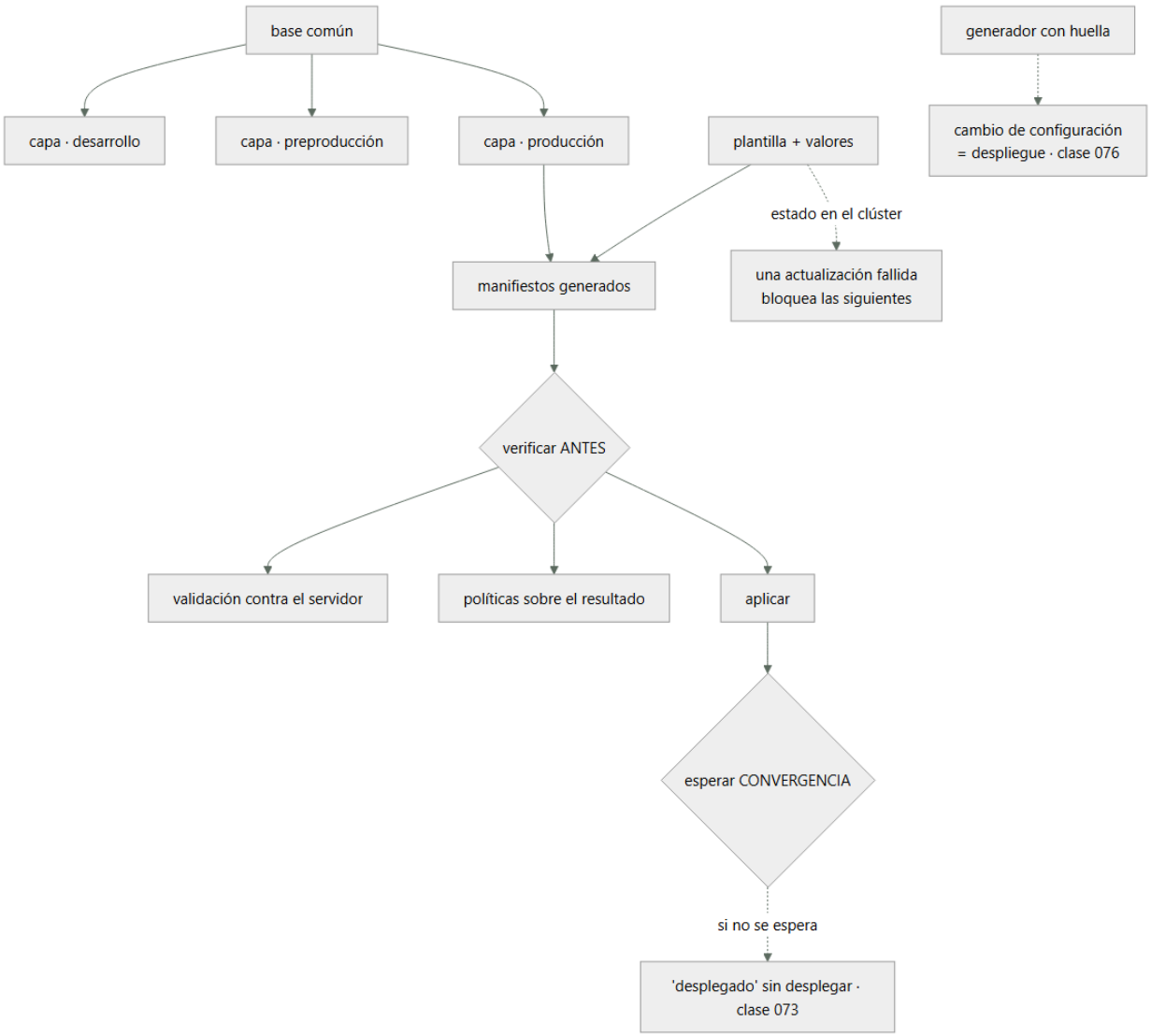
Concepto	Comprensión verificable
superposición	Capas que modifican una base común mediante parches. En cada paso el resultado es un manifiesto válido, así que se puede revisar.
plantilla con valores	Texto con marcadores que se rellenan. Es más flexible y puede generar algo que no es válido, porque hasta el final no hay estructura.
estado de la publicación	Registro que la herramienta de plantillas guarda en el clúster con lo que instaló. Una actualización fallida lo deja en un estado que bloquea las siguientes.
gancho de publicación	Trabajo que se ejecuta antes o después de aplicar. Es donde va una migración de esquema, con la advertencia de la clase 074: una vez, no una por réplica.
generador con huella	Mecanismo que crea el objeto de configuración con un sufijo derivado de su contenido. Resuelve por diseño el problema de la clase 076.
campo inmutable	Selector de un despliegue y otros campos que no se pueden modificar. Una herramienta que añade etiquetas comunes a los selectores rompe la actualización.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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. Dos filosofías, y el criterio para elegir

El problema es siempre el mismo: la misma aplicación en cuatro entornos, con diferencias pequeñas —réplicas, límites, nombres de host, huellas de imagen— y sin duplicar cuatro veces los manifiestos.

```
SUPERPOSICIÓN     una base común y capas que la parchean
  el resultado de cada paso es un manifiesto VÁLIDO
  se puede revisar, comparar y validar en cualquier punto
  no hay lógica: no hay condicionales ni bucles

PLANTILLAS        texto con marcadores y un fichero de valores
  permite lógica: condicionales, bucles, funciones
  hasta el final NO hay estructura: se puede generar algo inválido
  incluye empaquetado, versionado y ciclo de vida de la publicación
```

Y el criterio de elección, que evita la discusión religiosa:

```
¿el destinatario es TU equipo, con tus manifiestos?
  → superposición. Menos maquinaria, resultado revisable,
    y viene integrada en la herramienta de línea de órdenes

¿vas a DISTRIBUIR software a terceros que no conoces?
  → plantillas. Necesitas parámetros, valores por defecto,
    versiones y una forma de instalar y desinstalar

¿consumes software de terceros?
  → sus paquetes vienen en plantillas: no hay elección
```

La tercera fila es la razón por la que la mayoría de los equipos acaban usando ambas: sus propias aplicaciones con superposición y los componentes de terceros con paquetes. Y hay una combinación que funciona bien y conviene conocer: generar el paquete de terceros a manifiestos y tratarlos después como una base más, con lo que todo el inventario acaba en el mismo formato revisable.

```
$ helm template ingress-nginx ingress-nginx/ingress-nginx \
    --version 4.11.3 --values valores.yaml > base/ingress.yaml
```

Eso tiene una ventaja concreta: el cambio entre versiones del paquete se ve como un diferencial de manifiestos en el pull request, en vez de como un número de versión que nadie sabe qué implica.

Y dos disciplinas que se aplican igual en las dos filosofías, y que vienen de las clases 059, 061 y 062:

```
fijar la versión del paquete de terceros    nunca "la última"
referenciar las imágenes por huella          nunca por etiqueta
```

La segunda tiene un mecanismo propio en la superposición que conviene usar:

```
images:
  - name: registro/api
    digest: sha256:9f2c4a1b3d5e…
```

Con eso, la canalización escribe la huella en un solo sitio y todos los manifiestos que referencien esa imagen la reciben. Es la aplicación directa de la regla de la clase 061, y evita el incidente de la 074 —la vuelta atrás que no volvía a ninguna parte— por construcción.

2. El estado de la publicación, y cómo se atasca

La herramienta de plantillas guarda en el clúster un registro de lo que instaló, y eso da capacidades reales:

```
$ helm history api -n tienda
REVISION  STATUS      CHART        APP VERSION  DESCRIPTION
1         superseded  api-1.4.0    v7           Install complete
2         deployed    api-1.5.0    v8           Upgrade complete
$ helm rollback api 1 -n tienda
```

Y trae un problema operativo que aparece el primer mes: una actualización que falla deja el estado a medias y bloquea las siguientes.

```
$ helm upgrade api ./api -n tienda
Error: UPGRADE FAILED: another operation (install/upgrade/rollback) is in progress

$ helm history api -n tienda
3    pending-upgrade    api-1.6.0    v9    Preparing upgrade
```

El estado pending-upgrade se queda ahí si el proceso murió a mitad —un agente de canalización que se reinició, un tiempo de espera agotado— y ninguna actualización posterior funciona hasta resolverlo:

```
$ helm rollback api 2 -n tienda        # vuelve al último estado bueno conocido
```

Las dos opciones que evitan llegar ahí:

```
$ helm upgrade --install api ./api -n tienda \
    --atomic --timeout 5m --wait

--wait     espera a que los objetos estén LISTOS, no solo creados
           → es la corrección de la ley de la clase 073
--atomic   si falla o expira, revierte automáticamente
           → nunca queda un estado a medias
```

Y una consecuencia de --atomic que hay que conocer: la reversión automática también deshace lo que sí funcionó, lo que puede dejar el sistema en un estado que nadie ha probado. Para publicaciones pequeñas es lo correcto; para una que toca muchos objetos, conviene tiempo de espera generoso y revisar antes de automatizar la reversión.

Los ganchos son el mecanismo para el trabajo que rodea a la publicación, y ahí va la migración de esquema que la clase 074 sacó de los contenedores de inicialización:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: migrar
  annotations:
    "helm.sh/hook": pre-upgrade,pre-install
    "helm.sh/hook-weight": "-5"
    "helm.sh/hook-delete-policy": before-hook-creation,hook-succeeded
```

```
spec:
  backoffLimit: 2
  activeDeadlineSeconds: 900
  ttlSecondsAfterFinished: 3600
```

La política de borrado importa por lo que la clase 074 midió: sin ella, cada publicación deja un objeto terminado que nadie retira.

Y una limitación conocida que sorprende y hay que planificar: las definiciones de tipos que un paquete instala no se actualizan al actualizar el paquete. Se instalan la primera vez y después quedan congeladas. Actualizarlas es un paso manual:

```
$ kubectl apply --server-side -f https://…/crds-v1.6.0.yaml
$ helm upgrade api ./api -n tienda
```

Ese orden —primero los tipos, después el paquete— es el correcto, y saltárselo produce un paquete nuevo escribiendo campos que la definición antigua rechaza.

3. Las trampas de la superposición

La superposición tiene menos maquinaria y dos trampas propias que rompen despliegues.

Las etiquetas comunes tocan los selectores. Añadir una etiqueta a todos los objetos parece inocuo:

```
commonLabels:
  entorno: produccion
```

Pero el selector de un despliegue es inmutable, y esa etiqueta se añade también al selector:

```
The Deployment "api" is invalid: spec.selector: Invalid value:
  field is immutable
```

El despliegue existente no se puede actualizar y hay que borrarlo y recrearlo — con corte. La corrección es usar el mecanismo que añade etiquetas sin tocar selectores:

```
labels:
  - pairs: {entorno: produccion}
    includeSelectors: false          # ← lo importante
```

Misma advertencia con los prefijos de nombre: cambiarlos en un entorno ya desplegado crea objetos nuevos y deja los viejos huérfanos, sin que nadie los retire.

El generador con huella, que resuelve el problema de la clase 076.

```
configMapGenerator:
  - name: api-config
    files: [config.yaml]
secretGenerator:
  - name: api-tls
    files: [tls.crt, tls.key]
```

Eso genera api-config-7f4c9b2d con un sufijo derivado del contenido, y reescribe todas las referencias en los manifiestos. La consecuencia es exactamente lo que la clase 076 pedía a mano:

```
cambia el contenido → cambia el nombre → cambia la plantilla del pod
→ hay despliegue progresivo, con historial y vuelta atrás
```

Y viene con un efecto secundario que hay que gestionar: los objetos antiguos no se borran solos. Se acumulan versiones de configuración que nadie retira, con el mismo efecto sobre el almacén de estado que los trabajos terminados de la clase 074. Se resuelve con una limpieza periódica de los que ya nadie referencia.

Y las dos formas de parchear, con criterio para elegir:

```
# fusión estratégica: legible, para cambios de campos
patches:
  - target: {kind: Deployment, name: api}
    patch: |-
      apiVersion: apps/v1
      kind: Deployment
      metadata: {name: api}
      spec:
        replicas: 6
        template:
          spec:
            containers:
```

```
              - name: api
                resources: {requests: {cpu: 500m, memory: 1Gi}}

# por ruta: preciso, para listas y para borrar
  - target: {kind: Deployment, name: api}
    patch: |-
      - op: remove
        path: /spec/template/spec/containers/0/livenessProbe
```

La segunda es la única forma de quitar un campo, que la fusión no puede hacer. Y conviene conocerla porque el caso aparece siempre: una base con un valor que un entorno concreto no debe tener.

4. Generar no es desplegar

Las dos herramientas producen manifiestos. Lo que ocurre después es lo que decide si algo se ha desplegado, y aquí vuelve la ley de la clase 073 con un mecanismo nuevo.

El flujo completo, con las comprobaciones en su sitio:

```
# 1. generar y VER lo que va a aplicarse
$ kubectl kustomize entornos/produccion > salida.yaml
$ git diff --no-index anterior.yaml salida.yaml

# 2. validar contra el servidor: esquema, tipos, admisión
$ kubectl apply --server-side --dry-run=server -f salida.yaml

# 3. validar políticas propias sobre el resultado
$ conftest test salida.yaml

# 4. aplicar
$ kubectl apply --server-side -f salida.yaml

# 5. ESPERAR la convergencia
$ kubectl rollout status deploy/api -n tienda --timeout=300s
$ kubectl wait --for=condition=Available deploy/api -n tienda --timeout=300s
```

Los pasos 1 y 2 son los que este programa ha pedido en tres formas distintas —el what-if de la clase 047, el plan guardado de la 059 y ahora esto— y el 5 es la corrección de la ley de la clase 073.

Dos detalles del paso 4 que evitan problemas:

La aplicación del lado del servidor registra qué campo gestiona cada quien, con lo que dos herramientas que tocan el mismo objeto no se pisan en silencio:

```
Apply failed with 1 conflict: conflict with "otro-controlador":
  .spec.replicas
```

Eso es una buena noticia: sin ella, el objeto de escalado automático y el manifiesto se sobrescriben mutuamente y el número de réplicas oscila sin explicación. Con ella, el conflicto se hace visible y se resuelve declarando quién manda:

```
si el escalado automático gestiona las réplicas,
el manifiesto NO debe declararlas
```

El diferencial antes de aplicar responde a la pregunta que importa:

```
$ kubectl diff -f salida.yaml
```

Y su valor es el mismo que el del what-if de la clase 047, con la misma advertencia: si produce mucho ruido, el equipo deja de leerlo. El ruido aquí viene de campos que el servidor rellena y el manifiesto no declara, y se corrige declarándolos.

Y el cierre que enlaza con la parte 08: todo lo anterior sigue siendo alguien ejecutando órdenes. Lo que hace que el clúster converja de verdad hacia lo que dice el repositorio, y que una modificación manual se deshaga sola, es un controlador que reconcilie continuamente — el mismo bucle de la clase 073 aplicado al despliegue. Eso es la entrega continua declarativa, y es materia de la parte 08. Lo que hay que llevarse de aquí es la frase:

> Generar manifiestos, validarlos y aplicarlos no garantiza que el clúster se parezca al repositorio dentro de una semana. Solo un bucle que reconcilie lo garantiza.

5. Lo que ninguna de las dos debe contener

Una advertencia que cierra la clase y recoge tres partes anteriores: ni los paquetes ni las superposiciones deben llevar secretos.

Los dos ofrecen mecanismos para incluirlos, y los dos acaban en el repositorio:

```
valores con contraseñas          en el repositorio, y en el historial (ley de 072)
generador de secretos con ficheros  igual, salvo que los ficheros vengan de fuera
```

Las opciones correctas son las de la clase 076, por orden:

```
1. el valor nunca entra en el clúster: se monta desde el gestor externo
2. un controlador lo sincroniza desde el gestor, y el repositorio solo
   guarda una REFERENCIA
3. cifrado en el repositorio con una clave que solo el clúster puede usar
   → funciona, y hay que gestionar el ciclo de vida de esa clave
```

La tercera es popular y merece una precisión: el fichero cifrado en el repositorio es seguro mientras la clave lo sea, y esa clave es ahora un activo más que rotar, custodiar y auditar. No es peor que las alternativas; es una decisión con su propio coste.

Y la comprobación que conviene tener en la canalización, que es la de la clase 067 aplicada a manifiestos:

```
$ kubectl kustomize entornos/produccion \
  | grep -Ei '(password|token|secret|apikey)\s*[:=]\s*["\x27]?[A-Za-z0-9+/]{16,}' \
  && { echo "posible secreto en los manifiestos"; exit 1; }
```

Y la lista de comprobación de la clase, que alimenta el proyecto de la 084:

- una base y una capa por entorno; sin manifiestos duplicados
- versiones de paquetes de terceros fijadas, nunca la última
- imágenes por huella, escritas en un solo sitio por la canalización
- etiquetas comunes que NO tocan los selectores
- configuración por generador con huella, para que cambiarla sea un despliegue
- migraciones como gancho o trabajo previo, nunca por réplica
- ningún secreto en el repositorio, ni en valores ni en ficheros
- diferencial revisado y validación contra el servidor antes de aplicar
- espera de convergencia después de aplicar, con tiempo límite
- limpieza periódica de objetos de configuración que ya nadie referencia

Diez puntos, de los cuales cuatro son reglas que vienen de las clases 061, 072, 074 y 076 — lo que confirma que esta clase es sobre todo el sitio donde se aplican decisiones tomadas antes.

Ejemplo trabajado

CloudShop tiene cuatro entornos con manifiestos copiados y editados a mano. La unificación produce cuatro incidentes, y el más caro no es de la herramienta sino de una etiqueta.

Punto de partida:

```
4 carpetas con manifiestos completos, copiados y divergentes
diferencias reales entre entornos:      12
diferencias accidentales encontradas:   31
paquetes de terceros instalados a mano, sin versión anotada:  6
```

Las treinta y una diferencias accidentales incluían dos que explicaban incidentes anteriores: un plazo de gracia distinto en producción y una comprobación de vivacidad con umbral 1 en preproducción.

Incidente 1 — la etiqueta común que rompió el despliegue.

```
The Deployment "api" is invalid: spec.selector: Invalid value: … field is immutable
```

La unificación añadió entorno: a todos los objetos, y eso incluyó los selectores de los despliegues existentes.

```
mecanismo de etiquetas             etiquetas comunes   etiquetas sin selectores
despliegues que hubo que recrear         7                  0
corte por recreación                  40 s cada uno         —
detectado en                          producción         validación contra
                                                          el servidor en el PR
```

Se detectó en producción porque el paso de validación contra el servidor no estaba en la canalización. Añadirlo fue la corrección de fondo.

Incidente 2 — las publicaciones bloqueadas por un estado a medias.

```
$ helm upgrade ingress-nginx …
Error: UPGRADE FAILED: another operation (install/upgrade/rollback) is in progress
$ helm history ingress-nginx -n red | tail -1
7   pending-upgrade   ingress-nginx-4.11.1   Preparing upgrade
```

Un agente de canalización se había reiniciado a mitad de una actualización tres semanas antes. Desde entonces, ninguna publicación de ese componente funcionaba, y nadie lo había intentado.

```
opciones de la actualización            ninguna        --install --atomic
                                                        --wait --timeout 5m
estados bloqueados                        1                0
comprobación de estados pendientes      ninguna     en el panel, semanal
tiempo que llevaba bloqueado           3 semanas            —
```

Incidente 3 — los tipos que no se actualizaron.

Al subir de versión el componente de certificados, los objetos nuevos se rechazaban:

```
error validating data: ValidationError(Certificate.spec):
  unknown field "otherNames"
```

El paquete se había actualizado y sus definiciones de tipos no, porque la herramienta no las toca en una actualización.

```
orden de actualización              solo el paquete   tipos primero, paquete después
documentado en el procedimiento          no                sí
paquetes con tipos propios en el clúster   4        los 4, con nota en su ficha
```

Incidente 4 — el número de réplicas que oscilaba.

```
$ kubectl get deploy api -o jsonpath='{.spec.replicas}'
3
# dos minutos después
9
# y otra vez
3
```

El manifiesto declaraba tres réplicas y el objeto de escalado automático las subía a nueve. Cada aplicación de manifiestos las devolvía a tres.

```
$ kubectl apply --server-side --field-manager=kustomize -f salida.yaml
Apply failed with 1 conflict: conflict with "horizontal-pod-autoscaler": .spec.replicas
```

La aplicación del lado del servidor lo hizo visible; antes, la sobrescritura era silenciosa.

```
réplicas en el manifiesto               declaradas      no declaradas
modo de aplicación                    cliente           lado del servidor
oscilaciones al día                      ~40                0
capacidad efectiva en los picos       impredecible      la del escalado
```

Y el efecto de unificar, medido:

```
manifiestos duplicados                      4 copias    1 base + 4 capas
diferencias accidentales entre entornos        31             0
diferencias declaradas y justificadas          12            12
paquetes de terceros con versión fijada      0 de 6        6 de 6
cambios de configuración que despliegan      0 de 9        9 de 9
validación contra el servidor en el PR         no            sí
espera de convergencia tras aplicar            no            sí
tiempo de un despliegue completo            9 min 20 s     2 min 40 s
```

La fila de las diferencias accidentales es la más valiosa: treinta y una divergencias que nadie había decidido, acumuladas por copiar y editar durante dos años, y dos de ellas explicaban incidentes que se habían archivado sin causa.

La lección que esta clase traslada al resto de la parte 06: la herramienta importa menos de lo que parece; lo que importa es que el resultado sea revisable antes de aplicarse y verificable después. Los cuatro incidentes se detectan con dos pasos que no dependen de la herramienta elegida —validación contra el servidor y espera de convergencia— y ninguno de los dos estaba en la canalización. Y la unificación reveló que la mayor parte de las diferencias entre entornos no eran decisiones: eran erosión.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/081-helm-kustomize-y-gestion-de-paquetes/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 paquete-kubernetes 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 paquete-kubernetes para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

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

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
Un despliegue existente no se puede actualizar por un campo inmutable	Una etiqueta común se añadió también al selector, que no se puede modificar	Usa el mecanismo que añade etiquetas sin tocar selectores, y valida contra el servidor en el pull request.
Ninguna publicación de un componente funciona desde hace semanas	Una actualización murió a mitad y dejó el estado en pendiente	Revierte a la última revisión buena y usa siempre espera atómica con tiempo límite; vigila los estados pendientes.
Tras actualizar un paquete, los objetos nuevos se rechazan por campos desconocidos	Las definiciones de tipos no se actualizan al actualizar el paquete	Aplica los tipos primero y el paquete después, y documenta ese orden para cada componente con tipos propios.
El número de réplicas oscila sin explicación	El manifiesto y el escalado automático se sobrescriben mutuamente	Aplica del lado del servidor para que el conflicto sea visible, y no declares en el manifiesto lo que gestiona otro controlador.
Cambiar la configuración no despliega nada	El objeto de configuración tiene nombre fijo, así que la plantilla del pod no cambia	Usa un generador con sufijo derivado del contenido, y limpia periódicamente los objetos que ya nadie referencia.
Aparecen credenciales en el repositorio de manifiestos	Los valores o los ficheros del generador las contienen	Referencia el gestor externo en vez de incluir el valor, y añade una comprobación de patrones a la canalización.

■ 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 superposición y plantillas, y por qué la mayoría de los equipos acaba usando ambas?
2. ¿Qué deja bloqueado una actualización que muere a mitad, y qué dos opciones lo evitan?
3. ¿Por qué añadir una etiqueta común puede impedir actualizar un despliegue existente?
4. ¿Cómo resuelve un generador con huella el problema que la clase 076 planteó, y qué efecto secundario tiene?
5. ¿Por qué generar, validar y aplicar no garantiza que el clúster se parezca al repositorio dentro de una semana?

Referencias

• Kubernetes (2025). Declarative management with Kustomize — bases, capas, parches y generadores. <https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/>

• Helm (2025). Charts and release lifecycle — plantillas, valores, historial y estados. <https://helm.sh/docs/topics/charts/>

• Helm (2025). Custom Resource Definitions — por qué no se actualizan con el paquete. <https://helm.sh/docs/chartbestpractices/customresourcedefinitions/>

• Kubernetes (2025). Server-Side Apply — gestores de campos y detección de conflictos. <https://kubernetes.io/docs/reference/using-api/server-side-apply/>

• Kustomize (2025). Labels and selectors — etiquetas comunes y campos inmutables. <https://kubectl.docs.kubernetes.io/references/kustomize/kustomization/labels/>

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

Descargar: Parte 06 en PDF · Manual integral

Anterior	Índice	Siguiente
← 080 · Namespaces, RBAC, NetworkPolicy y admission	Parte 06 · Programa	082 · Logs, métricas, eventos y depuración →

Evaluación — 081 Helm, Kustomize y gestión de paquetes

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 paquete-kubernetes 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.

082 — Logs, métricas, eventos y depuración

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: observability · Estado: EXECUTABLECORE

Propósito

Ver lo que ocurre dentro de un clúster, con dos hechos que hay que asumir antes de diseñar nada: los sucesos caducan en una hora, así que la explicación de por qué algo falló anoche ya no existe; y la fuente que alimenta el escalado no es un sistema de vigilancia, no guarda historia y confundirla con uno deja al equipo sin datos justo cuando los necesita. La clase cierra además la traducción de la ley de la clase 073 a una métrica concreta que casi ningún panel tiene.

Resultados de aprendizaje

Al finalizar podrás:

1. Distinguir los cuatro tipos de señal del clúster y qué pregunta responde cada uno.
2. Conservar los sucesos y los registros más allá de su vida útil por defecto.
3. Separar las métricas de estado de las de recursos, y saber cuál detecta cada problema.
4. Alertar sobre la distancia entre lo declarado y lo que funciona.
5. Recorrer un diagnóstico completo con las órdenes en el orden que acota.

Conceptos centrales

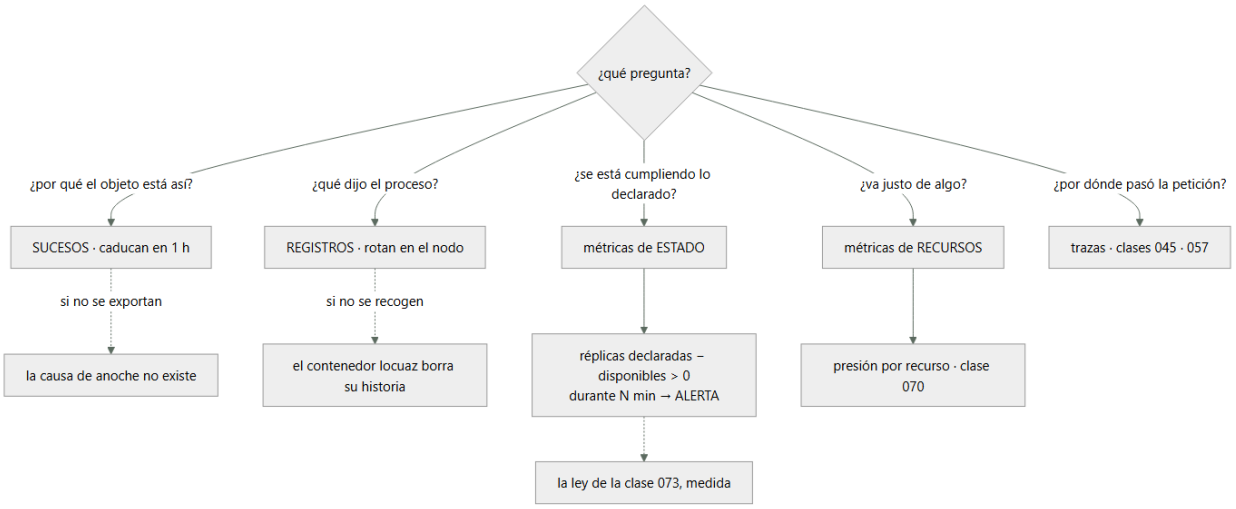
Concepto	Comprensión verificable
suceso	Explicación que un componente deja sobre un objeto. Caduca en una hora por defecto y no se guarda en ningún sitio: es la primera fuente que desaparece.
métrica de estado	Lo que el clúster declara: réplicas deseadas, fase de un pod, condiciones. Responde a si lo pedido está ocurriendo.
métrica de recursos	Lo que las cargas consumen: CPU, memoria, disco. Responde a si algo va justo, y no dice nada sobre si lo declarado se cumplió.
fuente para el escalado	Componente que publica el uso instantáneo para el escalado automático y para consultas puntuales. No guarda historia: no es un sistema de vigilancia.
rotación de registros del nodo	El kubelet rota los registros de cada contenedor por tamaño. Un contenedor locuaz pierde su propio historial en minutos si nadie lo recoge.
distancia entre declarado y disponible	Diferencia entre las réplicas que un objeto pide y las que sirven. Es la ley de la clase 073 expresada como número, y es la alerta que casi nunca existe.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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 señales, cuatro preguntas, y dos que caducan

En un clúster hay cuatro fuentes y cada una responde algo distinto. Usar la equivocada es la causa habitual de un diagnóstico largo:

```
sucesos      ¿por qué este objeto está en este estado?
             lo escriben el planificador, el kubelet y los controladores
registros    ¿qué dijo el proceso?
métricas de estado    ¿se está cumpliendo lo que se declaró?
métricas de recursos  ¿algo va justo de CPU, memoria o disco?
trazas       ¿por dónde pasó una petición concreta? (clases 045, 057)
```

Y las dos primeras tienen una propiedad que hay que asumir antes de diseñar nada.

Los sucesos caducan en una hora.

```
$ kubectl get events -n tienda --sort-by=.lastTimestamp | tail -5
# solo la última hora; lo anterior ya no existe
```

Eso significa que la explicación de por qué un pod se reinició anoche, por qué el planificador no lo colocó o por qué un volumen no montó ha desaparecido cuando alguien mira por la mañana. Es la fuente más útil para el diagnóstico de la clase 073 y la que primero se pierde.

La corrección es exportarlos, y hay dos formas:

```
un recolector que los lea de la API y los envíe al sistema de registros
una alerta que los convierta en métrica cuando importan
```

Y una advertencia sobre el volumen: los sucesos son muchos y repetitivos —el mismo se agrega con un contador— así que exportarlos todos es caro. La práctica sensata es exportar los de tipo aviso y los de razones concretas:

```
FailedScheduling · FailedMount · Unhealthy · BackOff · Killing · Preempted
OOMKilling · FailedCreate · NodeNotReady
```

Los registros rotan en el nodo. El kubelet los guarda en ficheros con un tamaño máximo y un número de ficheros:

```
por defecto, del orden de 10 MiB por fichero y 5 ficheros
→ un contenedor que escribe 50 MiB por minuto conserva su último minuto
```

Y hay una consecuencia peor: kubectl logs lee de ahí, así que la información se pierde antes de que nadie la busque. Sin un recolector que los envíe fuera, el clúster no tiene historial de registros.

Y la opción que salva más diagnósticos y menos gente conoce:

```
$ kubectl logs tienda-7d4b9-x2k4p --previous
```

Eso muestra los registros del contenedor anterior, el que murió. Sin --previous, en un contenedor que reinicia en bucle se leen los registros del que acaba de arrancar, que no dicen nada: la causa está en el que falló.

Y para cuando el pod ya no existe, no hay nada que leer: por eso el recolector no es opcional en cuanto el clúster deja de ser de pruebas.

2. Estado y recursos: dos familias que no se sustituyen

La confusión más extendida es tratar la fuente que alimenta el escalado como si fuera un sistema de vigilancia:

```
$ kubectl top pods -n tienda
NAME              CPU(cores)   MEMORY(bytes)
api-7d4b9-x2k4p   340m         1120Mi
```

Eso es el uso instantáneo y no se guarda. No hay ayer, no hay percentiles, no hay comparación. Sirve para que el escalado automático decida y para una consulta puntual, y no sirve para investigar nada que haya pasado.

El sistema de vigilancia real recoge dos familias distintas, y es importante no confundirlas:

```
métricas de RECURSOS      cuánto consume cada contenedor y cada nodo
  origen: el propio kubelet y agentes del nodo
  responden: ¿va justo? ¿hay presión? (clase 070)

métricas de ESTADO        qué declara el clúster sobre sus objetos
  origen: un componente que lee la API y publica su contenido
  responden: ¿lo declarado está ocurriendo?
```

La segunda familia es la que casi siempre falta y la que detecta los problemas de esta parte:

```
réplicas deseadas frente a disponibles        la ley de la clase 073
pods en estado no listo, por motivo
reinicios por contenedor
trabajos programados que no se ejecutaron
presupuestos con cero interrupciones permitidas   (clase 079)
reclamaciones de volumen sin vincular             (clase 077)
nodos no listos
certificados próximos a caducar
```

Y la alerta que traduce la ley de la clase 073 a un número, que es la aportación central de esta clase:

```
réplicas declaradas − réplicas disponibles > 0  durante 10 minutos
```

Eso detecta de una vez: un despliegue que no avanza, un pod que no se planifica, una imagen que no se descarga, un volumen que no monta y una comprobación que nunca pasa. Cinco causas distintas y un solo síntoma, que es exactamente lo que se quiere de una alerta de primer nivel.

Y su complemento, que detecta lo que la anterior no ve:

```
un objeto de despliegue existe y su controlador no lo ha reconciliado
  → marca de tiempo del arrendamiento de los controladores (clase 073)
un trabajo programado no ha creado ningún trabajo en dos intervalos
  → la tarea dejó de ejecutarse sin que nada fallara
```

La segunda es la versión de esta parte del fallo silencioso que el programa persigue desde la clase 060: algo que dejó de ocurrir no genera ningún error.

Y sobre el formato, una nota que conecta con la clase 057: el estándar de facto para publicar métricas en Kubernetes es el mismo que las tres nubes admiten de forma nativa. Eso convierte la instrumentación de una aplicación en algo portable de verdad: las mismas métricas y los mismos paneles valen en el clúster, en cualquier nube y en una máquina. Es uno de los pocos contratos del programa que se conserva sin traducción alguna.

3. El orden que acota un diagnóstico

Con las cuatro fuentes y el modelo de la clase 073, el diagnóstico sigue un orden fijo. Saltárselo es lo que convierte diez minutos en dos horas.

```
# 1. ¿qué dice el objeto de sí mismo, y qué sucesos tiene?
$ kubectl describe pod api-7d4b9-x2k4p

# 2. ¿qué dijo el proceso? y si reinició, el ANTERIOR
$ kubectl logs api-7d4b9-x2k4p --previous --tail=100

# 3. ¿qué ve la plataforma sobre el conjunto?
$ kubectl get deploy api -o wide
$ kubectl get rs -l app=api

# 4. ¿qué cambió?
$ kubectl rollout history deploy/api
$ kubectl get events -n tienda --sort-by=.lastTimestamp | tail -20
```

```
# 5. si hace falta entrar, sin intérprete de órdenes (clase 070)
$ kubectl debug -it api-7d4b9-x2k4p --image=nicolaka/netshoot --target=api

# 6. si el problema parece del nodo
$ kubectl describe node nodo-14
$ kubectl debug node/nodo-14 -it --image=busybox
```

Y una tabla que traduce el estado observado al eslabón responsable, ampliando la de la clase 073 con lo aprendido después:

Estado	Fuente que lo explica	Causa habitual
Pending	Sucesos del pod	Recursos, reglas de colocación, volumen zonal (078, 077)
ContainerCreating	Sucesos del pod	Volumen que no monta, secreto que no existe (076, 077)
ImagePullBackOff	Sucesos del pod	Huella inexistente, credenciales, admisión de firma (061, 067)
CrashLoopBackOff	logs --previous	La aplicación sale; el registro del contenedor muerto lo dice
Running y no listo	describe, comprobación de disponibilidad	Dependencia caída (068)
Terminating largo	Sucesos y finalizadores	Proceso que ignora la señal, o controlador ausente (073)
OOMKilled	Estado del contenedor	Límite frente a uso real (063, 078)
Evicted	Sucesos del nodo	Presión del nodo y clase de servicio (078)

La fila de la imagen merece una nota porque su mensaje se lee mal:

```
Failed to pull image "registro/api@sha256:…": … not found
  → la huella no existe: casi siempre, una etiqueta que se resolvió mal
Failed to pull image …: unauthorized
  → credenciales del registro, o identidad del nodo
admission webhook denied the request: no matching signatures
  → la política de la clase 067 está funcionando: eso es un ÉXITO
```

El tercero es un caso en el que el fallo es la señal de que un control funciona, y merece reconocerse para no «arreglarlo».

Y dos órdenes que ahorran tiempo y se conocen poco:

```
# ver el estado de todos los contenedores de un espacio, con motivos
$ kubectl get pods -n tienda -o custom-columns=\
NOMBRE:.metadata.name,LISTO:.status.containerStatuses[*].ready,\
REINICIOS:.status.containerStatuses[*].restartCount,\
MOTIVO:.status.containerStatuses[*].state.waiting.reason

# seguir los sucesos en tiempo real durante un despliegue
$ kubectl get events -n tienda --watch-only
```

La segunda durante un despliegue muestra la secuencia completa —creación, planificación, descarga, arranque, disponibilidad— y es la mejor forma de entender por qué un despliegue tarda lo que tarda.

4. Lo que cuesta, y lo que nadie consulta

La clase 057 estableció el método: mirar el historial de consultas antes de recortar. En un clúster, el resultado de ese ejercicio es casi siempre el mismo.

```
composición típica del volumen de registros de un clúster
  registros de acceso del controlador de entrada     30-50 %
  salida de un componente de plataforma muy locuaz   15-30 %
  aplicaciones                                       15-25 %
  sucesos exportados sin filtrar                      5-15 %
```

Las dos primeras filas suman más de la mitad y casi nunca originan una alerta. Las palancas, en el mismo orden que la clase 057:

```
1. bajar el nivel de registro de los componentes de plataforma
   muchos vienen con nivel de depuración por defecto en su instalación
2. filtrar los accesos correctos del controlador de entrada
   y archivar el volumen íntegro aparte, más barato
3. exportar solo los sucesos que importan, no todos
4. estructurar los registros de las aplicaciones
```

```
   una línea por evento, en formato estructurado, sin trazas de pila repetidas
```

Y una fuente de volumen específica del clúster que sorprende: un contenedor en bucle de reinicio genera registros sin parar, y con espera creciente sigue haciéndolo durante días. Un despliegue roto que nadie retira puede ser la mayor partida del mes.

```
$ kubectl get pods -A --field-selector status.phase!=Running \
  -o custom-columns=NS:.metadata.namespace,N:.metadata.name,\
R:.status.containerStatuses[0].restartCount --no-headers | sort -k3 -rn | head
```

Esa lista debería estar vigilada: un contador de reinicios de cuatro cifras es a la vez un servicio roto y una factura.

Y sobre las métricas, la trampa de coste es la cardinalidad, exactamente como en las clases 034 y 045:

```
una métrica por pod × 400 pods × 15 métricas × 20 etiquetas
  → cientos de miles de series, que se renuevan en CADA despliegue
```

La segunda parte es la propia del clúster: los pods tienen nombres nuevos en cada despliegue, así que cualquier métrica etiquetada con el nombre del pod crea series nuevas continuamente. La corrección es agregar por despliegue o por servicio y no conservar la etiqueta del pod salvo donde haga falta.

Y la lista de comprobación de observabilidad de un clúster:

- recolector de registros: sin él no hay historial
- sucesos exportados, filtrados por tipo y razón
- métricas de estado además de las de recursos
- alerta de réplicas declaradas frente a disponibles
- alerta de arrendamiento de controladores envejecido
- alerta de trabajos programados que dejan de ejecutarse
- alerta de presupuestos con cero interrupciones permitidas (079)
- alerta de reclamaciones sin vincular y nodos no listos
- presión por recurso, por contenedor y por nodo (070)
- nivel de registro de los componentes de plataforma revisado
- etiqueta de pod evitada en métricas de alta cardinalidad
- consultas de diagnóstico guardadas antes de necesitarlas

Doce puntos, de los cuales seis son alertas sobre cosas que dejan de ocurrir — que es la familia de fallos que este programa lleva persiguiendo desde la clase 060 y que en Kubernetes es especialmente fácil de producir, porque casi todo lo hace un bucle y un bucle que no funciona no da error.

5. Lo que ya se puede afirmar sobre observabilidad

Con las clases 034, 045, 057, 070 y esta, el programa ha recorrido cinco implementaciones distintas del mismo problema. Lo que se repite ya no es casualidad:

1. la señal por defecto es la equivocada
   utilización en vez de saturación, medias en vez de percentiles,
   recursos en vez de estado

2. lo que no se recoge mientras ocurre no existe
   registros apagados (045), exportación no retroactiva (049),
   sucesos que caducan (082)

3. el volumen lo domina un puñado de componentes que nadie consulta
   y la comprobación es mirar el historial de consultas, no intuir

4. las alertas útiles son sobre la experiencia y sobre lo que se detiene
   no sobre umbrales de recursos (057)

5. un incidente no es el momento de aprender a consultar
   las consultas se escriben antes y se guardan

Y la aportación propia de Kubernetes a esa lista, que no aparecía en las tres nubes:

6. casi todo lo hace un bucle, y un bucle que no funciona no da error
   → hace falta una métrica de "¿se está cumpliendo lo declarado?"
     y no solo de "¿está sano lo que existe?"

Esa sexta es la que justifica la alerta central de esta clase. En una máquina, si un proceso no arranca, alguien lo ve. En un clúster, un objeto puede estar declarado, aceptado, visible en el inventario y sin ninguna instancia funcionando, indefinidamente y en silencio.

Y dos números que conviene medir en un simulacro, porque son los que resumen si la observabilidad sirve:

```
tiempo hasta detectar          desde que algo se rompe hasta que alguien lo sabe
tiempo hasta localizar la causa desde que se sabe hasta que se sabe por qué
```

Todo lo de las cinco clases existe para bajar esos dos números. Cualquier panel, recolector o alerta que no los baje es decoración con coste mensual, y el simulacro es la única forma de comprobarlo.

Ejemplo trabajado

CloudShop investiga tres incidentes del mes anterior y no puede explicar ninguno. Los cinco hallazgos son sobre lo que faltaba recoger, y el más útil es una alerta que no existía.

Hallazgo 1 — la causa de anoche había caducado.

```
$ kubectl get events -n tienda | grep api-7d4b9
No resources found
```

El pod se había reiniciado siete veces entre las 02:00 y las 04:00. A las 09:00, los sucesos habían desaparecido y los registros del contenedor muerto también, porque el pod se había recreado.

```
sucesos exportados                        ninguno       filtrados, 30 días
registros recogidos                    solo del nodo    recolector a un
                                                        sistema central
historial disponible                       1 hora           30 días
incidentes explicables del mes anterior    0 de 3           —
```

Hallazgo 2 — se leían los registros equivocados.

Durante el diagnóstico de un bucle de reinicio, el equipo leía:

```
$ kubectl logs api-7d4b9-x2k4p --tail=50
[INFO] iniciando…
[INFO] conectando a la base de datos…
```

Siempre las mismas dos líneas, del contenedor que acababa de arrancar. La causa estaba en el que había muerto:

```
$ kubectl logs api-7d4b9-x2k4p --previous --tail=20
[FATAL] configuración inválida: campo 'tiempoEspera' esperaba entero, recibió "30s"
```

Diecinueve horas de diagnóstico por no usar una opción.

```
procedimiento documentado                 no había     con --previous como paso 2
tiempo hasta la causa en un bucle       19 h              4 min
```

Hallazgo 3 — la alerta que faltaba, y lo que destapó al ponerla.

Se añadió la alerta central de esta clase: réplicas declaradas menos disponibles mayor que cero durante diez minutos. En la primera semana disparó cinco veces:

```
causa                                          veces
despliegue detenido por presupuesto (079)        1
volumen sin vincular por zona (077)              1
imagen rechazada por firma (067)                 1
comprobación de disponibilidad que nunca pasaba  1
trabajo programado que llevaba 9 días sin ejecutarse  1
```

Las cinco existían desde antes y ninguna había generado una alerta. La última era la peor: un trabajo programado que sincronizaba precios llevaba nueve días sin ejecutarse porque su objeto había quedado suspendido durante un mantenimiento.

```
alertas de estado                          0                6
situaciones detectadas la primera semana   —                5
tiempo medio de detección                días o nunca       < 10 min
```

Hallazgo 4 — el 61 % del volumen de registros lo generaban dos cosas.

```
controlador de entrada, accesos completos       44 %
un componente de plataforma en modo depuración  17 %
aplicaciones                                    22 %
sucesos sin filtrar                             11 %
resto                                            6 %
```

El componente en modo depuración llevaba así desde su instalación, catorce meses atrás, porque el ejemplo de la documentación lo traía activado.

```
ingesta diaria                          38 GiB/día        9 GiB/día
nivel del componente de plataforma      depuración        aviso
```

```
accesos del controlador de entrada        completos      solo >= 400,
                                                          volumen íntegro
                                                          archivado aparte
costo mensual de registros                 ~570 USD         ~140 USD
capacidad de investigación perdida            —              ninguna
```

Hallazgo 5 — un pod en bucle costaba más que el resto del espacio.

```
$ kubectl get pods -A --field-selector status.phase!=Running \
  -o custom-columns=N:.metadata.name,R:.status.containerStatuses[0].restartCount \
  --no-headers | sort -k2 -rn | head -3
prueba-integracion-8f2   4127
viejo-informes-3a1        918
```

Un pod de una prueba abandonada llevaba once semanas reiniciando cada cuarenta segundos, escribiendo su traza de error en cada arranque.

```
pods en bucle sin dueño                       2                0
reinicios acumulados                      5.045                —
volumen que generaban                  4,1 GiB/día             0
alerta sobre contadores de reinicio       ninguna        > 20 en 1 h → aviso
```

Y la medición que cierra el ejercicio:

```
tiempo hasta detectar (simulacro)         no se detectaba    3 min 10 s
tiempo hasta localizar la causa               horas           6 min
incidentes del mes explicables              0 de 3            3 de 3
ingesta diaria                             38 GiB/día       9 GiB/día
costo mensual de observabilidad             ~640 USD         ~190 USD
alertas sobre cosas que dejan de ocurrir          0                6
consultas de diagnóstico guardadas                0                7
```

La lección que esta clase traslada al proyecto de la clase 084: los tres incidentes del mes anterior eran explicables y la información existió — durante una hora los sucesos y durante unos minutos los registros. Lo que faltaba no era capacidad de análisis sino conservación, y su corrección costó menos que lo que se ahorró recortando el volumen que nadie consultaba. Y la alerta central —declarado menos disponible— destapó en una semana cinco situaciones que llevaban entre nueve días y varios meses activas, todas invisibles porque en Kubernetes lo que no ocurre no produce ningún error.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/082-logs-metricas-eventos-y-depuracion/lab.py
```

El laboratorio selecciona el motor de práctica observability 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 diagnostico-kubernetes en evidence/ usando la plantilla indicada.
5. Para proveedor real, sigue sandbox.requires, registra costo y ejecuta destroy.

Evidencia esperada

El artefacto principal es telemetría correlacionada con una pregunta operativa. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye diagnostico-kubernetes 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
No se puede explicar por qué algo falló hace unas horas	Los sucesos caducan en una hora y los registros del contenedor muerto desaparecen con el pod	Exporta sucesos filtrados por tipo y razón, y despliega un recolector de registros: sin él el clúster no tiene historial.
Los registros de un contenedor en bucle no dicen nada	Se están leyendo los del contenedor recién arrancado, no los del que murió	Usa --previous; la causa está siempre en el contenedor anterior.
Un servicio lleva días sin funcionar y nadie se enteró	No hay alerta sobre la distancia entre réplicas declaradas y disponibles	Añade esa alerta: detecta cinco causas distintas con un solo síntoma.
Una tarea programada deja de ejecutarse sin que nada falle	En Kubernetes lo que deja de ocurrir no produce ningún error	Alerta cuando un trabajo programado no crea ejecuciones en dos intervalos consecutivos.
La factura de registros crece sin relación con el tráfico	Un componente de plataforma en modo depuración o un pod en bucle de reinicio	Revisa el nivel de registro de los componentes instalados y vigila los contadores de reinicio.
Las métricas crecen sin control en cada despliegue	Las series están etiquetadas con el nombre del pod, que cambia siempre	Agrega por despliegue o servicio y conserva la etiqueta del pod solo donde de verdad haga falta.

■ 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 responde cada una de las cuatro fuentes, y cuáles dos caducan?
2. ¿Por qué kubectl logs sin --previous es inútil en un bucle de reinicio?
3. ¿Qué cinco causas distintas detecta la alerta de réplicas declaradas frente a disponibles?
4. ¿En qué se diferencian las métricas de estado de las de recursos, y cuál falta casi siempre?
5. ¿Qué aportación propia de Kubernetes se añade a las cinco conclusiones de observabilidad del programa?

Referencias

- Kubernetes (2025). Events and their retention — origen, agregación y caducidad. <https://kubernetes.io/docs/reference/kubernetes-api/cluster-resources/event-v1/>
- Kubernetes (2025). Logging architecture — rotación en el nodo y necesidad de recolector. <https://kubernetes.io/docs/concepts/cluster-administration/logging/>
- Kubernetes (2025). Resource metrics pipeline — para qué sirve la fuente del escalado y qué no es. <https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/>

• kube-state-metrics (2025). Documentation — métricas de estado de los objetos del clúster. <https://github.com/kubernetes/kube-state-metrics/tree/main/docs>

• Kubernetes (2025). Debug running pods — orden de diagnóstico y contenedores efímeros. <https://kubernetes.io/docs/tasks/debug/debug-application/debug-running-pod/>

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

Descargar: Parte 06 en PDF · Manual integral

Anterior	Índice	Siguiente
← 081 · Helm, Kustomize y gestión de paquetes	Parte 06 · Programa	083 · EKS, AKS y GKE: similitudes y diferencias →

Evaluación — 082 Logs, métricas, eventos y depuración

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 diagnostico-kubernetes 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.

083 — EKS, AKS y GKE: similitudes y diferencias

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 4 Laboratorio: decision · Estado: EXECUTABLECORE

Propósito

Comparar las tres plataformas gestionadas de Kubernetes por lo que de verdad diferencia la operación, y no por una tabla de funciones. Tres ejes deciden casi todo: quién decide cuándo actualizas, cuántos pods caben en tu plan de direcciones y cómo obtiene una carga la identidad de la nube. Y la conclusión que interesa al programa es cuantificable: qué parte del trabajo de las clases 073 a 082 se conserva al cambiar de proveedor y qué parte hay que rehacer.

Resultados de aprendizaje

Al finalizar podrás:

1. Delimitar qué gestiona el proveedor y qué sigue siendo trabajo propio.
2. Anticipar una actualización forzada por fin de soporte y lo que puede bloquearla.
3. Calcular cuántos pods caben según el modelo de red de cada plataforma.
4. Federar la identidad de una carga con la nube, acotando la confianza correctamente.
5. Estimar qué se conserva y qué se rehace al cambiar de plataforma gestionada.

Conceptos centrales

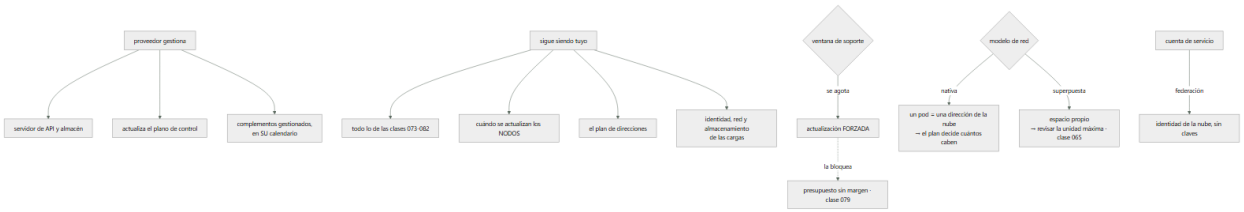
Concepto	Comprensión verificable
plano de control gestionado	El proveedor opera el servidor de API y el almacén de estado. No opera tus cargas ni tus complementos, que es donde está casi todo el trabajo de esta parte.
ventana de soporte	Periodo durante el cual una versión recibe correcciones. Al agotarse, el proveedor actualiza el clúster aunque nadie lo pida.
dirección de pod nativa de la red	Cada pod recibe una dirección del espacio de la nube. Simplifica el enrutamiento y consume el plan de direcciones de las clases 039 y 051.
red superpuesta	Los pods usan un espacio propio y su tráfico se encapsula. Ahorra direcciones y añade una capa —con la trampa de la unidad máxima de transmisión de la clase 065.
identidad federada de carga	La cuenta de servicio del clúster obtiene credenciales de la nube sin ninguna clave. Séptima aparición del mismo contrato del programa.
complemento gestionado	Pieza que el proveedor instala y actualiza en su calendario. Quita trabajo y quita control: su versión puede cambiar sin que nadie lo decida.
desviación de versiones	Diferencia admitida entre el plano de control y los nodos. Permite actualizar por fases y tiene un límite que, si se cruza, deja nodos sin soporte.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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

■ Flujo de razonamiento



Desarrollo

1. Qué gestiona el proveedor, y qué no

«Gestionado» se entiende como más de lo que es, y conviene delimitarlo antes de comparar nada:

```
el proveedor gestiona
  el servidor de API y el almacén de estado: disponibilidad, copias, parches
  la actualización del plano de control
  algunos complementos, en su calendario
  la integración con su red, su almacenamiento y su identidad

sigue siendo trabajo propio
  ABSOLUTAMENTE todo lo de las clases 073 a 082
  cuándo y cómo se actualizan los NODOS
  el plan de direcciones y sus consecuencias
  las políticas de red, de admisión y de permisos
  la observabilidad, las copias de las cargas con estado y su restauración
```

Es decir: el proveedor quita el trabajo que menos se nota y deja el que produce los incidentes de esta parte. Y merece decirse con claridad porque la expectativa contraria lleva a no dedicar equipo:

> Un clúster gestionado no reduce el trabajo de operar aplicaciones en Kubernetes. Reduce el de operar Kubernetes.

Sobre el coste del plano de control, las tres cobran de forma distinta y la cifra rara vez decide:

```
una tarifa por hora y por clúster, con independencia del tamaño
o una tarifa de gestión con algún nivel gratuito acotado
o un nivel gratuito sin acuerdo de servicio y uno de pago con él
```

Lo que sí decide es la consecuencia estructural: un clúster tiene un coste fijo, así que muchos clústeres pequeños cuestan más que uno grande. Y esa aritmética empuja a compartir clúster entre equipos, lo que devuelve al aislamiento de la clase 080 y a su límite: los espacios de nombres no bastan para inquilinos no confiables.

Y la decisión que aparece pronto, con la misma forma en las tres plataformas:

```
un clúster por entorno            frontera clara, coste fijo × entornos
un clúster con espacios por equipo  más barato, aislamiento más débil
un clúster por equipo y entorno     lo más aislado y lo más caro de operar
```

La respuesta razonable en la mayoría de las organizaciones es un clúster por entorno con espacios de nombres por equipo, y producción separada de todo lo demás — que es la misma conclusión a la que llegaron las clases 025, 037 y 049 con cuentas, suscripciones y proyectos. Cuarta aparición del mismo criterio.

2. Quién decide cuándo actualizas

Este es el eje que más cambia la operación y el que menos aparece en las comparativas.

Cada versión tiene una ventana de soporte de aproximadamente un año, y al agotarse el proveedor actualiza el clúster:

```
se publica una versión              cada ~4 meses
soporte estándar                    ~12-14 meses desde su publicación
soporte extendido, si existe         más meses, con sobrecoste
al agotarse                         ACTUALIZACIÓN AUTOMÁTICA por el proveedor
```

De ahí salen tres consecuencias operativas:

Actualizar es una tarea recurrente, no un proyecto. Con una ventana de un año y una versión cada cuatro meses, un clúster se actualiza dos o tres veces al año, siempre. Un equipo que trate cada actualización como un proyecto excepcional vive permanentemente en uno.

Lo que bloquea la actualización lo has puesto tú. El proveedor respeta los presupuestos de interrupción de la clase 079, así que un presupuesto sin margen bloquea también su actualización automática. Y ahí ocurre lo peor:

```
el proveedor intenta actualizar
un presupuesto sin margen bloquea el vaciado del nodo
la actualización se detiene o se agota su plazo
el clúster se queda con nodos en dos versiones, o sin actualizar
→ y el aviso llega por correo a una dirección que nadie lee
```

Es el incidente de la clase 079 con un origen distinto y una consecuencia mayor, porque termina con nodos ejecutando una versión sin soporte.

La desviación de versiones limita el orden. El plano de control se actualiza primero y los nodos después, y hay un margen máximo entre ambos:

```
los nodos pueden ir por detrás del plano de control, hasta un límite
nunca por delante
→ y saltarse versiones intermedias no está soportado:
  hay que pasar por cada una
```

La última línea es la que convierte un retraso en un problema: un clúster tres versiones por detrás necesita tres actualizaciones encadenadas, cada una con su ventana y su riesgo.

Y las funciones que se retiran son la otra mitad del trabajo. Cada versión elimina versiones de API antiguas, y un manifiesto que las use deja de aplicarse:

```
$ kubectl get --raw /metrics | grep apiserver_requested_deprecated_apis
```

Esa métrica dice qué APIs obsoletas se están usando ahora mismo y quién. Debería revisarse antes de cada actualización, y es la comprobación que evita descubrir el problema con el clúster ya actualizado.

Y una recomendación que se paga sola: un clúster de pruebas una versión por delante del de producción. Cuesta el plano de control y unos nodos pequeños, y convierte cada actualización en algo ya ensayado.

3. Cuántos pods caben: el modelo de red decide

Aquí está la divergencia real entre plataformas, y tiene consecuencias que se descubren tarde.

Hay dos modelos:

```
DIRECCIONES NATIVAS
  cada pod recibe una dirección del espacio de la nube
  ventaja: el enrutamiento es directo; los cortafuegos y el registro de flujo
           de las clases 039 y 051 ven al pod, no al nodo
  coste:   consume el plan de direcciones, y mucho

RED SUPERPUESTA
  los pods usan un espacio propio y el tráfico se encapsula
  ventaja: no consume direcciones de la nube
  coste:   una capa más, y la unidad máxima de transmisión de la clase 065
```

Y el cálculo que hay que hacer antes de crear el clúster, no después:

```
con direcciones nativas y una subred /22 (1.024 direcciones)
  reservadas por la nube                    ~5
  una por nodo                              × nodos
  una por pod                               × pods
  y direcciones en tránsito durante los despliegues, que no se liberan al instante

40 nodos × 30 pods = 1.200 direcciones necesarias
→ una /22 NO llega; hace falta al menos una /21, con margen
```

La última línea del cálculo es la que se olvida: durante un despliegue coexisten pods viejos y nuevos, así que el pico de direcciones es mayor que el estado estable. Y el síntoma del agotamiento es característico y despista:

```
pods en ContainerCreating de forma intermitente, sobre todo al desplegar
  failed to allocate for range 0: no IP addresses available in range
```

Parece un problema del complemento de red y es un problema de aritmética de la clase 051.

Y hay un segundo límite propio de algunos modelos: el número de pods por nodo puede depender del tipo de máquina, porque las direcciones se asignan al nodo en bloques ligados a sus interfaces. Un nodo pequeño puede admitir muy pocos pods aunque le sobren CPU y memoria:

```
síntoma   nodos con recursos libres y pods en Pending
          el suceso menciona pods insuficientes, no CPU ni memoria
```

Ese caso se corrige con máquinas mayores o con modos que reparten direcciones por prefijo en vez de una a una, y hay que conocerlo porque contradice la intuición de la clase 078: no siempre falta capacidad; a veces faltan direcciones.

Y los rangos secundarios que la clase 051 pedía reservar aparecen aquí con su factura: en las plataformas que los usan, pods y servicios consumen bloques propios, grandes y fijados al crear el clúster. Ampliarlos después no siempre es posible, lo que convierte el plan de direcciones en otra decisión de un solo sentido.

Y la conclusión que enlaza las dos cosas:

```
el tamaño máximo del clúster lo decide, en la práctica,
el plan de direcciones que alguien hizo antes de crearlo
```

4. La identidad, por séptima vez

Las tres plataformas resuelven lo mismo con el mismo mecanismo y distinta sintaxis: una carga del clúster obtiene credenciales de la nube sin ninguna clave.

```
la cuenta de servicio del pod recibe un testigo firmado por el clúster
la nube confía en el emisor del clúster
y cambia ese testigo por credenciales suyas
```

Es el mismo contrato de las clases 026, 038, 050, 069 y 080. Séptima aparición, con la misma pieza crítica:

```
la confianza del lado de la NUBE debe acotarse al espacio de nombres
y a la cuenta de servicio concretos
```

Sin acotar, cualquier pod del clúster puede pedir esa identidad. Y es exactamente el mismo fallo que el sujeto sin acotar de las federaciones anteriores, con un cuarto vocabulario:

```
condición de confianza correcta
  emisor = el del clúster
  sujeto = system:serviceaccount:<espacio>:<cuenta>
```

La comprobación es la misma prueba negativa que este programa ha ejecutado cuatro veces:

```
# desde un pod con la cuenta correcta
$ kubectl exec -n tienda api-x -- /probar-acceso-nube
ok
# desde un pod de otro espacio con otra cuenta
$ kubectl exec -n desarrollo prueba-y -- /probar-acceso-nube
acceso denegado                                                  ✓
```

Y la parte de la sintaxis, que es lo único que cambia entre plataformas:

```
# el mecanismo es el mismo; la anotación y el nombre del recurso, no
apiVersion: v1
kind: ServiceAccount
metadata:
  name: api
  namespace: tienda
  annotations:
    # <clave específica del proveedor>: <identidad de la nube>
```

Y los complementos gestionados merecen su nota porque cambian el reparto de responsabilidades:

```
complemento gestionado
  el proveedor lo instala y lo actualiza en su calendario
  menos trabajo, y su versión puede cambiar sin que nadie lo decida

complemento propio
  se elige la versión y el momento
  más trabajo, y es la única forma de fijar el comportamiento
```

La decisión importa sobre todo en el complemento de red, por lo que la clase 080 mostró: si no implementa políticas, los objetos no filtran nada. Y algunas plataformas ofrecen varios, con soporte distinto de políticas, así que la elección del complemento al crear el clúster decide si la segmentación de red es posible. Es una decisión de instalación con consecuencias de seguridad, y hay que tomarla con la clase 080 delante.

5. Qué se conserva al cambiar de plataforma

La pregunta que este programa lleva haciendo desde la clase 048 tiene aquí una respuesta cuantificable, y es la mejor noticia de la parte.

```
se conserva SIN CAMBIOS
  todos los manifiestos de carga: despliegues, servicios, configuración,
  presupuestos, políticas de red, perfiles de seguridad, escalado
  las imágenes y su cadena de suministro (clases 061-067)
  el contrato de la aplicación: señales, comprobaciones, apagado (068)
  la instrumentación: el formato de métricas es el mismo (082)
  las herramientas de gestión de manifiestos (081)

hay que REHACER
  la creación del clúster y su plan de direcciones
  la sintaxis de la federación de identidad
  los grupos de nodos y su escalado
  la elección e instalación de complementos
  la integración con el almacenamiento: nombres de clases y parámetros
  la integración con el balanceador de entrada: anotaciones
  la recogida de registros y métricas hacia el sistema del proveedor
```

El primer bloque es grande y el segundo es la capa de plataforma, no las aplicaciones. Expresado como proporción, en una migración real entre dos proveedores:

```
ficheros de manifiesto de aplicaciones      ~95 % sin cambios
capa de plataforma                          ~100 % reescrita
esfuerzo total                              concentrado en 6-8 componentes
```

Y eso confirma con datos lo que la clase 072 predijo: el contrato del contenedor es portable y las fugas están en los bordes. Aquí los bordes tienen nombre: red, almacenamiento, identidad y entrada — las mismas cuatro, por tercera vez.

Y hay una conclusión práctica que conviene sacar y que casi nadie saca: si la capa de plataforma se va a reescribir de todos modos, conviene aislarla desde el principio.

```
repositorio de aplicaciones     manifiestos portables, sin nada del proveedor
repositorio de plataforma       creación del clúster, complementos,
                                clases de almacenamiento, anotaciones de entrada,
                                federación de identidad
```

Con esa separación, cambiar de proveedor toca un repositorio y no doscientos ficheros. Y tiene un beneficio inmediato aunque nunca se cambie: hace visible qué parte del sistema está atada al proveedor, que es una cifra que casi ninguna organización conoce.

Y un aviso honesto sobre la portabilidad, para no exagerarla: que los manifiestos sean portables no significa que el sistema lo sea. Los servicios gestionados que las cargas usan —bases de datos, colas, almacenamiento de objetos— siguen siendo del proveedor, y suelen ser la parte más difícil de mover. Kubernetes hace portable el cómputo, que es la parte que ya era más fácil.

La lista de comprobación para elegir y montar una plataforma gestionada:

- plan de direcciones calculado para el tamaño máximo previsto, con margen de despliegue, y comprobado el límite de pods por nodo
- complemento de red que implementa políticas, verificado con prueba negativa
- federación de identidad acotada a espacio y cuenta, con prueba negativa
- ventana de soporte anotada, con fecha de la próxima actualización obligatoria
- clúster de pruebas una versión por delante
- métrica de APIs obsoletas revisada antes de cada actualización
- presupuestos con margen, para que la actualización del proveedor no se bloquee
- separación entre repositorio de aplicaciones y de plataforma
- complementos: decidido cuáles son gestionados y cuáles propios, y por qué

Ejemplo trabajado

CloudShop opera un clúster en una nube y evalúa mover una parte a otra. El ejercicio se hace en serio —con una migración real de un entorno— y produce cinco datos que ninguna comparativa daba.

Dato 1 — el 94 % de los manifiestos no cambió.

```
manifiestos de aplicaciones                214          13
  de esos 13:
```

```
    clase de almacenamiento                  4
    anotaciones del objeto de entrada        5
    anotaciones de cuenta de servicio        3
    tolerancias de nodos especiales          1
capa de plataforma                          31          31
```

Los trece cambios son exactamente las cuatro fugas de la clase 072, y ninguno estaba en la lógica de la aplicación.

```
reescribir la capa de plataforma          9 días-persona
adaptar los 13 manifiestos                1 día-persona
validar y comparar                        4 días-persona
```

Y la conclusión que se documentó: el coste de cambiar de plataforma gestionada está en seis componentes de infraestructura, no en las aplicaciones.

Dato 2 — el plan de direcciones limitaba el clúster a 27 nodos.

Al dimensionar el clúster nuevo:

```
subred asignada                   /22 · 1.024 direcciones
modelo                            direcciones nativas
pods por nodo previstos           30
nodos que caben                   1.024 / 31 ≈ 33
menos margen de despliegue (~20 %)  ≈ 27
```

El clúster original tenía el mismo problema y no lo sabían: llevaban meses sin poder pasar de 24 nodos, y lo atribuían a cuotas del proveedor.

```
$ kubectl describe node nodo-14 | grep -i 'pods:'
  pods:  29        ← límite por tipo de máquina, no por CPU ni memoria
```

```
subred del clúster                          /22              /20
nodos posibles                               27              110
límite de pods por nodo                      29               110 (modo por prefijo)
sucesos "no IP addresses available"     41 en un mes           0
pods en Pending por límite de pods       sí, en los picos       no
```

Dos correcciones distintas: la subred resolvía el techo del clúster y el modo de asignación por prefijo resolvía el techo por nodo. Ninguna se puede aplicar sobre un clúster existente sin recrearlo, lo que confirma que es una decisión de un solo sentido.

Dato 3 — una actualización forzada que llevaba tres meses bloqueada.

```
versión del clúster            fin de soporte hace 6 semanas
intentos del proveedor         4, todos detenidos
nodos actualizados             11 de 24
motivo                         presupuesto sin margen en 2 servicios (clase 079)
aviso                          correo a una lista que nadie leía
```

El clúster llevaba seis semanas con la mitad de los nodos en una versión sin soporte, y con el plano de control ya actualizado por el proveedor.

```
presupuestos que bloquean                    2                0
notificación del proveedor              correo a lista    alerta en el canal
                                                           de guardia
clúster de pruebas por delante            no había          sí, una versión
métrica de APIs obsoletas revisada         nunca      antes de cada actualización
duración de la actualización completa    no terminaba      4 h 10 min
```

Y la revisión de APIs obsoletas, hecha por primera vez, encontró dos manifiestos que la versión siguiente habría rechazado.

Dato 4 — el complemento de red decidía si la segmentación era posible.

La plataforma de destino ofrecía dos complementos, y solo uno implementaba políticas. La elección por defecto del asistente de creación era el otro.

```
políticas de red efectivas                   0               14
direcciones consumidas por pod               1                1
unidad máxima de transmisión            sin cambios      revisada (clase 065)
complemento gestionado por el proveedor      sí               sí
```

Si la migración se hubiera hecho aceptando el valor por defecto, las catorce políticas de la clase 080 habrían vuelto a no filtrar nada — el mismo incidente, en una plataforma nueva, por una casilla del asistente.

Dato 5 — la federación de identidad, sin acotar por defecto.

La primera configuración funcionaba desde el primer intento, lo que hizo sospechar:

```
$ kubectl run prueba -n desarrollo --rm -it --image=… -- /probar-acceso-nube
ok        ← desde OTRO espacio, con OTRA cuenta
```

La condición de confianza del lado de la nube aceptaba cualquier testigo del clúster.

```
condición de confianza              emisor del clúster   emisor + espacio + cuenta
pods que podían obtener la identidad    todos            solo el previsto
prueba negativa                          ninguna       en el guion de verificación
```

Cuarta vez en el programa que esta misma prueba negativa se ejecuta y tercera vez que falla la primera vez.

Resumen de la evaluación:

```
manifiestos de aplicación reescritos         —              13 de 214
capa de plataforma reescrita                 —              31 de 31
nodos que caben en el plan de direcciones    27              110
políticas de red efectivas                   14              14
federación acotada                           sí            sí (tras corregir)
semanas con versión sin soporte              6               0
esfuerzo total de la migración               —           14 días-persona
```

La lección que esta clase traslada al proyecto de la clase 084: la comparación entre plataformas gestionadas no se decide por funciones sino por tres cosas que se fijan al crear el clúster y no se pueden cambiar después — el plan de direcciones, el complemento de red y el modelo de identidad. Las tres son decisiones de un solo sentido, las tres se toman en un asistente de creación en cinco minutos, y las tres deciden el techo de crecimiento, la posibilidad de segmentar y el alcance de una credencial. El resto —el 94 % de los manifiestos— es portable, y esa es la confirmación cuantificada de la hipótesis que abrió la parte 05.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/083-eks-aks-y-gke-similitudes-y-diferencias/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 matriz-kubernetes-administrado 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 matriz-kubernetes-administrado 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 clúster no puede crecer más allá de cierto número de nodos	El plan de direcciones no da para más pods, y el pico de despliegue consume más que el estado estable	Calcula el tamaño máximo previsto con margen antes de crear el clúster; ampliarlo después suele exigir recrearlo.
Hay nodos con CPU y memoria libres y pods en Pending	El límite de pods por nodo depende del tipo de máquina y del modo de asignación de direcciones	Usa máquinas mayores o el modo de asignación por prefijo; el suceso menciona pods insuficientes, no recursos.
El proveedor intenta actualizar y el clúster queda a medias durante semanas	Un presupuesto de interrupción sin margen bloquea también la actualización automática	Mantén margen en los presupuestos y dirige los avisos del proveedor al canal de guardia, no a una lista.
Tras una actualización, algunos manifiestos dejan de aplicarse	La versión nueva retiró versiones de API que esos manifiestos usaban	Revisa la métrica de APIs obsoletas antes de cada actualización y mantén un clúster de pruebas por delante.
Las políticas de red no filtran en la plataforma nueva	El complemento elegido en el asistente de creación no las implementa	Elige el complemento con soporte de políticas y verifica con una prueba negativa antes de dar por buena la migración.
Cualquier pod del clúster obtiene la identidad de la nube	La condición de confianza acepta cualquier testigo del clúster, sin acotar espacio ni cuenta	Acota la confianza al espacio de nombres y a la cuenta concretos, y comprueba desde otro espacio que se deniega.
Se espera que la plataforma gestionada reduzca el trabajo de operación	Gestiona el plano de control, no las aplicaciones ni los complementos	Dimensiona el equipo para el trabajo de las clases 073 a 082, que es el que produce los incidentes.

■ 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 gestiona exactamente el proveedor y qué sigue siendo trabajo propio?
2. ¿Qué ocurre cuando se agota la ventana de soporte y qué puede impedir esa actualización?
3. Calcula cuántos nodos caben en una subred /22 con 30 pods por nodo y margen de despliegue.
4. ¿Por qué la elección del complemento de red al crear el clúster es una decisión de seguridad?
5. ¿Qué proporción de manifiestos se conserva al cambiar de plataforma, y dónde se concentra el esfuerzo?

Referencias

- Kubernetes (2025). Version skew policy — desviación admitida entre plano de control y nodos. <https://kubernetes.io/releases/version-skew-policy/>
- Kubernetes (2025). Deprecated API migration guide — versiones retiradas y cómo detectarlas. <https://kubernetes.io/docs/reference/using-api/deprecation-guide/>
- AWS (2025). Amazon EKS networking and IP address planning — direcciones por nodo y modos de asignación. <https://docs.aws.amazon.com/eks/latest/userguide/eks-networking.html>
- Microsoft (2025). AKS networking concepts — modelos de red y sus consecuencias de direccionamiento. <https://learn.microsoft.com/en-us/azure/aks/concepts-network>

- Google Cloud (2025). GKE cluster networking and IP allocation — rangos secundarios y límites por nodo. <https://cloud.google.com/kubernetes-engine/docs/concepts/alias-ips>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 06 en PDF · Manual integral

Anterior	Índice	Siguiente
← 082 · Logs, métricas, eventos y depuración	Parte 06 · Programa	084 · Proyecto: plataforma Kubernetes portable →

Evaluación — 083 EKS, AKS y GKE: similitudes y diferencias

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 matriz-kubernetes-administrado 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.

084 — Proyecto: plataforma Kubernetes portable

Parte: 06 — Kubernetes y plataformas administradas Nivel: intermedio-avanzado · Horas estimadas: 8 Laboratorio: capstone · Estado: EXECUTABLECORE

Propósito

Integrar las once clases anteriores en una plataforma de Kubernetes que se pueda operar y mover, y calificar la hipótesis de la clase 072. Acertó en las tres afirmaciones y se quedó corta en una palabra: dijo que Kubernetes «renombraría» las cuatro fugas, y lo que hizo fue renombrarlas y añadir un problema propio en cada una. De esa corrección y de los simulacros salen las dos leyes que esta parte aporta, y la primera explica seis incidentes de seis clases distintas.

Resultados de aprendizaje

Al finalizar podrás:

1. Trazar cada decisión de la plataforma hasta la clase que la tomó y su alternativa descartada.
2. Calificar la hipótesis de la clase 072 con evidencia, incluida la parte en que se quedó corta.
3. Enunciar las dos leyes que esta parte añade, con las apariciones que las respaldan.
4. Provocar tres fallos propios de un clúster y medir detección, impacto y recuperación.
5. Entregar una plataforma con su capa portable separada de la del proveedor.

Conceptos centrales

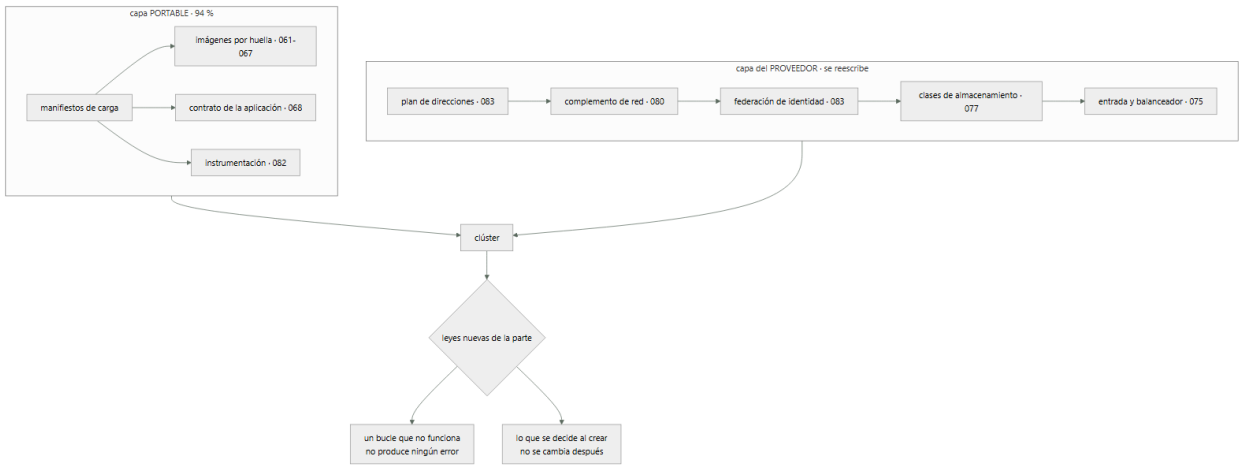
Concepto	Comprensión verificable
ausencia de acción como fallo	En un sistema declarativo, un bucle que no funciona no produce ningún error: simplemente nada ocurre. Es el modo de fallo por defecto, no una excepción.
decisión de creación	Elección que se toma en minutos al crear un recurso y que no se puede cambiar después: plan de direcciones, complemento de red, modelo de vinculación, clave de partición.
capa portable y capa del proveedor	Separación entre lo que se conserva al cambiar de plataforma —los manifiestos— y lo que se reescribe entero —la infraestructura del clúster.
hipótesis calificada	Predicción evaluada después con datos, incluida la parte imprecisa. La de la clase 072 acertó y usó una palabra de menos, y esa palabra es lo que aporta.
concentración tras la recuperación	Efecto de que las reglas de colocación se evalúen solo al planificar: tras una caída, las réplicas se recolocan donde caben y el reparto se pierde en silencio.
alerta de bucle detenido	Señal sobre la marca de tiempo de la última reconciliación de cada controlador. Es la única forma de detectar la ausencia de acción.

Modelo mental

Kubernetes es un sistema de reconciliación: declaras estado deseado y controladores reducen continuamente la diferencia con el estado observado.

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 plataforma 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
Esperar convergencia tras aplicar	Saber que se desplegó (073)	Confiar en el éxito de apply	Tres despliegues que nunca ocurrieron
Migraciones como trabajo, no en inicialización	Una sola ejecución (074)	Contenedor de inicialización	Migración aplicada tres veces
Auxiliares con orden garantizado	Sin errores al arrancar y parar (074)	Contenedor normal	70 errores por despliegue
Lista de destinos como primera comprobación	Diagnóstico rápido (075)	Empezar por los registros	30 min por un selector
Configuración con huella en el nombre	Un cambio es un despliegue (076)	Objeto de nombre fijo	Réplicas divergentes 11 días
Vinculación diferida de volúmenes	Planificar antes de elegir zona (077)	Aprovisionamiento inmediato	Pods en Pending con nodos libres
Solicitudes medidas, no estimadas	Capacidad real (078)	Valores a ojo	24 nodos donde bastaban 4
Presupuesto como máximo indisponible	Poder actualizar (079)	Mínimo disponible	124 días sin parchear
Espacios cerrados por política de red	Aislamiento real (080)	Confiar en el espacio de nombres	Desarrollo alcanzando producción
Validación contra el servidor antes de aplicar	Detectar antes (081)	Aplicar y ver	Selector inmutable en producción
Alerta de declarado frente a disponible	Ver la ausencia de acción (082)	Métricas de recursos	Cinco fallos activos meses
Capa portable separada de la del proveedor	Poder moverse (083)	Todo mezclado	214 ficheros atados a una nube

Y tres decisiones tomadas en contra de lo que sugiere el hábito:

1. La imagen no lleva intérprete de órdenes y el diagnóstico se hace con contenedores efímeros, que además están sujetos a la política de admisión (clases 070, 072).

2. Los presupuestos de interrupción se expresan como máximo indisponible aunque «mínimo disponible» sea más intuitivo: es lo único que sigue siendo correcto al escalar (079).

3. Un índice de búsqueda salió del clúster hacia un servicio gestionado, después de hacer la cuenta completa. Contenerizar no obliga a autogestionar (077).

2. La hipótesis de la clase 072, calificada

La clase 072 dejó escrito:

> Kubernetes resolverá reubicación, despliegue progresivo y reparto de carga, y no resolverá ninguna de las cuatro fugas: las renombrará y devolverá a la aplicación las mismas obligaciones. Y su ley nueva será que aceptado no es funcionando.

Afirmación 1 — resuelve reubicación, despliegue y reparto. CIERTA, con una cifra que corrige la expectativa.

```
reubicación tras caída de nodo    sí, y tarda ~5 min, no segundos (073)
                                  lo que sí tarda segundos es dejar de
                                  enviarle tráfico: 43 s medidos
despliegue progresivo             sí, con vuelta atrás en segundos (074)
reparto de carga                  sí, con la advertencia de las conexiones
                                  persistentes (075)
```

Las tres se cumplen. La corrección útil es la primera: la reubicación existe y no es instantánea, y dimensionar capacidad sin saberlo produce una expectativa falsa.

Afirmación 2 — no resuelve las cuatro fugas. CIERTA, y la palabra «renombrar» se quedó corta.

La clase 083 lo cuantificó: 13 de 214 manifiestos cambiaron al migrar, y los trece son exactamente las cuatro fugas. Pero Kubernetes hizo algo más que renombrarlas: añadió un problema propio en cada una.

```
fuga            renombrada como           y ADEMÁS añadió
████████████████████████████████████████████████████████████████████████████████
almacenamiento  reclamación y clase       el volumen elige zona antes que
                                          el planificador (077)
red             servicio, política,       una política puede existir sin que
                entrada                   nada la implemente (080)
identidad       cuenta de servicio        quien crea pods lee los secretos
                                          del espacio (076)
ciclo de vida   comprobaciones y          la ventana de retirada depende del
                presupuesto               tamaño del clúster (075, 079)
```

Cuatro problemas nuevos, uno por fuga. Así que la formulación correcta es más incómoda que la predicha:

> Kubernetes renombra las cuatro fugas, devuelve las mismas obligaciones y añade una dificultad propia en cada una.

Afirmación 3 — la ley nueva es que aceptado no es funcionando. CIERTA, y ahora medible.

La clase 082 la convirtió en una métrica —réplicas declaradas menos disponibles— y esa métrica destapó cinco situaciones activas en su primera semana. La ley no solo se confirma: se puede vigilar, que es lo que la hace útil.

3. Dos leyes nuevas, con sus apariciones

Ley 13. En un sistema declarativo, un bucle que no funciona no produce ningún error.

Seis apariciones en seis clases de esta misma parte, con seis mecanismos distintos:

```
el gestor de controladores sin reconciliar    tres despliegues no ocurridos   073
un objeto de entrada sin controlador          dos días sin tráfico            075
un objeto de escalado sin métrica             cuatro meses sin escalar        078
un presupuesto con selector desajustado       meses protegiendo a nadie       079
una política de red sin complemento que la
  implemente                                  ocho meses sin filtrar          080
un trabajo programado suspendido              nueve días sin ejecutarse       082
```

Las seis tienen la misma forma: el objeto existe, es válido, figura en el inventario y no hace nada. Ninguna produjo un error, ninguna apareció en un panel y todas se descubrieron por casualidad o al buscarlas.

Esta ley es una especialización de la que la clase 060 corrigió —un mecanismo que parece estar haciendo algo y no lo está— y merece enunciarse aparte porque su causa es distinta: allí era una configuración incompleta; aquí es el modo de fallo por defecto de un modelo declarativo. Si lo que hace el trabajo es un bucle, y el bucle no está, lo que ocurre es nada.

Y su antídoto es único y concreto:

```
cada bucle necesita una señal de "última reconciliación con éxito"
y una alerta cuando esa marca envejece
```

Eso vale para los controladores integrados, para los de terceros y para los propios. Y para lo que no publica esa señal, la alternativa es medir el efecto: la métrica de la clase 082 detecta seis causas distintas sin conocer ninguna.

Ley 14. Lo que se decide al crear un recurso es lo que no se puede cambiar después.

Apariciones en cinco partes distintas:

```
clave de partición de un contenedor            042
modo de una base de datos de documentos        054
modelo de vinculación de una clase de
  almacenamiento                               077
plan de direcciones, complemento de red y
  modelo de identidad de un clúster            083
ubicación y birregión de un bucket             053
```

Y la propiedad que las une es incómoda: son decisiones de cinco minutos en un asistente o en una plantilla, tomadas al principio, cuando menos se sabe. La consecuencia práctica es una regla de método:

```
antes de crear cualquier recurso, preguntar:
  ¿qué de esto no voy a poder cambiar?
  ¿qué tamaño o qué patrón tendrá esto dentro de dos años?
  ¿cuánto cuesta rehacerlo si me equivoco?
```

Tres preguntas que cuestan diez minutos y que en esta parte habrían evitado un clúster con techo de 27 nodos, una clase de almacenamiento que ataba pods a una zona y un complemento de red incapaz de segmentar.

4. Tres fallos provocados y lo que enseñó cada uno

Fallo 1 — pérdida de un nodo. Se apaga un nodo con seis pods de producción.

```
detección: deja de recibir tráfico          41 s
sustitución de los pods                      5 min 12 s
errores vistos por el cliente                0
capacidad durante el episodio                83 %
```

Cero errores, porque el margen de capacidad estaba dimensionado para perder un nodo. La reubicación tardó los cinco minutos que la clase 073 anticipó.

Y el hallazgo: al recuperarse, las réplicas no volvieron a repartirse.

```
$ kubectl get pods -l app=api -o custom-columns=N:.metadata.name,\
ZONA:.metadata.labels.'topology\.kubernetes\.io/zone' --no-headers | awk '{print $2}' | sort | uniq -c
      5 zona-b
      1 zona-c
```

Cinco de seis réplicas en la misma zona. Las reglas de reparto se evalúan al planificar (clase 078), y durante la caída los pods se colocaron donde cabían. Nadie los recolocó después, así que el clúster quedó en un estado en el que la siguiente caída de zona sería total.

```
síntoma observable   ninguno: todos los pods sanos, servicio correcto
consecuencia real    el reparto entre zonas había desaparecido
corrección           reparto entre zonas estricto (no como preferencia),
                     y un reequilibrado periódico que recoloca
señal añadida        alerta si la mayor concentración por zona supera
                     un umbral
```

Fallo 2 — actualizar el clúster con carga en curso.

```
nodos actualizados                          11
duración                                    3 h 40 min
errores HTTP durante la actualización       0
```

Y el hallazgo: cero errores HTTP y, al mirar el camino asíncrono como la clase 072 obligó a hacer:

```
mensajes reentregados durante la actualización   1.847
trabajos programados que no se ejecutaron         2
```

Los mensajes reentregados eran esperables y los absorbió la idempotencia. Los dos trabajos programados no: se saltaron su ventana porque el nodo que los ejecutaba se estaba vaciando y el plazo de arranque (clase 074) los descartó.

```
corrección   los trabajos críticos con plazo de arranque mayor que la ventana
             de vaciado de un nodo, y alerta cuando un trabajo programado
             no crea ejecuciones en dos intervalos (clase 082)
```

Fallo 3 — detener un controlador. Se reduce a cero el controlador de entrada durante veinte minutos, sin avisar al equipo.

```
tiempo hasta que alguien lo notó             18 min
cómo se notó                                 un usuario avisó
alertas disparadas                           1 (disponibilidad del servicio)
alertas que señalaban la CAUSA               0
```

Y el hallazgo: la alerta de disponibilidad detectó el efecto y nada señaló que el controlador no estaba. Se repitió el simulacro deteniendo un controlador cuyo efecto no es inmediato —el que renueva certificados—:

```
tiempo hasta que alguien lo notó             no se notó en 20 min
tiempo hasta que se habría notado            ~40 días, al caducar el certificado
```

Eso es la ley 13 en su forma más pura, y la corrección es la que esa ley pide:

```
señal de última reconciliación con éxito para cada controlador
alerta cuando esa marca envejece más de lo esperado
+ para los certificados, vigilar el VENCIMIENTO desde fuera (clase 058)
```

Los tres hallazgos comparten forma con los de las clases 048, 060 y 072: la infraestructura aguantó los tres fallos y en los tres había algo que ninguna revisión de configuración podía mostrar.

5. La entrega y la pregunta que abre la parte 07

La entrega, sin conocimiento tácito.

```
capa portable       manifiestos, sin nada del proveedor
capa de plataforma  creación, complementos, clases, entrada, identidad
línea base          26 afirmaciones, cada una con su prueba negativa
verificar.sh        ejecuta las 26 y devuelve código de salida
ADR                 12 decisiones con su alternativa descartada
riesgos residuales  5, con responsable y condición de revisión
alertas             14, de las cuales 7 sobre cosas que dejan de ocurrir
procedimientos      3 fallos ensayados, con verificación posterior
línea base medida   rendimiento, despliegue, actualización y coste
```

Los cinco riesgos residuales:

```
1. reubicación de ~5 min ante caída de nodo: aceptada, con margen de capacidad
2. un espacio de nombres no aísla frente a código no confiable (080)
3. el plan de direcciones fija el techo en 110 nodos; revisar a los 80
4. dos complementos gestionados se actualizan en el calendario del proveedor
5. las escrituras del índice migrado siguen en una sola región
```

La comparación con el punto de partida, medida con el mismo método:

```
nodos en horas valle                          24             4
solicitudes reservadas frente a uso        78 / 21 %      34 / 26 %
días sin poder actualizar el clúster         124             0
políticas de red efectivas                 0 de 14        14 de 14
personas con administración total             42             2
errores por despliegue                      20-60             0
errores al vaciar un nodo bajo carga         1.284             0
tiempo de detección de un fallo silencioso  días o nunca    < 10 min
coste mensual del clúster                  3.180 USD      1.120 USD
pruebas negativas ejecutadas                0 de 26        26 de 26
```

Y la pregunta que abre la parte 07.

Esta parte deja una plataforma que funciona y que se mantiene con órdenes ejecutadas por personas o por una canalización. Nada garantiza que dentro de un mes el clúster se parezca al repositorio: un cambio manual sobrevive, un objeto borrado no vuelve, una diferencia entre entornos se acumula — que es exactamente lo que la clase 081 midió al unificar cuatro copias y encontrar treinta y una divergencias que nadie había decidido.

Y resulta que el mecanismo que resuelve eso ya está descrito en esta parte: es el bucle de la clase 073, aplicado a la infraestructura en vez de a las cargas.

> Las clases 047 y 059 trataron la infraestructura como código que alguien ejecuta. Si el modelo declarativo con reconciliación continua es mejor que ejecutar órdenes —y esta parte sugiere que lo es—, ¿por qué la infraestructura se sigue gestionando con ejecuciones? ¿Qué se gana al ponerle un bucle, y qué problema nuevo aparece cuando dos bucles creen que mandan sobre el mismo recurso?

La hipótesis que se escribe ahora, para poder equivocarse de forma comprobable:

> El bucle continuo eliminará la desviación entre repositorio y realidad, y hará visible una clase de problema que hoy no se ve: la frontera de propiedad entre quienes escriben sobre el mismo objeto. Aparecerá al menos una de las catorce leyes con un mecanismo nuevo, y la más probable es la 13 — porque un bucle de infraestructura que se detiene tampoco produce ningún error.

La parte 07 la califica.

Ejemplo trabajado

Entrega del capstone de la parte 06, con las cifras que se llevan a la parte 07.

Verificación completa. Las 26 afirmaciones de la línea base:

```
$ ./verificar.sh
✓ despliegue verificado por convergencia        rollout status, no apply
✓ imágenes por huella en todos los manifiestos  0 etiquetas
✓ firma verificada en la admisión               imagen sin firma → rechazada
✓ migraciones como trabajo previo               0 en inicialización
✓ auxiliares con orden garantizado              7 de 7
✓ lista de destinos no vacía en todos los servicios  19 de 19
✓ solo el frontal publicado                     1 objeto de entrada
✓ resolución de nombres con nombre absoluto     4 servicios corregidos
✓ configuración con huella en el nombre         9 de 9
✓ ningún secreto en el repositorio              0 coincidencias
✓ secretos desde el gestor externo              11 de 11
✓ cifrado en reposo del almacén                 activo
✓ vinculación diferida en clases zonales        3 de 3
✓ política de recuperación de conservación      3 de 3
✓ restauración de volumen probada               1.284.391 filas · 2 h 40 min
✓ solicitudes y límites en todos los contenedores  74 de 74
✓ ningún pod en mejor esfuerzo en producción    0
✓ presupuestos con margen                       12 de 12, permitidas ≥ 1
✓ ningún presupuesto con selector vacío         0
✓ espacios cerrados por política de red         6 de 6
✓ política de red efectiva                      desarrollo → producción: rechazado
✓ perfil de seguridad en modo rechazo           6 de 6
✓ federación acotada a espacio y cuenta         otro espacio → denegado
✓ ningún testigo montado sin necesidad          4 de 74
✓ despliegue con carga: cero errores            0 HTTP · 0 reentregas
✓ vaciado de nodo con carga: cero errores       0
26/26 correctas
```

Línea base medida:

```
rps 981,2 · p50 40,4 ms · p95 95,1 ms · p99 201,8 ms · errores 0
despliegue completo                    1 min 40 s
vuelta atrás                              34 s
actualización del clúster completa     3 h 40 min
reubicación tras caída de nodo         5 min 12 s
retirada de tráfico de un nodo caído      41 s
costo mensual                          1.120 USD
```

Los tres fallos, con lo aprendido en cada uno:

pérdida de un nodo	41 s	0 errores, y el reparto entre zonas desapareció	las reglas de colocación no recolocan nada
actualización con carga	—	0 errores HTTP, 2 trabajos programados saltados	lo que no tiene métrica propia no se verifica
controlador detenido	18 min	lo notó un usuario; 0 alertas señalaban la causa	un bucle que se detiene no da ningún error

El hallazgo que justificó el capstone. Tras el simulacro de nodo, todo estaba en verde:

```
pods listos            6 de 6
servicio               disponible
presupuesto            cumplido
presupuesto de error   intacto
```

Y el reparto:

```
$ kubectl get pods -l app=api -o json | jq -r '.items[]
  | .metadata.labels["topology.kubernetes.io/zone"] // "?"' | sort | uniq -c
      5 zona-b
      1 zona-c
```

Cinco de seis réplicas en una zona, sin ninguna señal. La regla de reparto estaba puesta como preferencia —la corrección de la clase 078 para no bloquear el crecimiento— y durante la caída los pods se colocaron donde cabían. Al recuperarse, nadie los movió.

```
síntoma observable   ninguno
consecuencia real    la siguiente caída de zona habría sido total
causa                las reglas se evalúan al planificar, no después
qué lo destapó       contar las zonas después del simulacro,
                     no el simulacro en sí
```

Corrección y comprobación posterior:

```
reparto entre zonas                  preferencia        estricto
reparto entre nodos                  preferencia        preferencia
reequilibrado                         ninguno       periódico, en ventana
alerta de concentración por zona      ninguna       > 50 % en una zona → aviso
zonas tras repetir el simulacro      5/1/0            2/2/2
```

Se entrega a la parte 07 con:

```
catorce leyes observadas, dos de ellas nuevas de esta parte
las cuatro fugas, con el problema propio que Kubernetes añade a cada una
26 afirmaciones y su guion de verificación
12 decisiones con alternativa descartada
5 riesgos residuales con responsable
la proporción medida de portabilidad: 94 % de manifiestos, 0 % de plataforma
```

Y la hipótesis escrita para la parte 07:

> 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.

La lección que esta parte deja al programa: la hipótesis de la clase 072 acertó en todo y usó una palabra de menos. Kubernetes no solo renombra las cuatro fugas: añade una dificultad propia en cada una, y las cuatro nuevas tienen la misma forma que la ley 13 — un objeto que existe y no hace lo que su nombre promete. De las once clases de la parte, seis produjeron un incidente de esa familia, con seis mecanismos distintos. Esa frecuencia no es mala suerte: es el modo de fallo por defecto de un sistema donde el trabajo lo hacen bucles.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-06-kubernetes-managed-platforms/084-proyecto-plataforma-kubernetes-portable/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-kubernetes 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-kubernetes 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
Tras recuperarse de una caída, todas las réplicas quedan en la misma zona	Las reglas de colocación se evalúan al planificar y nadie recoloca después	Reparto estricto entre zonas, reequilibrado periódico y alerta sobre la concentración máxima por zona.
Una actualización con carga da cero errores y aun así se pierde trabajo	Solo se midió el camino HTTP; los trabajos programados que caían en la ventana de vaciado se saltaron	Amplía el criterio de verificación al camino asíncrono y alerta cuando un trabajo programado deja de crear ejecuciones.
Un controlador se detiene y nadie se entera	En un modelo declarativo la ausencia de acción no produce ningún error	Señal de última reconciliación por controlador con alerta de envejecimiento, y medición del efecto donde no haya señal.
El clúster no puede crecer y hay que recrearlo para arreglarlo	El plan de direcciones y el complemento se fijaron al crear el clúster	Antes de crear cualquier recurso, pregunta qué no se podrá cambiar, qué tamaño tendrá en dos años y cuánto cuesta rehacerlo.
Se espera que la reubicación tras una caída de nodo sea inmediata	Retirar el tráfico tarda segundos; sustituir los pods tarda minutos por diseño	Dimensiona el margen de capacidad para operar sin ese nodo durante la ventana, y documenta ambos tiempos.
Un objeto de seguridad existe y no protege nada	Es la ley 13: el bucle que debía implementarlo no está o el selector no coincide	Toda afirmación de control necesita una prueba negativa que la obligue a actuar.

■ 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. ¿En qué acertó la hipótesis de la clase 072 y qué palabra se quedó corta?
2. Enuncia la ley 13 y cita cuatro de sus seis apariciones con mecanismos distintos.
3. ¿Por qué la ley 14 es especialmente peligrosa, y qué tres preguntas la mitigan?
4. Tras el simulacro de nodo todo estaba en verde y el reparto entre zonas había desaparecido. ¿Por qué, y qué señal lo detecta?
5. ¿Qué proporción de la plataforma es portable y dónde se concentra exactamente lo que no lo es?

Referencias

- Kubernetes (2025). Production environment checklist — consideraciones de un clúster operable. <https://kubernetes.io/docs/setup/production-environment/>
- Kubernetes (2025). Configuration best practices — decisiones que conviene tomar al crear. <https://kubernetes.io/docs/concepts/configuration/overview/>
- CNCF (2025). Kubernetes cluster conformance — qué garantiza la conformidad y qué no. <https://www.cncf.io/certification/software-conformance/>
- Google (2018). The Site Reliability Workbook, cap. 15 — simulacros de fallo y hallazgos. <https://sre.google/workbook/postmortem-culture/>
- Kubernetes (2025). Descheduler — reequilibrado de pods tras cambios de topología. <https://github.com/kubernetes-sigs/descheduler>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 06 en PDF · Manual integral

Anterior	Índice	Siguiente
← 083 · EKS, AKS y GKE: similitudes y diferencias	Parte 06 · Programa	085 · Declarativo, imperativo, idempotencia y convergencia →

Evaluación — 084 Proyecto: plataforma Kubernetes portable

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-kubernetes 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.