Multi-Cloud Engineering Program

Parte 05 — Contenedores, Docker y OCI

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	05 de 23
Clases	061–072
Evaluaciones	12
Horas estimadas	52

Índice

Parte 05 — Contenedores, Docker y OCI

← Parte 04 · Índice completo · Parte 06 →

Descargar: Esta parte en PDF · Manual integral

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

Empaquetar servicios portables, observables y endurecidos.

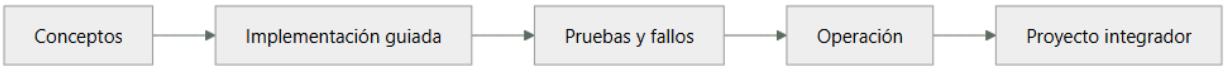
Resultados de la parte

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

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

Secuencia



Clases

ID	Clase	Laboratorio	Horas
061	Imágenes, capas, registros y estándar OCI	container	4
062	Dockerfile reproducible y builds multi-stage	container	4
063	Namespaces, cgroups y runtime de contenedores	container	4
064	Volúmenes, bind mounts y persistencia	storage	4
065	Redes bridge, DNS interno y publicación de puertos	network	4
066	Docker Compose y aplicaciones multiservicio	orchestration	4
067	Registros, SBOM, firma y procedencia de imágenes	supply-chain	4
068	Límites, health checks y apagado ordenado	reliability	4
069	Rootless, capabilities, seccomp y secretos	security	4
070	Diagnóstico de CPU, memoria, red y filesystem	observability	4
071	Migración de una aplicación legacy a contenedores	migration	4
072	Proyecto: stack OCI endurecido y observable	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

- Docker Deep Dive — Nigel Poulton.
- Container Security — Liz Rice.
- Cloud Native Patterns — Cornelia Davis.

061 — Imágenes, capas, registros y estándar OCI

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 4 Laboratorio: container · Estado: EXECUTABLECORE

Propósito

Establecer con precisión qué es una imagen de contenedor y qué garantiza exactamente el estándar OCI, porque de ahí sale la primera mitad del contrato que la clase 060 dejó por comprobar. Una imagen no es un fichero: es un manifiesto que apunta a capas por su huella, y esa propiedad decide tres cosas que en producción se pagan caras — que una etiqueta puede cambiar bajo tus pies, que un fichero borrado sigue dentro, y que la misma etiqueta puede entregar imágenes distintas según la arquitectura de quien la descarga.

Resultados de aprendizaje

Al finalizar podrás:

1. Describir qué normaliza cada una de las tres especificaciones OCI y qué queda fuera del contrato.
2. Desplegar por huella en vez de por etiqueta, y justificar la diferencia con un incidente concreto.
3. Demostrar que un fichero borrado en una capa posterior sigue presente en la imagen.
4. Construir y verificar un índice de imágenes para varias arquitecturas.
5. Leer la configuración de una imagen y distinguir lo que el motor aplica de lo que es solo documentación.

Conceptos centrales

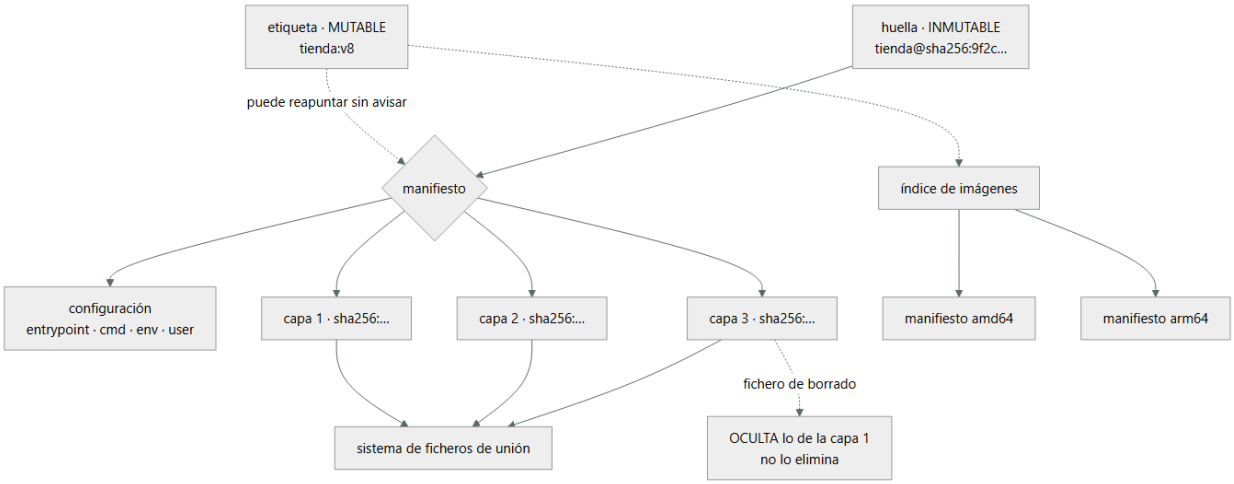
Concepto	Comprensión verificable
manifiesto	Documento JSON que enumera la configuración y las capas de una imagen por su huella. La imagen no es un archivo: es este documento y lo que referencia.
direccionamiento por contenido	Todo se identifica por sha256. Una etiqueta es un puntero mutable; una huella es inmutable y verificable.
índice de imágenes	Manifiesto que apunta a varios manifiestos, uno por arquitectura y sistema operativo. Explica que la misma etiqueta entregue binarios distintos a máquinas distintas.
capa	Archivo tar con las diferencias respecto de la anterior. Las capas se apilan en un sistema de ficheros de unión y una capa nunca modifica a la anterior: la tapa.
fichero de borrado	Marca especial que oculta un fichero de una capa inferior. Lo oculta: no lo elimina, así que sigue extraíble de la imagen.
configuración de la imagen	JSON con punto de entrada, comando, variables, usuario y directorio de trabajo. Es el contrato de ejecución — y parte de sus campos son solo documentación.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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

■ Flujo de razonamiento



Desarrollo

1. Tres especificaciones, y lo que queda fuera

«Contenedor OCI» no es una cosa sino tres normas que encajan:

```
especificación de IMAGEN        cómo se describe y se empaqueta
  manifiesto, configuración, capas, índice para varias arquitecturas

especificación de EJECUCIÓN     cómo se arranca un contenedor a partir de un
  paquete de sistema de ficheros y un `config.json` con espacios de nombres,
  límites y capacidades

especificación de DISTRIBUCIÓN  cómo se sube y se baja de un registro
  una API HTTP con verbos, huellas y un flujo de autenticación
```

Lo que garantizan juntas es exactamente lo que la clase 060 llamó «el contrato del contenedor»: una imagen construida con una herramienta se ejecuta con otro motor y se guarda en cualquier registro, sin acuerdos privados entre fabricantes. Esa portabilidad es real y es lo que hizo que las tres nubes convergieran.

Y conviene ser igual de preciso con lo que no normalizan, porque ahí están las fugas que la hipótesis predijo:

```
fuera del contrato
  cómo se CONSTRUYE la imagen        (Dockerfile no es una norma OCI)
  la red: nombres, DNS, publicación de puertos
  el almacenamiento persistente y su ciclo de vida
  la identidad con la que el contenedor habla con el exterior
  la orquestación: reinicio, escalado, ubicación
  la política de seguridad efectiva del anfitrión
```

Esa lista es el índice de las once clases siguientes, y no es casualidad: todo lo que la norma no cubre es donde cada plataforma decide por su cuenta, y donde una aplicación que «funciona en mi máquina» deja de funcionar en otra.

Una precisión que ahorra confusión de vocabulario:

```
imagen        el artefacto: manifiesto + configuración + capas
contenedor    una ejecución de esa imagen, con su capa de escritura propia
registro      el almacén que sirve manifiestos y capas por huella
motor         quien las descarga y prepara: containerd, CRI-O, Podman, Docker
tiempo de ejecución  quien crea el proceso aislado: runc, crun, youki
```

Docker es un producto que integra varias de esas piezas; ninguna de ellas es «Docker» en el sentido normativo. Saberlo importa porque las plataformas gestionadas de las partes 02 a 04 ejecutan contenedores sin Docker, y aun así ejecutan las mismas imágenes.

2. La etiqueta miente; la huella no

Esta es la regla operativa más importante de la clase, y la que más incidentes evita.

Una imagen se identifica de dos formas:

```
registro/tienda:v8                      ETIQUETA · puntero mutable
registro/tienda@sha256:9f2c4a…          HUELLA   · contenido exacto
```

Una etiqueta es un nombre que apunta a un manifiesto y puede reapuntar en cualquier momento. Nada impide reconstruir y volver a subir v8 con otro contenido. Y latest no tiene ninguna propiedad especial: es una etiqueta por defecto, no «la última».

Las consecuencias en producción son tres y se dan juntas:

```
1. la plataforma puede reiniciar una instancia y descargar OTRA imagen
   sin que nadie haya desplegado nada
2. dos instancias del mismo servicio pueden estar ejecutando
   contenidos distintos con la misma etiqueta
3. la investigación de un incidente no puede reconstruir qué se ejecutaba
```

La comprobación es directa:

```
$ docker inspect --format '{{index .RepoDigests 0}}' registro/tienda:v8
registro/tienda@sha256:9f2c4a1b…

$ crane digest registro/tienda:v8
sha256:9f2c4a1b…
```

Y la regla que se deriva, que debe estar en la línea base de cualquier plataforma:

```
se CONSTRUYE con etiquetas legibles
se DESPLIEGA por huella
la canalización resuelve la etiqueta a huella una vez y propaga la huella
```

Esa resolución única es la pieza práctica: la canalización descubre la huella al publicar y la escribe en el manifiesto de despliegue, de modo que lo que se revisó y lo que se ejecuta son verificablemente lo mismo — el mismo argumento del plan guardado de la clase 059.

Complementariamente, los registros permiten etiquetas inmutables, que rechazan la reescritura:

```
$ gcloud artifacts repositories update cls --immutable-tags
$ aws ecr put-image-tag-mutability --repository-name tienda --image-tag-mutability IMMUTABLE
```

Con eso, v8 significa siempre lo mismo. Es una defensa en profundidad y no sustituye a desplegar por huella: la etiqueta inmutable protege de la reescritura y no protege de que alguien despliegue una etiqueta distinta de la revisada.

Y hay una tercera cara del mismo problema, que aparece cuando alguien intenta "limpiar": borrar una etiqueta no borra el contenido, y borrar un manifiesto puede dejar imágenes rotas si comparten capas y se aplica una recolección agresiva. Los registros lo resuelven con recolección de basura por referencias; la regla de operación es no borrar por antigüedad sin comprobar que nada en producción apunta a esa huella.

3. Las capas apilan y tapan: lo borrado sigue dentro

Una imagen se compone de capas, cada una un archivo tar con las diferencias respecto de la anterior. Se montan apiladas en un sistema de ficheros de unión, y de ahí salen dos propiedades con consecuencias opuestas.

La buena: las capas se comparten. Dos imágenes con la misma base comparten esas capas en el registro y en el disco del nodo. Se descargan una vez. Ese es el argumento real para tener una base común en la organización, y se mide:

```
12 servicios con bases distintas   descarga inicial por nodo: 3,4 GB
12 servicios con base común        descarga inicial por nodo: 780 MB
```

La peligrosa: una capa no modifica a la anterior, la tapa. Borrar un fichero en una capa posterior escribe un fichero de borrado que lo oculta en la vista final. El contenido original sigue en la capa donde entró:

```
COPY .npmrc /root/.npmrc      # capa 3: el token entra en la imagen
RUN npm ci && rm /root/.npmrc # capa 4: lo oculta, no lo elimina
```

Y se demuestra en tres órdenes:

```
$ docker save registro/tienda:v8 -o imagen.tar
$ mkdir -p x && tar -xf imagen.tar -C x
$ for capa in x/blobs/sha256/*; do
    tar -tzf "$capa" 2>/dev/null | grep -q '\.npmrc' && echo "token en $capa"
  done
token en x/blobs/sha256/4b1e9c…
```

Cualquiera que pueda descargar la imagen puede extraer ese fichero. Un secreto que ha estado en una capa está comprometido, y la respuesta no es cambiar el Dockerfile: es rotar el secreto y después cambiar el Dockerfile. La forma correcta de construir sin dejar rastro es de la clase 062.

La misma propiedad explica por qué «limpiar al final» no reduce el tamaño:

```
RUN apt-get install -y build-essential   # capa: +380 MB
RUN apt-get remove -y build-essential    # capa: +2 KB de marcas de borrado
# la imagen sigue pesando 380 MB más
```

Y una precisión sobre el tamaño que evita optimizar lo que no importa: lo que cuesta tiempo no es el tamaño de la imagen sino las capas que el nodo aún no tiene. Un servicio que cambia solo su capa de aplicación descarga unos megabytes aunque la imagen pese 900. Por eso el orden de las capas —lo estable abajo, lo volátil arriba— importa más que el total, y es el eje de la clase siguiente.

Un último detalle que aparece al inspeccionar imágenes ajenas: el historial de construcción viaja dentro de la configuración, así que las órdenes ejecutadas son legibles por cualquiera que tenga la imagen:

```
$ docker history --no-trunc registro/tienda:v8 | head -5
```

Un argumento con una contraseña en línea queda ahí registrado aunque el fichero no exista en el sistema de ficheros final.

4. Una etiqueta, varias arquitecturas

El índice de imágenes es un manifiesto que apunta a otros manifiestos, uno por plataforma:

```
$ docker manifest inspect registro/tienda:v8 | \
    jq -r '.manifests[] | "\(.platform.os)/\(.platform.architecture)  \(.digest[0:19])"'
linux/amd64   sha256:9f2c4a1b3d5e
linux/arm64   sha256:71ae0c9d2f48
```

El motor elige el que corresponde a su máquina. Por eso docker pull tienda:v8 entrega binarios distintos en un portátil con procesador ARM y en un servidor amd64, con la misma etiqueta y sin que nada lo indique.

El fallo que produce es característico y desconcierta la primera vez:

```
exec /usr/local/bin/app: exec format error
```

No es una imagen corrupta: es una imagen para otra arquitectura. Ocurre cuando alguien construye en su portátil y publica, porque la construcción local produce solo la plataforma del anfitrión.

La construcción correcta declara las plataformas:

```
$ docker buildx build --platform linux/amd64,linux/arm64 \
    -t registro/tienda:v8 --push .
```

Y la verificación forma parte de la publicación, no de la confianza:

```
$ docker manifest inspect registro/tienda:v8 | jq -e \
    '[.manifests[].platform.architecture] | contains(["amd64","arm64"])' \
  && echo "índice correcto"
```

Dos matices operativos que conviene conocer antes de adoptar varias arquitecturas:

La construcción cruzada por emulación es lenta. Construir arm64 sobre amd64 emulando puede multiplicar por cinco o por diez el tiempo, sobre todo si hay compilación. La alternativa es construir cada plataforma en un nodo nativo y unir los manifiestos, que es más trabajo de canalización y mucho más rápido.

No todo se comporta igual en ambas. Dependencias con extensiones nativas, bibliotecas con rutas de código específicas y diferencias de rendimiento por núcleo hacen que «funciona en amd64» no implique «funciona en arm64». Si se publican las dos, hay que probar las dos; publicar una arquitectura sin ejecutar sus pruebas es publicar una promesa.

Y una tercera diferencia que muerde en desarrollo: un portátil ARM ejecutando una imagen amd64 por emulación puede ser entre tres y diez veces más lento, lo que lleva a diagnosticar problemas de rendimiento que no existen en producción.

5. La configuración: lo que el motor aplica y lo que solo documenta

La configuración de la imagen es el contrato de ejecución. Conviene leerla entera al menos una vez:

```
$ docker inspect registro/tienda:v8 --format '{{json .Config}}' | jq
{
  "User": "10001",
  "Env": ["PATH=/usr/local/bin:…", "NODE_ENV=production"],
  "Entrypoint": ["/usr/local/bin/app"],
```

```
  "Cmd": ["--puerto=8080"],
  "WorkingDir": "/app",
  "ExposedPorts": {"8080/tcp": {}},
  "Labels": {"org.opencontainers.image.revision": "a1b2c3d"}
}
```

Y separar lo que tiene efecto de lo que no, porque la mitad de las suposiciones erróneas están aquí:

```
tiene efecto
  Entrypoint y Cmd   qué proceso arranca y con qué argumentos
  Env                variables presentes en el proceso
  User               identidad efectiva del proceso
  WorkingDir         directorio inicial

es solo documentación
  ExposedPorts       NO publica nada ni abre ningún puerto
  Labels             metadatos; útiles y sin efecto en la ejecución
```

ExposedPorts declara una intención para quien lea la imagen. Publicar un puerto es una decisión de ejecución, de la clase 065. Creer lo contrario lleva a desplegar un servicio inalcanzable y a buscar el problema en la imagen.

Y tres precisiones sobre los campos que sí tienen efecto:

Entrypoint y Cmd se combinan: el segundo son los argumentos por defecto del primero, y se sustituyen al ejecutar. La forma de lista —no la de cadena— es la correcta, porque la de cadena arranca un intérprete de órdenes intermedio que se traga las señales, y ahí está la mitad del problema de apagado ordenado de la clase 068.

Env queda escrito en la imagen. Una variable con una contraseña es un secreto publicado, legible con docker inspect por cualquiera que tenga la imagen. Las variables de la imagen sirven para configuración no sensible y valores por defecto; lo sensible se inyecta al ejecutar.

User es un identificador numérico en la práctica. El motor resuelve un nombre contra el /etc/passwd de la imagen, y las plataformas suelen exigir un número para poder aplicar políticas. Declarar USER 10001 en vez de un nombre evita que una política de «no ejecutar como root» no pueda verificarlo.

Y las etiquetas estándar merecen ponerse siempre, porque son lo que permite responder «de qué código salió esto» sin adivinar:

```
org.opencontainers.image.source     repositorio
org.opencontainers.image.revision   commit exacto
org.opencontainers.image.created    fecha de construcción
```

Con la huella en el despliegue y la revisión en la etiqueta, la cadena queda cerrada: del contenedor en ejecución al commit, sin preguntar a nadie. Es la misma trazabilidad que las clases 047 y 059 pedían para la infraestructura, aplicada al artefacto.

Ejemplo trabajado

CloudShop empaqueta sus servicios en contenedores. Las imágenes se construyen y funcionan en la primera tarde; los cinco problemas del primer mes son todos propiedades de la imagen que nadie había tenido que mirar.

Punto de partida:

```
construcción en el portátil de cada desarrollador
etiqueta `latest` en los despliegues
registro público como base, sin espejo
una imagen por servicio, cada una con su base
```

Problema 1 — producción cambió sin que nadie desplegara.

Un servicio empieza a fallar a las 11:40. Nadie ha desplegado desde hace cuatro días.

```
$ kubectl get pods -o jsonpath='{.items[*].status.containerStatuses[*].imageID}' | tr ' ' '\n' | sort
-u
registro/tienda@sha256:9f2c4a1b…
registro/tienda@sha256:c74e0182…
```

Dos huellas distintas con la misma etiqueta. Una instancia se había reiniciado a las 11:38 y descargó la imagen que una canalización de pruebas había publicado sobre latest esa mañana.

```
referencia en el despliegue                etiqueta          huella
etiquetas inmutables en el registro           no               sí
```

```
resolución de etiqueta a huella          en cada nodo     una vez, en la canalización
versiones distintas en producción a la vez     2               0
reconstruir qué se ejecutaba             imposible       trivial: la huella
```

Problema 2 — la mitad de la flota no arranca.

```
exec /usr/local/bin/app: exec format error
```

La imagen se había construido en un portátil con procesador ARM y publicada tal cual. Los nodos amd64 no podían ejecutarla; los nodos ARM sí, y por eso el fallo parecía intermitente.

```
plataformas en el índice                    1 (arm64)      amd64 + arm64
construcción                          portátil de cada uno   canalización
                                                             con nodos nativos
verificación del índice al publicar          ninguna     obligatoria, falla el paso
duración de la construcción                  4 min          6 min
```

Seis minutos en vez de cuatro, con nodos nativos por arquitectura. La alternativa por emulación tardaba 21.

Problema 3 — un token de registro de paquetes dentro de la imagen.

Una revisión de seguridad extrae las capas:

```
$ for capa in x/blobs/sha256/*; do
    tar -tzf "$capa" 2>/dev/null | grep -q '\.npmrc' && echo "$capa"
  done
x/blobs/sha256/4b1e9c…
$ tar -xzf x/blobs/sha256/4b1e9c… -O root/.npmrc | grep _authToken
//registro.interno/:_authToken=npm_A7f2…
```

El Dockerfile lo borraba en la orden siguiente. La imagen llevaba once semanas publicada en un registro con lectura para toda la organización.

```
token en una capa                              sí              no
método de construcción                 COPY + rm         montaje de secreto (062)
token rotado                                   —             sí, el mismo día
comprobación de secretos en capas          ninguna     en la canalización, bloquea
```

El orden importó: primero rotar, después corregir. Un secreto que estuvo en una capa publicada se considera comprometido, exactamente igual que el del historial de despliegues de la clase 047 y el del estado de Terraform de la clase 059. Tercera vez en el programa.

Problema 4 — los despliegues fallan a las horas punta.

```
toomanyrequests: You have reached your pull rate limit
```

Todas las imágenes partían de una base descargada del registro público en cada construcción y en cada nodo nuevo.

```
origen de las imágenes base            registro público   espejo propio
descargas externas por día                   ~900              12
despliegues fallidos por límite de tasa    7 en un mes          0
base común de la organización                no          sí, 3 variantes
capas descargadas por nodo nuevo            3,4 GB          780 MB
```

La base común no se adoptó por el ahorro de descarga sino por poder parchear una vulnerabilidad en un sitio; la reducción de 3,4 GB a 780 MB fue una consecuencia.

Problema 5 — fallos intermitentes de resolución de nombres.

Al pasar la imagen a una base minimalista con otra biblioteca de sistema, aparecieron fallos de DNS que solo se daban con ciertos nombres largos y bajo carga.

```
causa    la biblioteca de sistema de esa base resuelve nombres de forma
         distinta: manejo diferente de dominios de búsqueda y del cambio
         a TCP cuando la respuesta no cabe en un paquete UDP
```

```
base                                    minimalista con     minimalista con
                                        otra biblioteca     biblioteca estándar
tamaño de la imagen                         78 MB            121 MB
fallos de resolución por día                 ~40                0
```

Cuarenta y tres megabytes más por eliminar una clase entera de fallos intermitentes. La lección que se anota: una base más pequeña no es gratis, y la diferencia no está en el tamaño sino en qué implementación de las bibliotecas del sistema trae.

Resumen del empaquetado:

```
referencia en el despliegue                  etiqueta        huella
versiones distintas en producción a la vez        2             0
plataformas publicadas                            1             2
secretos extraíbles de las capas                  1             0
despliegues fallidos por límite del registro      7             0
capas descargadas por nodo nuevo               3,4 GB        780 MB
fallos de resolución de nombres al día          ~40             0
trazabilidad de contenedor a commit         imposible      etiqueta estándar
```

La lección que esta clase traslada al resto de la parte 05: el contrato OCI garantiza que la imagen se ejecute en cualquier sitio, y no garantiza nada sobre cuál imagen es. Las cinco incidencias se reducen a dos preguntas que hay que poder responder siempre: qué huella exacta se está ejecutando, y qué hay dentro de sus capas. Las dos se responden con una orden, y ninguna de las dos se responde con una etiqueta.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/061-imagenes-capas-registros-y-estandar-oci/lab.py
```

El laboratorio selecciona el motor de práctica container 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 inspeccion-imagen 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 imagen mínima, escaneada y ejecutada sin privilegios. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye inspeccion-imagen 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
Producción cambia de comportamiento sin que nadie despliegue	El despliegue referencia una etiqueta, que es un puntero mutable	Resuelve la etiqueta a huella en la canalización y despliega por huella; activa además etiquetas inmutables en el registro.
exec format error en parte de la flota	La imagen se construyó para una sola arquitectura, la del equipo que la construyó	Construye un índice con todas las plataformas de destino y verifica su contenido como paso obligatorio de la publicación.
Un secreto sigue siendo extraíble aunque el Dockerfile lo borre	Una capa posterior oculta el fichero pero no lo elimina de la capa donde entró	Rota el secreto primero y construye después con montajes de secreto en vez de copiarlo.
Los despliegues fallan por límite de descargas del registro público	Cada construcción y cada nodo nuevo descargan la base desde fuera	Usa un espejo propio y una base común de la organización, que además comparte capas entre servicios.
Un servicio publica un puerto y es inalcanzable	EXPOSE es documentación: no publica nada	Publica el puerto en la ejecución o en la plataforma; la imagen solo declara la intención.
Una imagen más pequeña introduce fallos intermitentes de red o de fecha	La base minimalista trae otra implementación de las bibliotecas del sistema	Valida la base con las mismas pruebas de integración; el tamaño no es el único criterio.

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 normaliza cada una de las tres especificaciones OCI y qué queda explícitamente fuera?
2. ¿Por qué desplegar por etiqueta impide reconstruir qué se ejecutaba durante un incidente?
3. Demuestra en tres órdenes que un fichero borrado en el Dockerfile sigue dentro de la imagen.
4. ¿Por qué la misma etiqueta puede entregar binarios distintos a dos máquinas, y cómo se verifica?
5. ¿Qué campos de la configuración tienen efecto en la ejecución y cuáles son solo documentación?

Referencias

- Open Container Initiative (2025). Image Format Specification — manifiesto, configuración, capas e índice. <https://github.com/opencontainers/image-spec/blob/main/spec.md>
- Open Container Initiative (2025). Distribution Specification — API del registro, huellas y autenticación. <https://github.com/opencontainers/distribution-spec/blob/main/spec.md>
- Open Container Initiative (2025). Runtime Specification — paquete de sistema de ficheros y configuración de ejecución. <https://github.com/opencontainers/runtime-spec/blob/main/spec.md>
- Docker (2025). Multi-platform builds — buildx, índices de imágenes y construcción nativa frente a emulada. <https://docs.docker.com/build/building/multi-platform/>
- Docker (2025). Image layers and storage drivers — sistema de ficheros de unión y ficheros de borrado. <https://docs.docker.com/engine/storage/drivers/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 05 en PDF · Manual integral

Anterior	Índice	Siguiente
← 060 · Proyecto: aplicación de tres capas en Google Cloud	Parte 05 · Programa	062 · Dockerfile reproducible y builds multi-stage →

Evaluación — 061 Imágenes, capas, registros y estándar OCI

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 inspeccion-imagen 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.

062 — Dockerfile reproducible y builds multi-stage

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 4 Laboratorio: container · Estado: EXECUTABLECORE

Propósito

Escribir un Dockerfile que produzca imágenes reproducibles, pequeñas y con poca superficie, y hacerlo entendiendo que las dos mitades del problema son distintas: la caché decide cuánto tarda y las etapas deciden qué acaba dentro. La clase 061 dejó tres deudas concretas —un token dentro de una capa, un índice de paquetes de hace meses y la imposibilidad de saber qué se ejecutaba—; las tres se pagan aquí, y con ellas se establece el principio que gobierna las partes 08 y siguientes: se construye una vez y se promueve, nunca se reconstruye por entorno.

Resultados de aprendizaje

Al finalizar podrás:

1. Ordenar las instrucciones para que la caché invalide lo mínimo y medir la diferencia.
2. Separar la etapa de construcción de la de ejecución y justificarlo por superficie, no por tamaño.
3. Usar montajes de caché y de secreto para no dejar rastro ni repetir descargas.
4. Fijar base, dependencias y contexto para que dos construcciones del mismo commit se comporten igual.
5. Aplicar el principio de construir una vez y promover por huella entre entornos.

Conceptos centrales

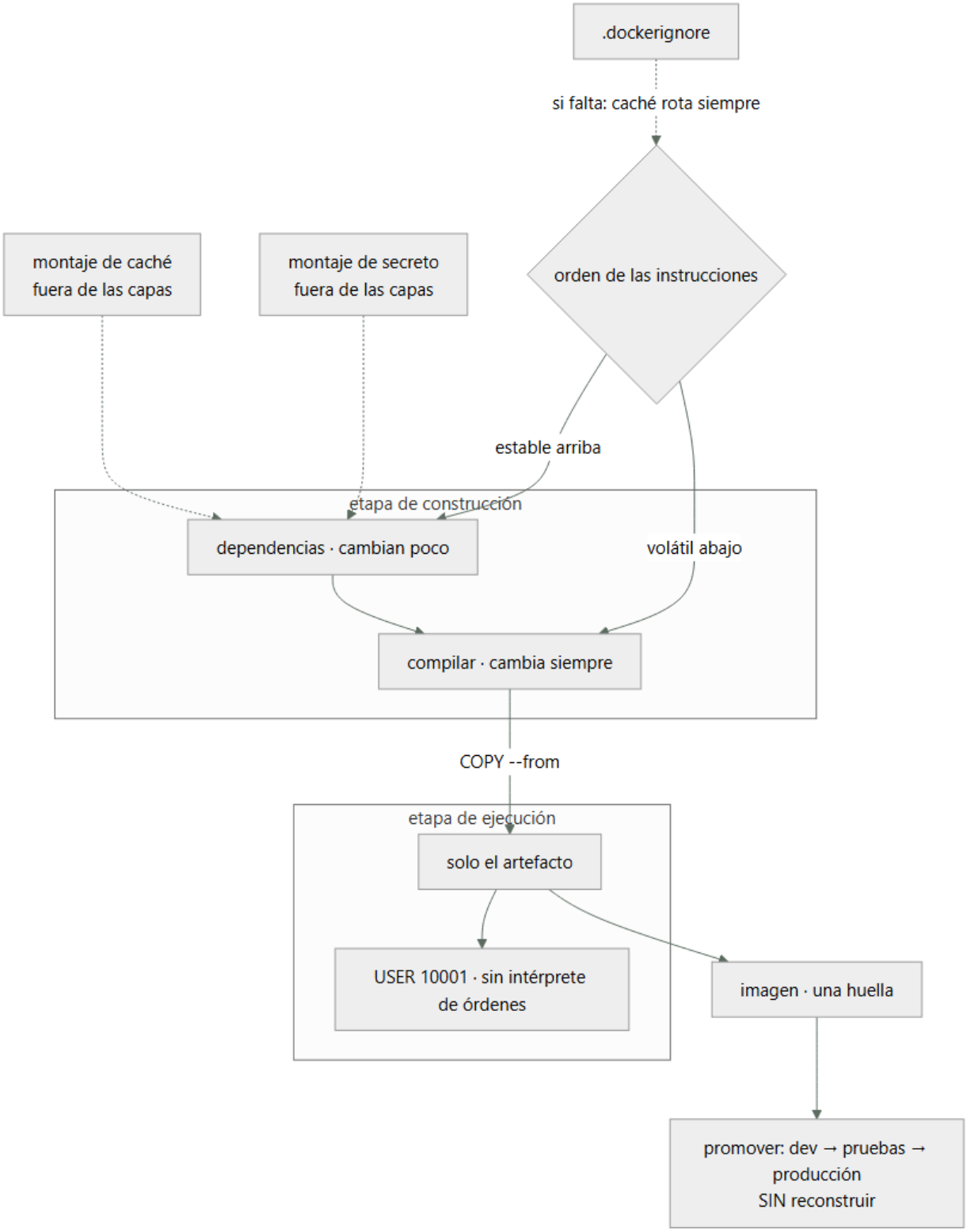
Concepto	Comprensión verificable
invalidación de caché	Una instrucción reutiliza su capa si su entrada no cambió y ninguna anterior cambió. Por eso el orden decide el tiempo de construcción mucho más que el contenido.
construcción por etapas	Varias imágenes intermedias de las que solo se copia el resultado. La etapa final no contiene compilador, gestor de paquetes ni código fuente.
montaje de caché	Directorio persistente entre construcciones que no forma parte de ninguna capa. Acelera sin engordar la imagen ni filtrar nada.
montaje de secreto	Fichero disponible solo durante una instrucción y ausente de la capa resultante. Es la corrección del token filtrado de la clase 061.
contexto de construcción	Lo que se envía al motor. Sin .dockerignore incluye el repositorio entero, así que engorda la imagen y invalida la caché en cada cambio.
construir una vez y promover	El mismo artefacto, identificado por huella, recorre todos los entornos. Lo que cambia entre ellos es la configuración inyectada al ejecutar, nunca la imagen.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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

■ Flujo de razonamiento



Desarrollo

1. El orden decide el tiempo

La regla de la caché tiene una sola frase y una consecuencia grande: una instrucción reutiliza su capa si su entrada no cambió y ninguna anterior cambió. Un cambio en una línea invalida todo lo que viene después.

El error más extendido cabe en dos líneas:

```
COPY . .            # cualquier cambio en el repositorio invalida aquí
RUN npm ci          # → se reinstalan TODAS las dependencias en cada construcción
```

Y la corrección, que separa lo que cambia poco de lo que cambia siempre:

```
COPY package.json package-lock.json ./
RUN npm ci                        # solo se reejecuta si cambian las dependencias
COPY . .                          # el código, que cambia en cada commit
RUN npm run build
```

La diferencia es medible y grande, porque las dependencias son casi siempre la parte lenta:

```
construcción tras cambiar una línea de código
  con COPY . . primero      6 min 40 s
  con dependencias aparte      48 s
```

El mismo principio vale para cualquier lenguaje: primero el manifiesto de dependencias y su instalación, después el código. En compilados con un gestor propio, el patrón se repite con el fichero de bloqueo.

Y el contexto de construcción es la otra mitad del problema, porque decide qué entra en COPY . .. Sin .dockerignore, entra el repositorio completo:

```
.git/               historial entero, a veces con secretos ya rotados
node_modules/       megabytes que se van a reinstalar igualmente
.env                configuración local, a menudo con credenciales
cobertura, registros, artefactos de compilación previos
```

Y hay un efecto secundario que sorprende: cualquier cambio en cualquiera de esos ficheros invalida la caché, incluido un fichero de registro que la ejecución anterior acaba de escribir. Un .dockerignore decente no es higiene, es rendimiento:

```
.git
node_modules
.env*
**/*.log
coverage
dist
Dockerfile
.dockerignore
```

Dos trampas más de orden, con consecuencias distintas:

Actualizar el índice de paquetes en una instrucción aparte.

```
RUN apt-get update                          # ← esta capa se cachea
RUN apt-get install -y curl ca-certificates # y usa un índice de hace meses
```

La primera capa se reutiliza indefinidamente porque su texto no cambia, así que la instalación resuelve contra un índice antiguo y puede traer versiones que ya se corrigieron. Van siempre en la misma instrucción, y limpiando en ella:

```
RUN apt-get update \
 && apt-get install -y --no-install-recommends curl ca-certificates \
 && rm -rf /var/lib/apt/lists/*
```

La limpieza en la misma instrucción sí reduce el tamaño; en una posterior no, por lo que la clase 061 demostró sobre los ficheros de borrado.

ADD en vez de COPY. ADD extrae automáticamente los archivos comprimidos y descarga direcciones remotas, dos comportamientos implícitos que rara vez se quieren y que dificultan razonar sobre el resultado. COPY para todo, salvo que se necesite justo lo que ADD hace de más.

2. Las etapas deciden la superficie

La construcción por etapas se explica casi siempre por el tamaño, y el tamaño es el argumento menos importante. El argumento real es qué queda dentro para que alguien lo use.

```
# ---- construcción ----
FROM golang:1.23@sha256:1a2b3c... AS construccion
WORKDIR /src
COPY go.mod go.sum ./
RUN --mount=type=cache,target=/go/pkg/mod go mod download
COPY . .
RUN --mount=type=cache,target=/root/.cache/go-build \
    CGO_ENABLED=0 go build -trimpath -ldflags="-s -w" -o /bin/app ./cmd/app

# ---- ejecución ----
FROM gcr.io/distroless/static-debian12:nonroot@sha256:4d5e6f...
```

```
COPY --from=construccion /bin/app /bin/app
USER 65532:65532
ENTRYPOINT ["/bin/app"]
```

Lo que la etapa final no tiene, y lo que eso significa:

```
sin compilador ni cadena de herramientas   no se puede compilar nada dentro
sin gestor de paquetes                     no se puede instalar nada dentro
sin intérprete de órdenes                  una ejecución de órdenes remota
                                           no tiene con qué ejecutarlas
sin código fuente ni ficheros de proyecto  no hay nada que leer
```

La tercera línea es la que más cambia el resultado de una intrusión: buena parte de las cadenas de explotación conocidas dependen de disponer de un intérprete de órdenes o de una utilidad de descarga dentro del contenedor. Sin ellos, la misma vulnerabilidad rinde mucho menos.

Y tiene una contrapartida honesta que hay que aceptar por adelantado: no se puede depurar entrando. docker exec no tiene nada que ejecutar. La respuesta correcta es la de la clase 057 —si hace falta entrar a menudo, falta instrumentación— y la técnica concreta es un contenedor efímero de depuración que comparte los espacios de nombres del que se investiga, que la clase 070 desarrolla.

Los tamaños resultantes ordenan la elección de base final:

```
imagen sin etapas, base completa      1,2 GB
base minimalista con paquetes         180 MB
base minimalista sin intérprete        96 MB
binario estático sobre base vacía      12 MB
```

Y la advertencia de la clase 061 sigue vigente: bajar de base no es gratis. Una base sin bibliotecas estándar exige un binario realmente estático, y una base con otra implementación de las bibliotecas del sistema cambia comportamientos de red y de fecha. Se elige la base más pequeña que pase las pruebas de integración, no la más pequeña.

Un detalle de etapas que ahorra tiempo en canalizaciones grandes: las etapas que no se necesitan no se construyen. --target permite construir solo hasta una, lo que hace posible tener en el mismo fichero una etapa de pruebas, una de análisis y una de ejecución sin pagar todas en cada construcción.

```
$ docker build --target pruebas -t tienda:test .
$ docker build -t tienda:v8 .
```

3. Montajes: acelerar sin engordar, usar secretos sin filtrarlos

Los montajes de construcción resuelven dos problemas que antes obligaban a elegir entre velocidad y limpieza.

Montaje de caché. Un directorio que persiste entre construcciones y no forma parte de ninguna capa:

```
RUN --mount=type=cache,target=/root/.npm \
    npm ci --prefer-offline

RUN --mount=type=cache,target=/var/cache/apt,sharing=locked \
    --mount=type=cache,target=/var/lib/apt,sharing=locked \
    apt-get update && apt-get install -y --no-install-recommends curl
```

La diferencia con dejar la caché dentro de una capa es doble: no engorda la imagen y sobrevive a la invalidación. Cuando cambia el fichero de dependencias, la instrucción se reejecuta pero los paquetes ya descargados siguen ahí, así que la reinstalación completa tarda segundos en vez de minutos.

Montaje de secreto. Es la corrección directa del incidente de la clase 061:

```
RUN --mount=type=secret,id=npmrc,target=/root/.npmrc \
    npm ci
```

```
$ docker build --secret id=npmrc,src=$HOME/.npmrc -t tienda:v8 .
```

El fichero existe durante esa instrucción y no aparece en la capa resultante. La comprobación es la misma de la clase 061 y debe formar parte de la canalización:

```
$ docker save tienda:v8 -o img.tar && mkdir -p x && tar -xf img.tar -C x
$ for c in x/blobs/sha256/*; do tar -tzf "$c" 2>/dev/null; done | grep -c npmrc
0                                                                          ✓
```

Y para clonar repositorios privados durante la construcción, el montaje de agente evita meter una clave:

```
RUN --mount=type=ssh git clone git@interno:cls/biblioteca.git
```

Lo que no hay que usar para secretos, y aparece en muchas guías antiguas:

```
ARG TOKEN                queda en el historial de la imagen (clase 061)
ENV TOKEN                queda en la configuración, legible con inspect
COPY secreto / + RUN rm  queda en la capa donde entró
```

Los tres dejan rastro recuperable. El montaje de secreto es la única forma que no lo deja.

Y en la canalización, la caché compartida en un registro es lo que hace que la primera construcción de un agente efímero no empiece de cero:

```
$ docker buildx build \
    --cache-from type=registry,ref=registro/tienda:cache \
    --cache-to   type=registry,ref=registro/tienda:cache,mode=max \
    -t registro/tienda:v8 --push .
```

Es la diferencia entre una canalización que tarda once minutos y una que tarda noventa segundos, y cuesta un repositorio adicional en el registro.

4. Reproducible: lo que se puede prometer y lo que no

«Reproducible» se usa con dos significados distintos y conviene separarlos, porque uno es alcanzable y el otro es un proyecto en sí mismo.

```
mismo commit → mismo COMPORTAMIENTO      alcanzable, y es lo que importa
mismo commit → mismos BYTES exactos      difícil: marcas de tiempo, orden de
                                         ficheros, rutas absolutas, aleatoriedad
```

Perseguir lo segundo sin necesitarlo consume tiempo. Conseguir lo primero es cuestión de fijar cuatro cosas:

La base, por huella. Una etiqueta de base se mueve, con lo que dos construcciones del mismo commit pueden partir de sistemas distintos:

```
FROM node:22-bookworm-slim@sha256:0c1f2e3d...
```

Y hay que aceptar la consecuencia: la base fijada no recibe parches automáticamente. La actualización pasa a ser un cambio explícito en el repositorio, revisable y trazable — que es exactamente lo que se quiere, siempre que exista un proceso que la proponga con regularidad. Sin ese proceso, fijar por huella congela vulnerabilidades.

Las dependencias, por fichero de bloqueo. npm ci en vez de npm install, y el equivalente en cada ecosistema. La orden que instala «lo que diga el manifiesto» resuelve versiones nuevas en cada construcción.

Los paquetes del sistema, por versión, cuando importa:

```
RUN apt-get update && apt-get install -y --no-install-recommends \
      curl=7.88.1-10+deb12u* ca-certificates
```

Con el matiz de que los repositorios de distribución retiran versiones antiguas, así que fijar de más produce construcciones que fallan solas con el tiempo. La práctica sostenible es fijar la base por huella —que ya congela el conjunto de paquetes— y no fijar cada paquete.

El contexto, con .dockerignore, para que un fichero local no cambie el resultado.

Y para acercarse a la reproducibilidad de bytes cuando de verdad hace falta —cadenas de suministro auditadas, que es materia de la clase 067—, existen mecanismos que normalizan las marcas de tiempo:

```
$ SOURCE_DATE_EPOCH=$(git log -1 --pretty=%ct) \
  docker buildx build --output type=registry,rewrite-timestamp=true .
```

Y el principio que gobierna todo lo demás, que es el que se lleva a las partes 08 y 12:

```
se construye UNA vez y se promueve
  dev → pruebas → preproducción → producción
  la misma HUELLA en los cuatro
  lo que cambia es la configuración inyectada al ejecutar
```

Reconstruir por entorno destruye la garantía entera: lo que se probó en preproducción no es lo que se ejecuta en producción, aunque el commit sea el mismo. Y la señal de que un equipo lo está haciendo mal es fácil de buscar: si el Dockerfile menciona el entorno, algo va mal.

```
ARG ENTORNO=produccion         # ← señal de alarma
RUN cp config.$ENTORNO.json config.json
```

La configuración por entorno se inyecta al ejecutar, como variables o como ficheros montados, que es el modelo que las clases 043, 055 y 066 usan.

5. El resultado, revisado como se revisa el código

Un Dockerfile bien escrito se reconoce por lo que no tiene, y merece una lista de revisión corta que se pueda aplicar en un pull request:

- base fijada por huella, con un proceso que proponga actualizarla
- dependencias antes que código, para que la caché sirva
- .dockerignore presente y con .git, secretos y artefactos
- actualización e instalación de paquetes en la MISMA instrucción, con limpieza
- etapa de ejecución sin compilador, sin gestor de paquetes y sin intérprete
- ningún secreto por ARG, ENV ni COPY: solo montaje de secreto
- USER numérico y distinto de 0
- ENTRYPOINT en forma de lista, no de cadena
- etiquetas estándar de origen y revisión (clase 061)
- nada específico de entorno

La penúltima merece una explicación porque su efecto es invisible hasta que hace falta. En forma de cadena, el motor arranca un intérprete de órdenes que ejecuta el programa como hijo:

```
ENTRYPOINT /bin/app --puerto=8080     # forma de cadena: sh -c "…"
ENTRYPOINT ["/bin/app", "--puerto=8080"]  # forma de lista: el proceso ES el 1
```

Con la primera, el proceso número uno es el intérprete, las señales le llegan a él y no las reenvía. Una petición de parada se ignora y la plataforma acaba matando el contenedor pasado el plazo, cortando peticiones en curso. Es la causa más frecuente del apagado no ordenado, y la clase 068 lo desarrolla; aquí basta con no crearlo.

Y dos comprobaciones automáticas que conviene tener en la canalización, porque detectan casi todo lo de la lista:

```
$ hadolint Dockerfile                      # reglas del propio formato
$ docker scout cves registro/tienda:v8     # o trivy / grype (clase 067)
```

Más las dos pruebas negativas propias de esta clase, que son las que demuestran lo que no se puede ver:

```
# ningún secreto en las capas
$ docker save registro/tienda:v8 | tar -xO | strings | grep -Ec 'authToken|BEGIN PRIVATE KEY'
0                                                                          ✓

# la etapa final no tiene intérprete de órdenes
$ docker run --rm --entrypoint sh registro/tienda:v8 -c 'echo hola'
docker: Error response from daemon: … "sh": executable file not found      ✓
```

La segunda es una prueba negativa en el sentido exacto de las clases 046 y 058: el éxito es que falle. Y como toda prueba negativa del programa, su valor está en ejecutarse en cada construcción y no en haberse ejecutado una vez.

Ejemplo trabajado

CloudShop reescribe sus Dockerfile. La construcción tarda once minutos, la imagen pesa 1,2 GB y las tres deudas de la clase 061 siguen abiertas. Cinco cambios las cierran y dejan un principio que gobierna el resto del programa.

Punto de partida:

```
FROM node:22
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build
COPY .npmrc /root/.npmrc
RUN npm ci --production && rm /root/.npmrc
EXPOSE 8080
CMD npm start
```

Cambio 1 — el orden, y once minutos que pasan a noventa segundos.

```
construcción tras cambiar una línea         6 min 40 s       48 s
construcción en agente efímero de CI       11 min 20 s    1 min 30 s
contexto enviado al motor                     412 MB          9 MB
```

Tres medidas a la vez: dependencias antes que código, .dockerignore con .git y nodemodules, y caché compartida en el registro para los agentes efímeros. El contexto de 412 MB era casi todo .git y nodemodules, y además invalidaba la caché en cada construcción porque cualquier fichero local cambiaba.

Cambio 2 — etapas, y lo que deja de estar dentro.

```
FROM node:22-bookworm-slim@sha256:0c1f2e3d... AS deps
WORKDIR /app
COPY package.json package-lock.json ./
RUN --mount=type=cache,target=/root/.npm \
    --mount=type=secret,id=npmrc,target=/root/.npmrc \
    npm ci

FROM deps AS construccion
COPY . .
RUN npm run build

FROM gcr.io/distroless/nodejs22-debian12:nonroot@sha256:7a8b9c...
WORKDIR /app
COPY --from=construccion /app/dist ./dist
COPY --from=deps /app/node_modules ./node_modules
USER 65532:65532
ENTRYPOINT ["/nodejs/bin/node", "dist/main.js"]
```

```
tamaño de la imagen                          1,2 GB         148 MB
intérprete de órdenes en la imagen final       sí             no
gestor de paquetes en la imagen final          sí             no
código fuente en la imagen final               sí             no
paquetes del sistema con vulnerabilidades
  conocidas (alta o crítica)                    41              3
```

Los tres restantes son de la propia base y se siguen con el proceso de actualización, no con un parche local.

Cambio 3 — el token, cerrado de verdad.

El orden fue el mismo que en la clase 061 y no es negociable: rotar primero, corregir después.

```
$ docker save registro/tienda:v9 | tar -xO | strings | grep -Ec 'authToken'
0                                                                    ✓
```

```
método                              COPY + rm en capa    montaje de secreto
token extraíble de las capas                  sí               no
comprobación en la canalización             ninguna     bloquea la publicación
```

Cambio 4 — el índice de paquetes de hace cuatro meses.

```
$ docker run --rm --entrypoint cat registro/tienda:v8 /var/lib/apt/lists/lock 2>/dev/null
$ docker history registro/tienda:v8 | grep 'apt-get update'
<hace 118 días>   RUN apt-get update
```

La capa de actualización llevaba cuatro meses reutilizándose, así que la instalación resolvía contra un índice de entonces. Dos paquetes con correcciones publicadas seguían entrando en cada construcción nueva.

```
actualización e instalación           dos instrucciones   una, con limpieza
antigüedad del índice de paquetes         118 días          la de la base
paquetes desactualizados en la imagen         2                 0
```

Cambio 5 — se construía una vez por entorno.

La canalización ejecutaba una construcción para preproducción y otra para producción, con un argumento distinto:

```
ARG ENTORNO   →   cp config.$ENTORNO.json config.json
```

Eso producía dos imágenes distintas del mismo commit, y explicaba un incidente anterior: un fallo que solo se daba en producción y no se reproducía en preproducción, porque no eran el mismo artefacto.

```
imágenes por commit                              2                1
configuración                          dentro de la imagen  inyectada al ejecutar
lo probado y lo desplegado             commit igual,      MISMA HUELLA
                                       artefacto distinto
tiempo total de la canalización            22 min           3 min 10 s
```

La última fila es consecuencia de todo lo anterior: una construcción en vez de dos, con caché compartida y contexto pequeño.

Resumen del empaquetado:

```
construcción en agente efímero              11 min 20 s    1 min 30 s
tamaño de la imagen                            1,2 GB        148 MB
contexto enviado                               412 MB          9 MB
secretos extraíbles de las capas                  1            0
```

```
intérprete de órdenes en la imagen final         sí            no
vulnerabilidades de alta o crítica               41             3
imágenes distintas por commit                     2             1
usuario del proceso                            root         65532
```

La lección que esta clase traslada al resto de la parte 05: la construcción por etapas se vende por el tamaño y vale por la superficie —una imagen sin intérprete de órdenes cambia lo que una intrusión puede hacer—, y la caché se arregla ordenando, no añadiendo herramientas. Pero el cambio con más consecuencias fue el quinto, que no toca el Dockerfile: construir una vez y promover por huella es lo que hace que «lo probado» y «lo desplegado» sean la misma frase. Sin eso, todas las pruebas del programa hablan de un artefacto que no es el que corre.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/062-dockerfile-reproducible-y-builds-multi-stage/lab.py
```

El laboratorio selecciona el motor de práctica container 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 imagen-minima 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 imagen mínima, escaneada y ejecutada sin privilegios. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye imagen-minima 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
Cada construcción reinstala todas las dependencias	El código se copia antes de instalarlas, así que cualquier cambio invalida esa capa	Copia el manifiesto de dependencias e instálalas antes de copiar el código.
La caché nunca acierta aunque el código no cambie	Falta .dockerignore y ficheros locales —registros, artefactos, .git— entran en el contexto	Escribe .dockerignore con .git, dependencias instaladas, secretos y artefactos, y comprueba el tamaño del contexto.
La imagen instala versiones antiguas de paquetes ya corregidos	La actualización del índice está en una instrucción propia y su capa se reutiliza indefinidamente	Actualiza e instala en la misma instrucción, con limpieza en esa misma instrucción.
Un secreto sigue siendo extraíble pese a no aparecer en el sistema de ficheros final	Se pasó por ARG, por ENV o por COPY, y queda en el historial o en la capa	Usa montaje de secreto, rota lo ya expuesto y añade la comprobación de capas a la canalización.
Un fallo solo se reproduce en producción con el mismo commit	Se construye una imagen por entorno, así que los artefactos son distintos	Construye una vez y promueve por huella; la configuración se inyecta al ejecutar.
El contenedor ignora la petición de parada y lo acaban matando	ENTRYPOINT en forma de cadena arranca un intérprete que no reenvía las señales	Usa la forma de lista para que el proceso de la aplicación sea el número uno.

Seguridad, ética y costo

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

Preguntas de comprobación

1. ¿Por qué copiar el código antes de instalar dependencias multiplica el tiempo de construcción?
2. ¿Cuál es el argumento principal de la construcción por etapas, y por qué el tamaño es el secundario?
3. ¿Qué tres formas de pasar un secreto dejan rastro recuperable y cuál no?
4. ¿Qué se puede prometer sobre reproducibilidad fijando la base por huella, y qué obligación crea?
5. ¿Qué señal en un Dockerfile indica que el equipo está reconstruyendo por entorno, y por qué importa?

Referencias

- Docker (2025). Dockerfile best practices — orden de instrucciones, caché y contexto. <https://docs.docker.com/build/building/best-practices/>
- Docker (2025). Multi-stage builds — etapas, --target y copia entre etapas. <https://docs.docker.com/build/building/multi-stage/>
- Docker (2025). Build secrets and cache mounts — --mount=type=secret y type=cache. <https://docs.docker.com/build/building/secrets/>
- Docker (2025). Cache storage backends — caché compartida en registro para canalizaciones. <https://docs.docker.com/build/cache/backends/>
- Google (2025). Distroless container images — bases sin intérprete de órdenes ni gestor de paquetes. <https://github.com/GoogleContainerTools/distroless>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 05 en PDF · Manual integral

Evaluación — 062 Dockerfile reproducible y builds multi-stage

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 imagen-minima 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.

063 — Namespaces, cgroups y runtime de contenedores

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 4 Laboratorio: container · Estado: EXECUTABLECORE

Propósito

Entender qué es un contenedor de verdad, porque no es una cosa: es un proceso normal con tres mecanismos del núcleo aplicados encima. De ahí salen los comportamientos que desconciertan en producción —un proceso que ignora la señal de parada, un límite de CPU que no aparece como CPU alta, un contenedor que sigue sano con su trabajador muerto— y de ahí sale también la respuesta a la pregunta que abrió la parte: qué queda fuera del aislamiento, que es exactamente donde la portabilidad del contenedor deja de valer.

Resultados de aprendizaje

Al finalizar podrás:

1. Construir un aislamiento equivalente a un contenedor con órdenes sueltas, para demostrar que no hay ningún objeto «contenedor».
2. Explicar por qué el proceso número uno ignora las señales que no ha declarado, y qué consecuencia tiene.
3. Diagnosticar una limitación de CPU distinguiéndola de una CPU saturada.
4. Detectar un proceso terminado por falta de memoria, incluido el caso en que el contenedor sobrevive.
5. Enumerar qué no está aislado y qué alternativas existen cuando el núcleo compartido no es aceptable.

Conceptos centrales

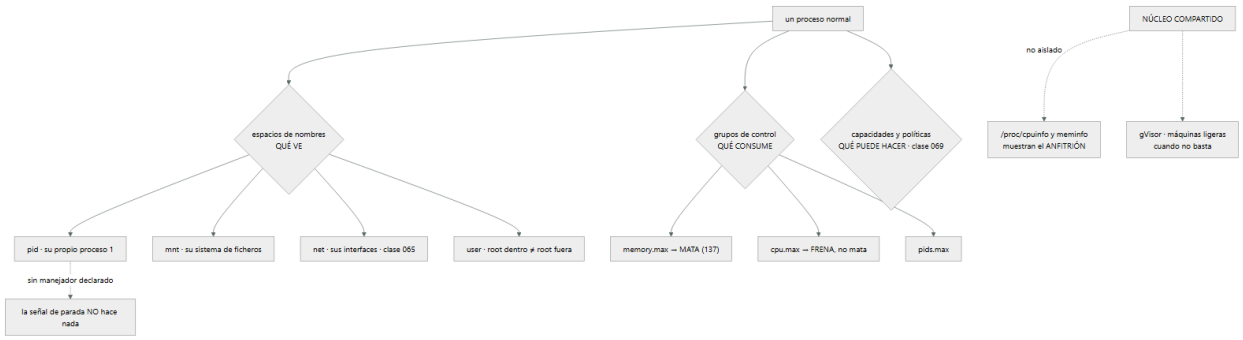
Concepto	Comprensión verificable
espacio de nombres	Mecanismo que cambia lo que un proceso ve: sus procesos, su red, sus montajes, su nombre de máquina. No limita nada; solo delimita la vista.
grupo de control	Mecanismo que limita y contabiliza lo que un proceso puede consumir: memoria, CPU, entrada/salida y número de procesos.
proceso número uno	El primero del espacio de nombres de procesos. El núcleo no le instala los manejadores por defecto: una señal que el programa no declare simplemente no hace nada.
limitación de CPU	El límite de CPU no corta: frena. Se manifiesta como latencia con uso aparentemente bajo, y su señal es el contador de periodos limitados.
terminación por memoria	El límite de memoria no ralentiza: mata. El núcleo elige un proceso del grupo, que puede no ser el número uno, y el contenedor sigue en pie sin hacer nada.
núcleo compartido	Todos los contenedores de una máquina usan el mismo núcleo. Es lo que los hace ligeros y lo que fija el límite de su aislamiento.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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. No hay ningún objeto «contenedor»

El núcleo no tiene una estructura llamada contenedor. Hay procesos, y sobre un proceso se pueden aplicar tres cosas independientes:

```
espacios de nombres   cambian lo que VE
grupos de control     limitan lo que CONSUME
capacidades y políticas de seguridad   acotan lo que PUEDE HACER  (clase 069)
```

Un contenedor es la aplicación conjunta de las tres. Y se puede demostrar sin ningún motor, lo que conviene hacer una vez porque cambia el modelo mental para siempre:

```
# un "contenedor" a mano: vista de procesos, montajes y nombre propios
$ sudo unshare --pid --mount --uts --ipc --net --fork --mount-proc \
    /bin/bash

# dentro:
# ps aux
USER  PID  COMMAND
root    1  /bin/bash        ← este intérprete es el proceso 1
root    8  ps aux

# hostname aislado
# ip link
1: lo: <LOOPBACK> mtu 65536 …     ← solo el bucle local: no hay red
```

Y el límite se aplica escribiendo en el sistema de ficheros de grupos de control, que también es texto plano:

```
$ sudo mkdir /sys/fs/cgroup/demo
$ echo 268435456 | sudo tee /sys/fs/cgroup/demo/memory.max   # 256 MiB
$ echo "20000 100000" | sudo tee /sys/fs/cgroup/demo/cpu.max # 0,2 CPU
$ echo $PID | sudo tee /sys/fs/cgroup/demo/cgroup.procs
```

Eso es todo lo que hace un motor de contenedores, más la preparación del sistema de ficheros a partir de las capas de la clase 061 y la aplicación de las políticas de la clase 069.

La cadena real, de arriba abajo, aclara un comportamiento operativo que confunde:

```
plataforma (orquestador o motor)
  → containerd            gestiona imágenes y ciclo de vida
    → shim                proceso intermedio, uno por contenedor
      → runc              aplica espacios, grupos y políticas
        → tu proceso      y runc SALE
```

Como runc termina en cuanto ha arrancado el proceso, y quien queda como padre es el shim, reiniciar el motor no mata los contenedores. Es lo que permite actualizar containerd sin cortar el servicio, y explica por qué al listar procesos en el anfitrión no aparece ningún runc.

Y los espacios de nombres que importan, con la clase donde se cobra cada uno:

```
pid     su propio proceso 1              → señales y apagado ordenado (068)
mnt     su vista del sistema de ficheros → volúmenes y persistencia   (064)
net     sus interfaces y sus rutas        → red y publicación         (065)
user    su propia correspondencia de uid  → sin privilegios           (069)
uts     su nombre de máquina
ipc     su memoria compartida
cgroup  su vista del árbol de grupos
```

2. El proceso número uno no es un proceso normal

Esta es la particularidad del núcleo que más incidentes produce y la que menos gente conoce.

Para el proceso número uno de un espacio de nombres, el núcleo no instala los manejadores de señal por defecto. En un proceso normal, recibir una petición de terminación lo mata aunque el programa no haga nada. En el proceso número uno:

```
señal de terminación (TERM) sin manejador declarado  →  NO OCURRE NADA
señal de matar (KILL)                                →  siempre mata
```

La consecuencia es directa: una plataforma que pide parar de forma ordenada envía la primera señal, espera el plazo de gracia —treinta segundos por defecto en la mayoría— y después envía la segunda. Si el programa no declaró el manejador, el plazo entero se desperdicia y el proceso muere de golpe, cortando peticiones en curso.

```
$ docker run --rm -d --name demo imagen:v1
$ time docker stop demo
real    0m10.4s        ← el plazo completo: nadie escuchó
```

Hay dos causas y conviene distinguirlas:

```
1. el programa no declara el manejador
   → se corrige en el código, y es lo correcto
2. el proceso 1 es un intérprete de órdenes que no reenvía
   → ENTRYPOINT en forma de cadena (clase 062)
   → el intérprete recibe la señal y su hijo no se entera
```

La segunda es la más común y la más fácil de arreglar: forma de lista en el punto de entrada, para que el proceso de la aplicación sea el uno.

Y hay un tercer efecto del proceso uno que produce fugas lentas: es el responsable de recoger a los procesos huérfanos. Si la aplicación lanza procesos hijos y no los espera, sus entradas se acumulan como procesos zombis y consumen entradas de la tabla del núcleo hasta agotar pids.max.

```
$ docker exec demo ps -eo pid,stat,comm | grep -c ' Z'
1847
```

La corrección es un proceso de inicio mínimo que recoja hijos y reenvíe señales, que la mayoría de motores ofrecen como interruptor:

```
$ docker run --init imagen:v1
```

Y cuándo hace falta, dicho con precisión, porque añadirlo a ciegas es innecesario:

```
hace falta   si la aplicación lanza procesos hijos que no espera
             o si el punto de entrada es un guion con varios procesos
no hace falta si el proceso 1 es un binario único que declara sus manejadores
```

La clase 068 desarrolla el apagado ordenado completo; lo que hay que llevarse aquí es por qué el mecanismo se comporta así: no es una peculiaridad del motor, es el contrato del proceso número uno en el núcleo.

3. La memoria mata y la CPU frena

Los dos límites más usados se comportan de forma opuesta, y confundirlos lleva a diagnosticar al revés.

El límite de memoria mata. No hay ralentización previa ni aviso: cuando el grupo supera su límite, el núcleo elige un proceso del grupo y lo termina.

```
$ docker inspect demo --format '{{.State.ExitCode}} {{.State.OOMKilled}}'
137 true
```

El código 137 es la firma. Y hay un caso que produce el fallo silencioso más difícil de ver de esta clase: el núcleo no siempre mata al proceso número uno. Si la aplicación tiene un proceso principal ligero y trabajadores pesados, el elegido puede ser un trabajador:

```
el trabajador muere
el proceso 1 sigue vivo
el contenedor sigue "en ejecución"
la comprobación de estado responde 200
y el trabajo deja de hacerse
```

Es exactamente la familia de fallos que la clase 060 identificó como la más cara: un mecanismo que parece estar haciendo algo y no lo está. Se detecta contando trabajadores vivos, no comprobando que el proceso responda:

```
$ cat /sys/fs/cgroup/memory.events
low 0
high 0
max 4127
oom 3
oom_kill 3
```

Ese contador debería ser una métrica vigilada. Un oomkill mayor que cero es siempre un incidente, incluso si nadie ha notado nada.

El límite de CPU frena. No mata, no falla, y no aparece como CPU alta. El grupo recibe una cuota por periodo y, al agotarla, sus hilos se detienen hasta el siguiente:

```
cpu.max = "20000 100000"   → 20 ms de cada 100: 0,2 CPU
```

El síntoma es latencia con uso aparentemente bajo, y la señal es otra:

```
$ cat /sys/fs/cgroup/cpu.stat
usage_usec 41283991
nr_periods 129483
nr_throttled 41207          ← el 32 % de los periodos, frenado
throttled_usec 8814299
```

Un 32 % de periodos frenados explica un percentil alto que ninguna gráfica de CPU muestra. nrthrottled es la métrica que falta en casi todos los paneles.

Y la causa más frecuente de limitación no es que falte CPU, sino un tercer hecho que une los dos apartados anteriores: lo que no está aislado.

4. Lo que el contenedor ve del anfitrión, y por qué se dimensiona mal solo

Los espacios de nombres delimitan procesos, red y montajes. No delimitan lo que el núcleo publica sobre el hardware. Dentro de un contenedor con 2 CPU y 512 MiB de límite:

```
$ docker run --rm --cpus 2 --memory 512m alpine sh -c 'nproc; free -m | head -2'
64                                  ← los del ANFITRIÓN
              total   used   free
Mem:         257843  91223  16620    ← la del ANFITRIÓN
```

Y de ahí sale el problema que produce la mitad de las limitaciones de CPU y de las terminaciones por memoria del mundo real: los tiempos de ejecución se dimensionan solos leyendo esa información.

```
un tiempo de ejecución que crea un hilo por CPU     → 64 hilos con 2 CPU de cuota
                                                      → limitación constante

una máquina virtual que calcula su montón como una
fracción de la memoria "disponible"                  → 64 GiB de montón
                                                       con 512 MiB de límite
                                                       → terminación por memoria
un compilador que lanza trabajos según los núcleos   → 96 trabajos con 2 CPU
```

Las correcciones son por tiempo de ejecución y hay que ponerlas explícitamente:

```
Go          GOMAXPROCS acorde a la cuota, o una biblioteca que la lea del grupo
Java        versiones recientes detectan el límite; conviene fijar el porcentaje
            máximo de memoria en vez del tamaño absoluto
Node        el tamaño del espacio de memoria antiguo no se deduce del límite
Python      el número de trabajadores del servidor no se calcula con nproc
compiladores  el paralelismo se fija, no se detecta
```

La comprobación general es una línea y debería estar en la lista de revisión de cualquier imagen:

```
$ docker run --rm --cpus 2 --memory 512m imagen:v9 \
    sh -c 'cat /sys/fs/cgroup/cpu.max /sys/fs/cgroup/memory.max'
200000 100000
536870912
```

Si el proceso no está leyendo esos dos ficheros —directamente o a través de su tiempo de ejecución—, se está dimensionando con datos del anfitrión.

Y la lista completa de lo que no está aislado, que es la respuesta a la pregunta con la que se abrió la parte 05:

```
el NÚCLEO                    uno solo para toda la máquina
  → una vulnerabilidad del núcleo es una fuga completa
la información de hardware   /proc/cpuinfo, /proc/meminfo, /sys
la mayoría de parámetros del núcleo   son del anfitrión
```

```
el reloj                       salvo con espacio de nombres de tiempo
la caché de páginas            compartida: un contenedor puede desalojar
                               lo que otro estaba leyendo
el ancho de banda de disco y de red   salvo que se limiten explícitamente
```

La penúltima explica un fenómeno molesto: un proceso que lee un fichero enorme puede degradar a los vecinos sin superar ningún límite, porque compite por la caché del anfitrión y no por una cuota.

Y cuando el núcleo compartido no es aceptable —ejecución de código de terceros, requisitos regulatorios— existen dos familias de alternativas:

```
núcleo en espacio de usuario   intercepta las llamadas al sistema y las
                               reimplementa: aislamiento mucho mayor,
                               a costa de compatibilidad y de rendimiento
máquinas virtuales ligeras     un núcleo por contenedor, con arranque
                               en decenas de milisegundos
```

Ambas ejecutan las mismas imágenes OCI, que es la confirmación de la hipótesis de la clase 060: el contrato de la imagen se conserva, y lo que cambia es lo que hay debajo.

5. Diagnosticar con los ficheros del grupo de control

Casi todo lo de esta clase se responde leyendo cuatro ficheros, y conviene conocerlos porque son la fuente de verdad detrás de cualquier panel:

```
# dentro del contenedor, con grupos de control v2
$ cat /sys/fs/cgroup/memory.max      # límite; "max" si no hay
$ cat /sys/fs/cgroup/memory.current  # uso actual
$ cat /sys/fs/cgroup/memory.events   # oom_kill: un número mayor que 0 es un incidente
$ cat /sys/fs/cgroup/cpu.stat        # nr_throttled y throttled_usec
$ cat /sys/fs/cgroup/pids.current /sys/fs/cgroup/pids.max
```

Y el orden de diagnóstico, que distingue las tres causas que se confunden entre sí:

```
síntoma: el servicio va lento
  1. ¿nr_throttled sube?          → falta cuota de CPU, no CPU
  2. ¿memory.current cerca del máximo?  → presión: el núcleo desaloja caché
                                          y la E/S se dispara
  3. ¿ninguno de los dos?         → el problema no es el contenedor

síntoma: el proceso desaparece
  1. código de salida 137 y oom_kill > 0   → terminación por memoria
  2. código 137 sin oom_kill               → alguien envió una señal de matar
  3. código 143                            → terminación tras señal de parada

síntoma: no se puede crear un proceso o un hilo
  1. pids.current contra pids.max          → tope de procesos
  2. procesos zombis acumulados            → falta recolección (proceso 1)
```

La segunda línea del primer bloque merece una nota, porque es un caso que se diagnostica mal a menudo: un contenedor cerca de su límite de memoria no se ralentiza por la memoria, se ralentiza porque el núcleo desaloja continuamente la caché de páginas para no superarlo, y eso convierte lecturas que estaban en memoria en lecturas de disco. El síntoma es E/S alta con una aplicación que no ha cambiado.

Y las tres métricas que deberían estar en el panel de cualquier plataforma de contenedores y casi nunca están completas:

```
oom_kill acumulado por contenedor        cualquier valor mayor que 0 alerta
proporción de periodos limitados         por encima del 5 % ya duele
procesos frente al tope                  detecta fugas antes del fallo
```

Con el uso de CPU y de memoria solos, los tres problemas de esta clase son invisibles: la limitación aparece como CPU baja, la terminación por memoria aparece como un reinicio sin causa y la fuga de procesos no aparece hasta que algo falla.

Un cierre que conecta con el resto de la parte: los límites no son una medida de ahorro, son una frontera de radio de impacto. Un contenedor sin límite de memoria puede tumbar al anfitrión entero y con él a todos sus vecinos; con límite, se mata a sí mismo. La decisión no es «cuánto le pongo» sino «qué prefiero que ocurra cuando se pase», y esa pregunta tiene una respuesta clara en cualquier plataforma multiinquilino.

Ejemplo trabajado

CloudShop despliega sus contenedores con límites puestos a ojo. Los cuatro incidentes del primer mes tienen la misma raíz —el contenedor ve el anfitrión y se dimensiona con esa información— y ninguno se manifiesta como lo que es.

Punto de partida:

```
nodos de 64 CPU y 256 GiB
límites por contenedor: 2 CPU y 512 MiB
vigilancia: uso de CPU y de memoria
```

Incidente 1 — el servicio de catálogo va lento con la CPU al 38 %.

```
$ kubectl exec catalogo-7f4 -- cat /sys/fs/cgroup/cpu.stat
nr_periods 129483
nr_throttled 41207
throttled_usec 8814299
```

Un 32 % de los periodos frenados. La causa, dentro:

```
$ kubectl exec catalogo-7f4 -- sh -c 'nproc; echo $GOMAXPROCS'
64
(vacío)
```

El tiempo de ejecución creaba 64 hilos ejecutables con una cuota de 2 CPU, así que el planificador repartía la cuota entre 64 hilos y cambiaba de contexto sin parar.

```
paralelismo del tiempo de ejecución            64              2
periodos limitados                           32 %            0,4 %
p95 del listado de catálogo                 412 ms           96 ms
cuota de CPU                                 2 CPU           2 CPU   ← sin cambios
```

El percentil bajó cuatro veces sin añadir un solo núcleo.

Incidente 2 — reinicios cada pocas horas atribuidos a una fuga de memoria.

```
$ kubectl get pod pedidos-3a1 -o jsonpath='{.status.containerStatuses[0].lastState.terminated.reason}'
OOMKilled
$ kubectl exec pedidos-3a1 -- sh -c 'java -XX:+PrintFlagsFinal -version | grep MaxHeapSize'
   size_t MaxHeapSize = 68719476736    ← 64 GiB
```

Con un límite de 512 MiB, la máquina virtual había calculado un montón de 64 GiB a partir de la memoria del anfitrión. No había ninguna fuga: el proceso pedía memoria hasta cruzar el límite del grupo.

```
tamaño máximo del montón                    64 GiB          320 MiB
                                                          (62,5 % del límite)
terminaciones por memoria al día               6               0
límite de memoria del contenedor             512 MiB         512 MiB   ← sin cambios
```

Incidente 3 — el contenedor sano que había dejado de trabajar.

Durante seis horas no se procesó ningún pedido en segundo plano. El contenedor figuraba en ejecución y su comprobación de estado devolvía 200.

```
$ kubectl exec procesador-9c2 -- cat /sys/fs/cgroup/memory.events | grep oom_kill
oom_kill 3
$ kubectl exec procesador-9c2 -- ps -eo pid,comm
  PID COMMAND
    1 supervisor
```

El núcleo había terminado los tres procesos trabajadores y había dejado vivo al supervisor, que era ligero. El contenedor «existía» y no hacía nada.

```
comprobación de estado             proceso 1 responde   nº de trabajadores vivos
                                                        y edad del último trabajo
alerta sobre oom_kill                   ninguna         > 0 → aviso inmediato
tiempo hasta detectar la parada          6 h              90 s
```

Es la cuarta aparición en el programa de la misma familia de fallos, y merece anotarse junto a las otras tres: el punto de conexión privado que funcionaba por el camino público, la cola de fallidos que no recibía y los registros apagados. Un mecanismo que parece estar haciendo algo y no lo está.

Incidente 4 — la construcción en contenedor tarda más que en el portátil.

```
$ docker run --rm --cpus 2 imagen-build make -j$(nproc)
# → make -j64 con una cuota de 2 CPU
```

```
trabajos paralelos                              64              4
duración de la construcción                14 min 20 s     3 min 40 s
periodos limitados durante la construcción     71 %            2 %
```

El paralelismo excesivo no solo no ayuda: multiplica el cambio de contexto y el uso de memoria, y los dos empeoran el resultado.

Y una revisión de límites que cambió el criterio.

Al repasar la flota apareció que once contenedores no tenían límite de memoria, por miedo a las terminaciones:

```
contenedores sin límite de memoria            11                0
qué ocurre al pasarse              el anfitrión se queda    se mata el contenedor
                                   sin memoria y el núcleo   que se pasó
                                   mata algo al azar
incidentes de nodo por memoria                 2                0
```

El criterio quedó escrito así: un límite no es un ahorro, es una frontera de radio de impacto. La pregunta no es cuánta memoria darle, sino qué se prefiere que ocurra cuando se pase — y que se mate el culpable siempre es mejor que que el núcleo elija por sorpresa entre todos los vecinos.

Resumen:

```
periodos de CPU limitados (catálogo)            32 %          0,4 %
p95 del catálogo                               412 ms          96 ms
terminaciones por memoria al día                  6             0
tiempo hasta detectar un trabajador muerto        6 h           90 s
duración de la construcción en contenedor    14 min 20 s    3 min 40 s
contenedores sin límite de memoria               11             0
métricas de grupo de control en el panel         0 de 3         3 de 3
```

La lección que esta clase traslada al resto de la parte 05: los cuatro incidentes se corrigieron sin añadir capacidad, porque ninguno era un problema de capacidad. Los tres primeros salen del mismo hecho —el contenedor ve el hardware del anfitrión y sus tiempos de ejecución se dimensionan con esa información—, y ese hecho es una consecuencia directa de lo que los espacios de nombres no aíslan. Conocer la frontera del aislamiento no es teoría: es lo que permite leer nrthrottled en vez de pedir más CPU.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/063-namespaces-cgroups-y-runtime-de-contenedores/lab.py
```

El laboratorio selecciona el motor de práctica container 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-aislamiento 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 imagen mínima, escaneada y ejecutada sin privilegios. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye mapa-aislamiento 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 servicio va lento con el uso de CPU bajo	El grupo de control está limitando: la cuota se agota antes de que la CPU parezca alta	Vigila nrthrottled en cpu.stat y ajusta el paralelismo del tiempo de ejecución a la cuota, no a los núcleos del anfitrión.
Reinicios periódicos que parecen una fuga de memoria	El tiempo de ejecución dimensiona su memoria leyendo la del anfitrión, no el límite del grupo	Fija el tamaño máximo como porcentaje del límite del contenedor y comprueba memory.max desde dentro.
El contenedor está en ejecución y su trabajo no avanza	El núcleo terminó por memoria a un proceso hijo y dejó vivo al proceso número uno	Alerta sobre oomkill mayor que cero y comprueba trabajadores vivos, no solo que el proceso responda.
La petición de parada no tiene ningún efecto durante el plazo de gracia	El proceso número uno no tiene manejadores por defecto, o es un intérprete que no reenvía	Declara el manejador en el programa y usa la forma de lista en el punto de entrada.
Se acumulan procesos zombis hasta agotar el tope	El proceso número uno no recoge a los huérfanos	Usa un proceso de inicio mínimo cuando la aplicación lance hijos que no espera.
Un contenedor sin límite tumba el nodo entero	Sin límite de memoria, el núcleo elige una víctima entre todos los procesos de la máquina	Pon siempre límite: no es ahorro, es acotar el radio de impacto al que se pasa.

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. Demuestra con órdenes sueltas que un contenedor es un proceso con espacios de nombres y grupos de control.
2. ¿Por qué el proceso número uno ignora una señal de parada que no ha declarado, y qué dos causas lo producen?
3. ¿En qué se diferencian el comportamiento del límite de memoria y el de CPU, y qué métrica revela cada uno?
4. ¿Por qué un tiempo de ejecución se dimensiona mal dentro de un contenedor y cómo se corrige?
5. Enumera cuatro cosas que no están aisladas y di qué consecuencia práctica tiene cada una.

Referencias

- Linux (2025). namespaces(7) — los siete espacios de nombres y su semántica. <https://man7.org/linux/man-pages/man7/namespaces.7.html>
- Linux (2025). cgroups(7) y documentación de cgroup v2 — memory.max, cpu.max e interfaces de eventos. <https://man7.org/linux/man-pages/man7/cgroups.7.html>
- Linux (2025). pidnamespaces(7) — semántica de señales del proceso número uno y recolección de huérfanos. <https://man7.org/linux/man-pages/man7/pidnamespaces.7.html>
- Open Container Initiative (2025). Runtime Specification — configuración que aplica el tiempo de ejecución. <https://github.com/opencontainers/runtime-spec/blob/main/config-linux.md>

- Google (2025). gVisor: what is it — aislamiento con núcleo en espacio de usuario y sus compromisos. <https://gvisor.dev/docs/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 05 en PDF · Manual integral

Anterior	Índice	Siguiente
← 062 · Dockerfile reproducible y builds multi-stage	Parte 05 · Programa	064 · Volúmenes, bind mounts y persistencia →

Evaluación — 063 Namespaces, cgroups y runtime de contenedores

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-aislamiento 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.

064 — Volúmenes, bind mounts y persistencia

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 4 Laboratorio: storage · Estado: EXECUTABLECORE

Propósito

Persistir datos en contenedores, que es la primera de las cuatro fugas que la clase 060 predijo y la que más datos ha destruido en la práctica. El sistema de ficheros de un contenedor es efímero por diseño; todo lo que deba sobrevivir es un montaje, y cada tipo de montaje trae su propio problema: el volumen gestionado se puede borrar con una orden de limpieza, el montaje del anfitrión choca con los identificadores de usuario, y el de memoria cuenta contra el límite de memoria de la clase 063.

Resultados de aprendizaje

Al finalizar podrás:

1. Elegir entre volumen gestionado, montaje del anfitrión y montaje en memoria según lo que debe sobrevivir y a qué.
2. Resolver el desajuste de identificadores de usuario sin abrir permisos a todo el mundo.
3. Explicar por qué escribir en la capa de escritura es lento además de efímero.
4. Anticipar las restricciones del almacenamiento en red: un escritor por volumen de bloque, y semántica distinta en sistemas compartidos.
5. Demostrar la recuperación de un volumen en vez de suponerla.

Conceptos centrales

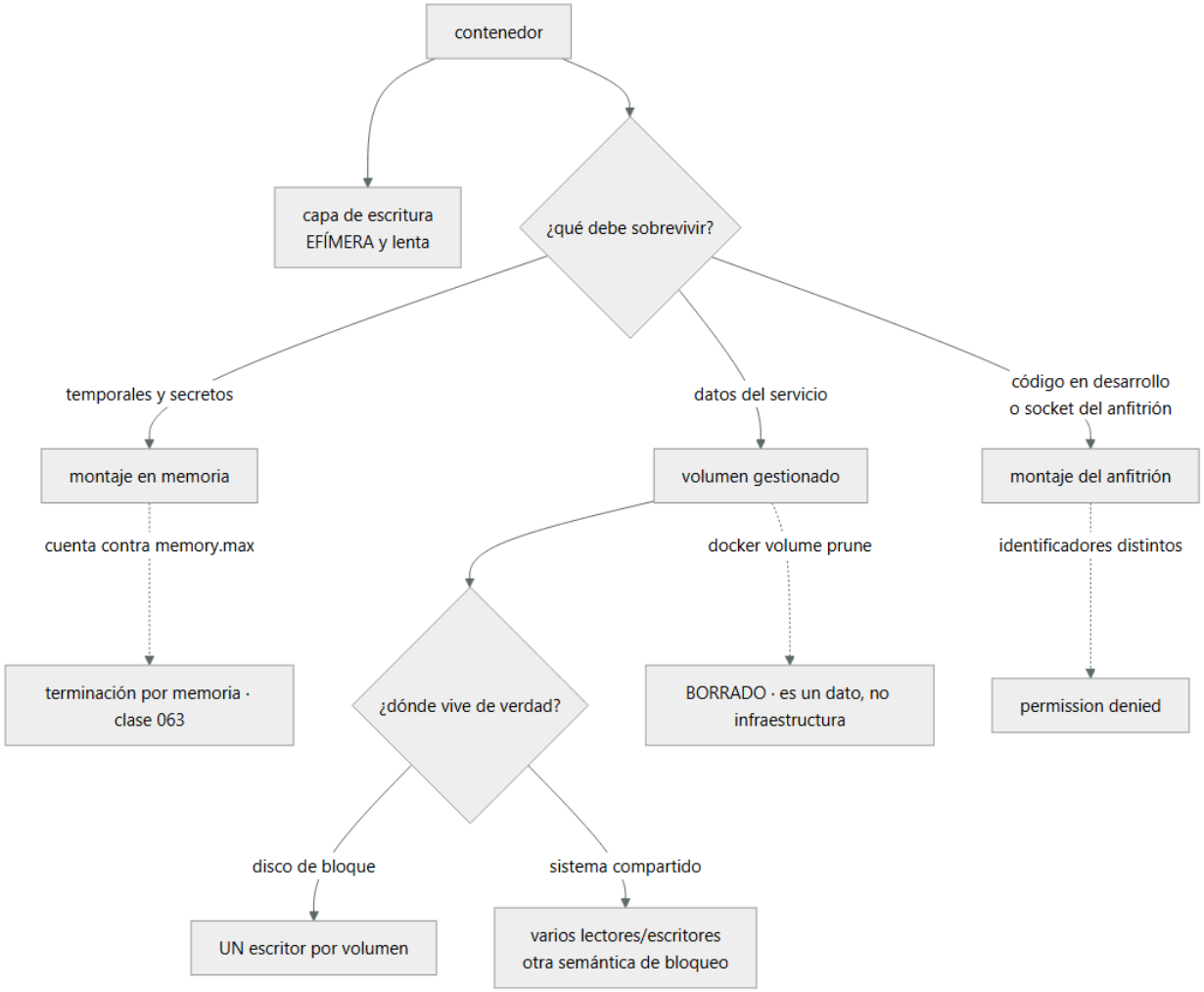
Concepto	Comprensión verificable
capa de escritura	Capa temporal que el motor añade sobre las de la imagen. Muere con el contenedor y, además, es lenta: modificar un fichero existente obliga a copiarlo entero primero.
volumen gestionado	Almacenamiento con nombre que el motor administra. Sobrevive al contenedor y no sobrevive a una orden de limpieza: es datos, no infraestructura.
montaje del anfitrión	Ruta concreta de la máquina montada dentro. Máxima potencia y cero portabilidad: ata el contenedor a un nodo y expone al anfitrión.
montaje en memoria	Sistema de ficheros que nunca toca el disco. Ideal para ficheros temporales y secretos — y cuenta contra el límite de memoria del grupo de control.
desajuste de identificadores	El proceso escribe con un identificador numérico que el directorio del anfitrión no reconoce. Es la causa del permission denied más frecuente, y chmod 777 no es su corrección.
un escritor por volumen	Un volumen de bloque se conecta a un nodo cada vez. Es la restricción que da forma a cualquier carga con estado, y es idéntica en las tres nubes del programa.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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

■ Flujo de razonamiento



Desarrollo

1. Lo efímero es el diseño, no un defecto

Un contenedor arranca con las capas de la imagen en solo lectura y una capa de escritura encima. Todo lo que escriba va ahí, y esa capa se destruye con el contenedor.

Eso no es una limitación que haya que rodear: es la propiedad que hace que un contenedor sea reemplazable, y por tanto que se pueda reiniciar, escalar y desplegar sin ceremonia. Un servicio que necesita conservar su capa de escritura ha dejado de ser reemplazable.

Y hay una segunda razón para no escribir ahí que se conoce menos: es lenta. El sistema de ficheros de unión funciona por copia al escribir, así que la primera modificación de un fichero existente obliga a copiarlo entero a la capa superior antes de tocar un solo byte:

```
modificar 4 KB de un fichero de 2 GB en la capa de escritura
  → se copian los 2 GB primero
modificar 4 KB del mismo fichero en un volumen
  → se escriben 4 KB
```

Por eso una base de datos en la capa de escritura no solo pierde los datos: va mal desde el primer minuto, y el síntoma —latencia de escritura alta que empeora con el tamaño— no apunta a la causa.

Los tres tipos de montaje y para qué sirve cada uno:

	Vive en	Sobrevive al contenedor	Portable	Cuándo
Volumen gestionado	Área del motor o un controlador	Sí	Sí	Datos del servicio

	Vive en	Sobrevive al contenedor	Portable	Cuándo
Montaje del anfitrión	Una ruta concreta de la máquina	Sí	No	Desarrollo, sockets, configuración del nodo
Montaje en memoria	Memoria	No	Sí	Temporales y secretos

```
$ docker run -d \
    -v datos-pedidos:/var/lib/postgresql/data \
    --tmpfs /tmp:rw,noexec,nosuid,size=64m \
    -v /var/run/docker.sock:/var/run/docker.sock:ro \
    imagen:v9
```

Y una regla que ahorra discusiones sobre qué merece un volumen:

```
sí
  ficheros de una base de datos que se opera uno mismo
  contenido subido por usuarios que no está en almacenamiento de objetos
  cachés que compensa conservar entre reinicios

no
  configuración        se inyecta al ejecutar
  secretos             gestor de secretos o montaje en memoria
  registros            salida estándar (clases 045, 057)
  estado que podría estar en un servicio gestionado
```

La última línea merece decirse con claridad después de cuatro partes: para un servicio sin estado, la cantidad correcta de volumen persistente es cero, y las partes 02 a 04 mostraron las alternativas gestionadas para casi todo lo demás. Un volumen es una decisión que hay que justificar, no un valor por defecto.

2. El desajuste de identificadores, y por qué 777 no es la corrección

Es el error más frecuente de esta clase y su corrección equivocada es peor que el problema.

El proceso dentro del contenedor escribe con un identificador numérico —el de USER, de la clase 061—. El directorio del anfitrión pertenece a otro. El núcleo compara números, no nombres, y no hay ninguna traducción:

```
$ docker run --rm -u 10001 -v /datos/tienda:/datos alpine touch /datos/x
touch: /datos/x: Permission denied

$ ls -ldn /datos/tienda
drwxr-xr-x 2 1000 1000 4096 /datos/tienda
```

Dentro es 10001, fuera pertenece a 1000. No hay usuario «app» ni nada parecido: hay números que no coinciden.

Las correcciones, ordenadas de mejor a peor:

```
1. hacer coincidir el identificador
   el proceso escribe con el mismo número que posee el directorio
   → nada que cambiar en el anfitrión, nada que abrir

2. dejar que la plataforma lo resuelva
   los orquestadores ajustan la propiedad del volumen al arrancar
   (fsGroup y equivalentes): es la vía normal en producción

3. ajustar la propiedad al arrancar, desde el punto de entrada
   funciona, exige privilegios para hacerlo y no escala bien

4. espacios de nombres de usuario
   traducen los identificadores del contenedor a otros del anfitrión
   → clase 069; resuelve esto y mucho más
```

Y la que aparece en la mitad de las respuestas de internet:

```
$ chmod -R 777 /datos/tienda      # ← no
```

Abre el directorio a todos los procesos de la máquina, incluidos los de cualquier otro contenedor que monte esa ruta y cualquier proceso del anfitrión. Convierte un problema de configuración en una exposición permanente, y no arregla el caso en que el fichero lo crea el contenedor y luego lo tiene que leer otro proceso con otro identificador.

Un caso particular que conviene conocer porque tiene solución propia: el socket del motor de contenedores. Montarlo dentro de un contenedor —para que una canalización construya imágenes, por ejemplo— es equivalente a dar acceso

de administrador al anfitrión, porque quien habla con ese socket puede arrancar un contenedor con privilegios y montar el disco raíz. No es una exageración:

```
$ docker run -v /var/run/docker.sock:/var/run/docker.sock imagen \
    docker run -v /:/anfitrion --privileged alpine chroot /anfitrion sh
```

La alternativa para construir dentro de contenedores es un constructor sin demonio y sin privilegios, que la clase 069 desarrolla.

Y el montaje de solo lectura merece ser el valor por defecto en todo lo que no necesite escribir, incluida la raíz del contenedor:

```
$ docker run --read-only --tmpfs /tmp -v datos:/var/lib/app imagen:v9
```

Una raíz de solo lectura elimina de golpe la posibilidad de que un atacante escriba un binario o modifique un fichero de configuración, y obliga a declarar explícitamente dónde se escribe. Es una comprobación barata de la lista de la clase 072.

3. El montaje en memoria cuenta contra la memoria

Este detalle une esta clase con la 063 y produce un diagnóstico erróneo que cuesta días.

Un montaje en memoria es un sistema de ficheros que vive en la memoria del anfitrión, y su ocupación se contabiliza en el grupo de control del contenedor:

```
$ docker run -d --memory 512m --tmpfs /tmp:size=1g imagen:v9
```

Esa configuración permite escribir hasta 1 GiB en /tmp con un límite de memoria de 512 MiB. Al llegar a los 512 MiB entre proceso y ficheros temporales, el núcleo termina un proceso del grupo. El síntoma es una terminación por memoria que no se corresponde con el consumo del proceso, así que el equipo busca una fuga que no existe.

```
regla   el tamaño del montaje en memoria SIEMPRE menor que el límite de memoria,
        y con margen para el proceso
```

```
$ docker run -d --memory 512m --tmpfs /tmp:rw,noexec,nosuid,size=64m imagen:v9
```

Las tres opciones adicionales tienen su razón: noexec impide ejecutar desde ahí —que es lo primero que intenta cualquier cadena de explotación que consigue escribir—, nosuid desactiva la elevación por permisos especiales, y el tamaño acotado impide que un fichero temporal descontrolado consuma la memoria del contenedor.

Y el uso más valioso del montaje en memoria es para secretos: un fichero que nunca toca el disco, que desaparece con el contenedor y que no puede quedarse en una instantánea del volumen. Es el mecanismo que la clase 058 recomendaba para que una rotación se pudiera releer sin redesplegar.

Hay un segundo consumo de memoria menos evidente y que aparece en los diagnósticos de la clase 070: la caché de páginas de los ficheros que el contenedor lee también se contabiliza. Un proceso que lee secuencialmente un fichero grande puede acercar el grupo a su límite sin haber reservado memoria, y el núcleo responde desalojando caché —con el resultado de que la E/S se dispara y el servicio se ralentiza sin que ninguna métrica de memoria del proceso lo explique.

```
$ cat /sys/fs/cgroup/memory.stat | grep -E '^(anon|file|slab) '
anon   183238656
file   291504128        ← caché de ficheros, dentro del mismo límite
slab    12058624
```

Saber que esa fila existe es lo que separa «el contenedor consume 460 MiB» de «el proceso consume 183 MiB y hay 291 de caché que el núcleo puede soltar».

4. Almacenamiento en red: un escritor, y la semántica que cambia

En cuanto el contenedor puede ejecutarse en cualquier nodo, el volumen tiene que estar disponible desde cualquier nodo, y aparecen dos familias con propiedades muy distintas.

Volumen de bloque —el disco de las clases 040, 052 y equivalentes—:

```
se conecta a UN nodo cada vez
rendimiento y semántica de un disco local
si el contenedor se mueve, el volumen se desconecta y se vuelve a conectar
```

La restricción de un solo escritor es la que da forma a toda carga con estado, y es idéntica en las tres nubes. Consecuencias prácticas:

```
no se pueden tener dos réplicas del mismo servicio escribiendo el mismo volumen
un despliegue progresivo con volumen de bloque exige apagar antes de arrancar,
  porque el nuevo no puede conectar hasta que el viejo suelte
el tiempo de desconexión y reconexión forma parte del tiempo de recuperación
```

La segunda es la que sorprende: un servicio con estado no admite el patrón de «crear el nuevo antes de retirar el viejo» que la clase 052 recomendaba para los sin estado. Hay que invertir el orden y aceptar una ventana.

Sistema de ficheros compartido —los servicios de archivos de las tres nubes—:

```
varios lectores y escritores a la vez
semántica de red: el bloqueo de ficheros y la confirmación en disco
  NO se comportan igual que en un disco local
```

La segunda línea es la que produce corrupciones difíciles de explicar. Un motor de base de datos que asume que una confirmación en disco garantiza durabilidad puede recibir esa confirmación de un cliente de red que aún no ha escrito, y un bloqueo de fichero que en local es atómico puede no serlo. Por eso la recomendación de casi todos los motores es no ponerlos sobre un sistema de ficheros de red, y por eso conviene tratar esa recomendación como un requisito y no como una preferencia.

Un sistema compartido sí es la respuesta correcta para lo que de verdad necesita acceso simultáneo: contenido subido, artefactos compartidos, directorios de trabajo de procesos por lotes que se coordinan por otro medio.

Y una tercera opción que este programa ya ha usado tres veces y conviene volver a poner sobre la mesa: no usar un volumen. Contenido subido por usuarios en almacenamiento de objetos (clases 030, 041, 053), estado de sesión en una caché gestionada, datos relacionales en un servicio gestionado. La pregunta antes de crear cualquier volumen es la misma:

```
¿este dato existe porque el diseño lo necesita en un sistema de ficheros,
 o porque la aplicación se escribió cuando no había otra opción?
```

5. Un volumen con datos necesita una restauración probada

Los volúmenes se borran con una orden, y las órdenes de limpieza son cómodas:

```
$ docker volume prune -f        # borra TODOS los volúmenes sin contenedor asociado
$ docker compose down -v        # la -v borra los volúmenes del proyecto
```

La segunda es especialmente peligrosa porque aparece en guiones de reinicio y en instrucciones de «empezar de cero». Un contenedor parado durante un mantenimiento deja su volumen sin asociar, y una limpieza rutinaria se lo lleva.

Es la cuarta vez que este programa llega a la misma conclusión, después de las clases 030, 041 y 053:

> Un almacén de datos sin una restauración probada no tiene copia de seguridad, tiene una intención.

El procedimiento completo para un volumen, que cabe en tres órdenes y hay que ensayar:

```
# copia
$ docker run --rm -v datos-pedidos:/origen:ro -v $(pwd):/destino alpine \
    tar czf /destino/pedidos-$(date +%F).tar.gz -C /origen .

# restauración en un volumen NUEVO, nunca sobre el original
$ docker volume create datos-pedidos-prueba
$ docker run --rm -v datos-pedidos-prueba:/destino -v $(pwd):/origen:ro alpine \
    tar xzf /origen/pedidos-2026-08-01.tar.gz -C /destino

# verificación cuantitativa: no "parece bien", sino un recuento
$ docker run --rm -v datos-pedidos-prueba:/var/lib/postgresql/data postgres:16 \
    postgres --single -D /var/lib/postgresql/data pedidos <<< 'SELECT count(*) FROM pedidos;'
```

El tercer paso es el que casi nunca se hace y el único que demuestra algo. Y para una base de datos, hay una precisión importante: copiar los ficheros de un motor en marcha no produce una copia consistente. Hay que usar el mecanismo del propio motor —una copia en caliente, un volcado lógico— o detener el servicio. Un archivo tar de un directorio de datos vivo restaura a menudo, y cuando no restaura es siempre en el peor momento.

Las tres protecciones que conviene tener antes del primer dato real:

```
1. copia con el mecanismo del motor, no del sistema de ficheros
2. restauración mensual sobre un volumen nuevo, con recuento verificado
3. la duración de esa restauración se REGISTRA: es el tiempo de recuperación real
```

La tercera se olvida siempre y es la única cifra que un plan de recuperación puede usar. En las clases 042 y 048 esa medición dio números bastante mayores que los que figuraban en el plan, y no hay ninguna razón para que aquí sea distinto.

Y una última comprobación que cierra la clase y enlaza con la 072: el inventario de volúmenes con dueño. Un volumen sin nombre reconocible y sin responsable es un dato que nadie sabe si se puede borrar, y esa duda se resuelve siempre de la peor manera —conservándolo para siempre o borrándolo justo cuando hacía falta—.

```
$ docker volume ls --format '{{.Name}}\t{{.Labels}}' | grep -v 'sistema='
```

Ejemplo trabajado

CloudShop mueve a contenedores un servicio con estado. Los cinco problemas del primer trimestre son todos de persistencia, y el más caro no fue una pérdida de datos sino un diagnóstico equivocado que duró once días.

Problema 1 — permission denied y la corrección que abrió el anfitrión.

Al montar el directorio de subidas, el contenedor no podía escribir. Se resolvió esa tarde:

```
$ chmod -R 777 /datos/subidas
```

Seis semanas después, una revisión de seguridad lo marcó: cualquier proceso del anfitrión y cualquier contenedor que montara esa ruta podía leer y modificar el contenido subido por los usuarios.

permisos del directorio	777	750
identificador del proceso	10001	10001
propietario del directorio	1000	10001
procesos del anfitrión con acceso	todos	1
montaje	lectura-escritura	lectura-escritura
raíz del contenedor	escribible	solo lectura + tmpfs

Hacer coincidir el número fue la corrección entera. No hizo falta cambiar la imagen ni el anfitrión: solo el propietario del directorio.

Problema 2 — la base de datos que iba mal y además se perdió.

Un servicio auxiliar guardaba su base en la capa de escritura. Iba lento desde el principio y nadie sabía por qué; un redespliegue rutinario se llevó cuatro días de datos.

ubicación de los datos	capa de escritura	volumen gestionado
latencia de escritura p95	84 ms	6 ms
datos tras un redespliegue	se pierden	persisten
causa de la lentitud	copia al escribir del fichero completo en la primera modificación	escritura directa

Catorce veces más rápido por cambiar dónde se escribe. La lentitud y la pérdida tenían la misma causa y solo una era visible.

Problema 3 — once días buscando una fuga de memoria que no existía.

terminaciones por memoria	3-5 al día
memoria del proceso según su propio informe	210 MiB
límite del contenedor	512 MiB

El proceso consumía menos de la mitad del límite y aun así lo cruzaba.

```
$ kubectl exec proc-4a1 -- df -h /tmp
Filesystem   Size  Used Avail Use% Mounted on
tmpfs        1.0G  289M  735M  29% /tmp
$ kubectl exec proc-4a1 -- cat /sys/fs/cgroup/memory.stat | grep -E '^(anon|file) '
anon 220200960
file 303038464
```

El montaje en memoria tenía 1 GiB de tamaño con un límite de contenedor de 512 MiB, y los ficheros temporales de la conversión de imágenes contaban contra ese límite.

tamaño del montaje en memoria	1 GiB	64 MiB
opciones del montaje	rw	rw,noexec,nosuid
terminaciones por memoria al día	3-5	0
días buscando una fuga inexistente	11	—

La métrica que lo habría mostrado el primer día estaba disponible desde el principio: la separación entre memoria anónima y caché de ficheros en las estadísticas del grupo de control.

Problema 4 — una limpieza rutinaria borró producción.

```
$ docker volume prune -f
Deleted Volumes:
datos-pedidos
datos-informes
Total reclaimed space: 41.7GB
```

El servicio estaba parado por un mantenimiento, así que sus volúmenes figuraban como no asociados. La copia de seguridad existía —un archivo tar del directorio de datos con el motor en marcha— y no restauró: el motor rechazó los ficheros por inconsistencia.

```
mecanismo de copia              tar del directorio vivo   copia en caliente
                                                            del propio motor
restauración probada                      nunca         mensual, en volumen nuevo
verificación                             ninguna        recuento de filas
tiempo de recuperación en el plan         "1 hora"       2 h 40 min medidos
órdenes de limpieza en guiones          prune -f        lista explícita de nombres
etiquetas de propietario en volúmenes       no                sí
```

La recuperación real se hizo desde una réplica lógica que existía por otro motivo. La cifra de 2 h 40 min es la del ensayo posterior, y sustituyó a la hora que figuraba en el plan sin haberse medido nunca.

Problema 5 — dos réplicas y un volumen de bloque.

Al escalar el servicio a dos réplicas, la segunda no arrancaba:

```
MultiAttachError: Volume is already exclusively attached to one node
```

No era un fallo de configuración: es la restricción del volumen de bloque. Se replanteó el servicio en vez de buscar un rodeo:

```
contenido subido                  volumen de bloque    almacenamiento de objetos
                                  compartido            (clase 053)
índice de búsqueda local          volumen de bloque    se reconstruye al arrancar
réplicas posibles                        1                     6
volúmenes persistentes                   2                     0
```

Al final del ejercicio, el servicio no necesitaba ningún volumen. Los dos que tenía existían porque la aplicación se había escrito cuando el almacenamiento de objetos no formaba parte del diseño.

Resumen de la persistencia:

```
permisos del directorio de subidas            777            750
latencia p95 de escritura                    84 ms           6 ms
terminaciones por memoria al día              3-5             0
volúmenes en producción                         4             1
restauración probada                           no        mensual, verificada
tiempo de recuperación                    "1 hora"        2 h 40 min medidos
réplicas máximas del servicio                   1             6
raíz del contenedor escribible                 sí            no
```

La lección que esta clase traslada al resto de la parte 05: los cinco problemas venían de tratar el almacenamiento del contenedor como si fuera el de una máquina. Y el resultado más útil es el último: al terminar, el servicio necesitaba un volumen en vez de cuatro, porque tres de ellos existían por inercia de un diseño anterior. Antes de resolver un problema de persistencia conviene preguntar si hay que persistir eso, y las partes 02 a 04 dieron la alternativa gestionada para casi todos los casos.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/064-volumenes-bind-mounts-y-persistencia/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 persistencia-contenedor 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 persistencia-contenedor 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
permission denied al escribir en un montaje del anfitrión	El identificador numérico del proceso no coincide con el propietario del directorio	Haz coincidir el identificador o deja que la plataforma ajuste la propiedad; chmod 777 expone el directorio a toda la máquina.
La escritura es lenta y empeora con el tamaño de los ficheros	Se escribe en la capa de escritura, que copia el fichero entero en la primera modificación	Monta un volumen para los datos; la capa de escritura es para lo efímero y pequeño.
Terminaciones por memoria con el proceso muy por debajo del límite	Un montaje en memoria mayor que el límite del contenedor consume ese mismo presupuesto	Dimensiona el montaje muy por debajo del límite y añade noexec,nosuid; revisa la separación entre memoria anónima y caché.
Una limpieza rutinaria borra datos de producción	Los volúmenes de un servicio parado figuran como no asociados y las órdenes de purga los eliminan	Nunca purgues por defecto: borra por nombre explícito, etiqueta los volúmenes con su responsable y ten una restauración probada.
Una copia del directorio de datos no restaura	Se copió con el motor en marcha, así que los ficheros no son consistentes entre sí	Usa el mecanismo de copia del propio motor y verifica cada restauración con un recuento, registrando su duración.
La segunda réplica no arranca por el volumen	Un volumen de bloque se conecta a un solo nodo a la vez	Mueve el dato a almacenamiento de objetos o a un servicio gestionado, o acepta una sola réplica y una ventana en el despliegue.

■ Seguridad, ética y costo

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

■ Preguntas de comprobación

1. ¿Por qué escribir en la capa de escritura es lento además de efímero?
2. ¿Cuál es la corrección correcta del desajuste de identificadores y por qué chmod 777 es peor que el problema?
3. ¿Qué relación hay entre un montaje en memoria y el límite de memoria del contenedor?
4. ¿Qué restricción impone un volumen de bloque a un despliegue progresivo, y por qué?
5. ¿Qué tres pasos convierten una copia de seguridad de un volumen en algo demostrable?

Referencias

• Docker (2025). Manage data in Docker — volúmenes, montajes del anfitrión y montajes en memoria. <https://docs.docker.com/engine/storage/>

• Docker (2025). Storage drivers and copy-on-write — coste de la primera escritura sobre un fichero existente. <https://docs.docker.com/engine/storage/drivers/>

• Linux (2025). tmpfs — contabilidad de memoria y opciones de montaje. <https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html>

• Kubernetes (2025). Persistent volumes and access modes — un escritor por volumen de bloque y sistemas compartidos. <https://kubernetes.io/docs/concepts/storage/persistent-volumes/>

• Docker (2025). Back up, restore, or migrate data volumes — procedimiento y sus límites con motores en marcha. <https://docs.docker.com/engine/storage/volumes/#back-up-restore-or-migrate-data-volumes>

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

Descargar: Parte 05 en PDF · Manual integral

Anterior	Índice	Siguiente
← 063 · Namespaces, cgroups y runtime de contenedores	Parte 05 · Programa	065 · Redes bridge, DNS interno y publicación de puertos →

Evaluación — 064 Volúmenes, bind mounts y persistencia

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 persistencia-contenedor 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.

065 — Redes bridge, DNS interno y publicación de puertos

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 4 Laboratorio: network · Estado: EXECUTABLECORE

Propósito

Conectar contenedores entre sí y con el exterior, que es la segunda fuga que la clase 060 predijo y la que produce los diagnósticos más largos. Tres hechos concretos explican casi todos los incidentes: la red por defecto no resuelve nombres entre contenedores, publicar un puerto puede saltarse el cortafuegos del anfitrión, y una unidad máxima de transmisión mal ajustada hace que las peticiones pequeñas funcionen y las grandes se queden colgadas — el fallo de red más difícil de atribuir que existe.

Resultados de aprendizaje

Al finalizar podrás:

1. Explicar qué crea el motor al conectar un contenedor y por qué la red por defecto no resuelve nombres.
2. Publicar un puerto sabiendo qué regla se inserta y por qué puede eludir el cortafuegos del anfitrión.
3. Diagnosticar un problema de unidad máxima de transmisión distinguiéndolo de un fallo de aplicación.
4. Anticipar el efecto de la caché de resolución de nombres cuando un contenedor cambia de dirección.
5. Elegir el modo de red adecuado y justificar cuándo compensa renunciar al aislamiento.

Conceptos centrales

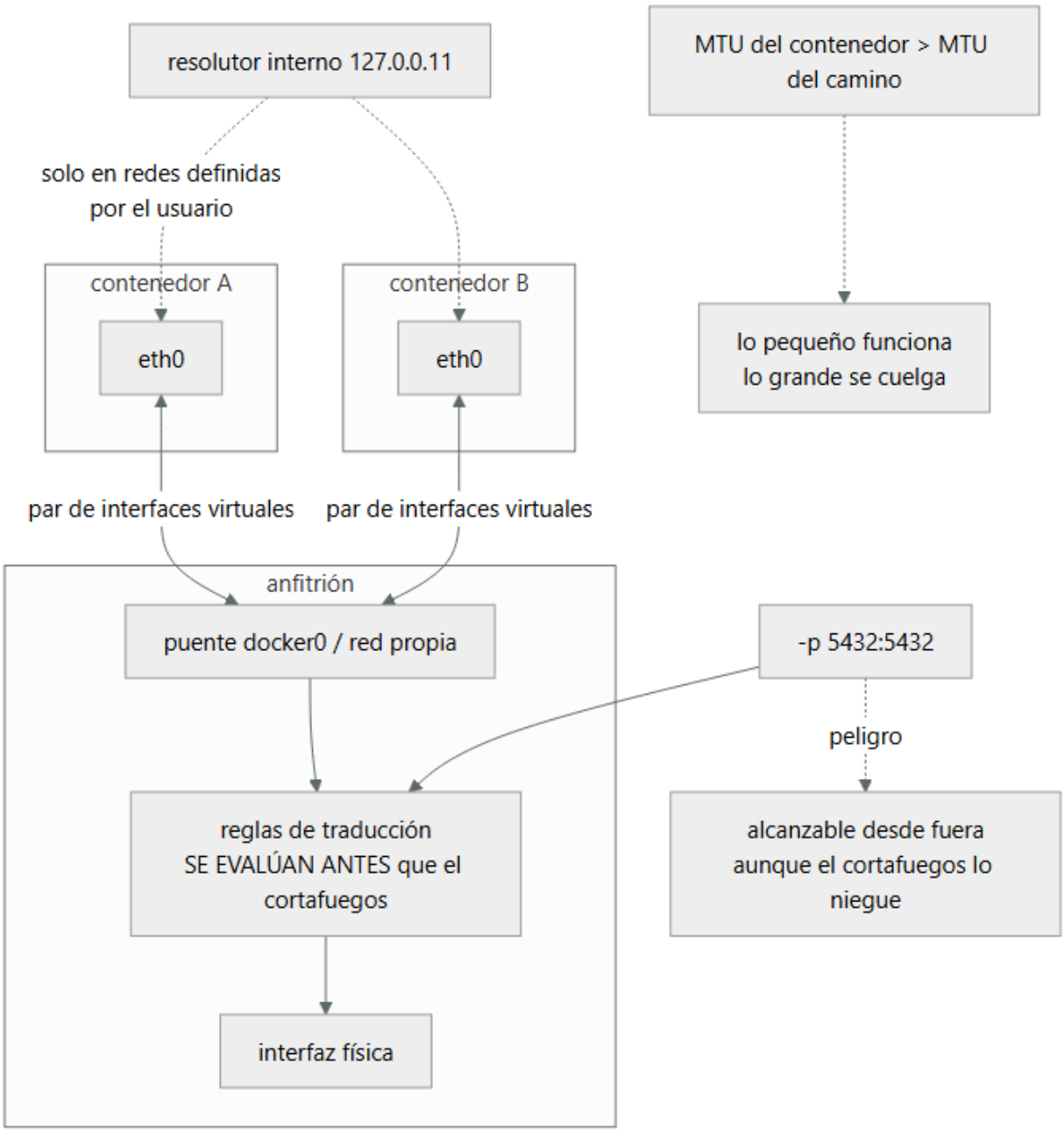
Concepto	Comprensión verificable
par de interfaces virtuales	Cable virtual con dos extremos: uno dentro del contenedor y otro en el anfitrión, conectado a un puente. Es toda la conectividad del modo por defecto.
red definida por el usuario	Red propia con resolución de nombres integrada. La red por defecto no la tiene: ahí los contenedores solo se alcanzan por dirección.
publicación de puertos	Regla de traducción en el anfitrión que redirige un puerto suyo al contenedor. Se evalúa antes que muchas reglas de cortafuegos, así que puede exponer lo que se creía cerrado.
unidad máxima de transmisión	Tamaño máximo de paquete. Si la del contenedor es mayor que la del camino real, los paquetes grandes se descartan: lo pequeño funciona y lo grande se cuelga.
caché de resolución	Dirección guardada por la aplicación tras resolver un nombre. Si el contenedor destino se recrea con otra dirección, el cliente sigue hablando con una que ya no existe.
modo de red del anfitrión	El contenedor comparte la pila de red de la máquina: sin traducción, sin aislamiento y sin publicación. Se elige por rendimiento, no por comodidad.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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 crea el motor, y por qué la red por defecto no resuelve nombres

Al arrancar un contenedor en el modo habitual, el motor hace tres cosas:

```
1. crea un espacio de nombres de red propio (clase 063)
2. crea un par de interfaces virtuales: un extremo dentro, otro en el anfitrión
3. conecta el extremo del anfitrión a un puente y añade reglas de traducción
```

Se puede ver desde los dos lados:

```
$ docker run -d --name a --network cls alpine sleep 1d
$ docker exec a ip -br addr
lo    UNKNOWN  127.0.0.1/8
eth0  UP       172.19.0.2/16

$ ip -br link | grep veth
veth8a1c2f@if12  UP
$ bridge link | grep veth8a1c2f
… master br-4f2a91c
```

Y aquí está la diferencia que causa el «no encuentro el otro contenedor»:

```
red por defecto (bridge)        NO hay resolución de nombres entre contenedores
red definida por el usuario     SÍ: un resolutor interno traduce el nombre
                                del contenedor o del servicio a su dirección
```

```
$ docker run --rm alpine ping -c1 a                  # red por defecto
ping: bad address 'a'

$ docker network create cls
$ docker run --rm --network cls alpine ping -c1 a    # red propia
64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.077 ms
```

La regla operativa es corta: nunca se usa la red por defecto. Una red por aplicación o por grupo de servicios da resolución de nombres, aislamiento respecto de contenedores ajenos del mismo anfitrión y un rango de direcciones controlado.

El resolutor vive en una dirección de bucle local dentro del contenedor y reenvía al del anfitrión lo que no conoce:

```
$ docker exec a cat /etc/resolv.conf
nameserver 127.0.0.11
options ndots:0
```

Y de ahí sale un fallo que aparece al conectar un contenedor a varias redes: el orden de resolución y de rutas depende del orden de conexión, así que un contenedor en dos redes puede salir por la que no se esperaba. La corrección es no conectar a varias redes salvo que haga falta, y cuando haga falta, declarar explícitamente por dónde sale.

Una precisión sobre lo que la resolución de nombres no arregla: sigue siendo una dirección, y las direcciones cambian. Un contenedor recreado obtiene otra, así que la resolución hay que hacerla cuando se necesita, no una vez al arrancar. Es la tercera vez que la resolución de nombres aparece como causa raíz en este programa —la zona privada de la clase 039 y el acceso privado de la 051— y no será la última.

2. Publicar un puerto no es abrir un puerto: es insertar una regla

Este es el hecho de seguridad más importante de la clase y el menos conocido.

```
$ docker run -d -p 5432:5432 postgres:16
```

Eso no «abre» nada en el sentido del cortafuegos: inserta una regla de traducción de destino que redirige el puerto del anfitrión al contenedor. Y esa regla se evalúa en una cadena que, en la configuración habitual, se procesa antes que las reglas del cortafuegos de usuario.

La consecuencia es directa y sorprende siempre:

```
$ sudo ufw status
Status: active
To          Action  From
22/tcp      ALLOW   Anywhere
(todo lo demás denegado)

$ nmap -p 5432 el-servidor-desde-fuera
PORT     STATE
5432/tcp open        ← el cortafuegos dice que no y el puerto responde
```

Una base de datos publicada con la sintaxis corta queda escuchando en todas las interfaces, incluida la pública, con un cortafuegos activo que parece protegerla. Es una de las causas más habituales de bases de datos expuestas en internet.

Las tres correcciones, y hay que aplicar al menos una:

```
1. publicar solo en bucle local
   -p 127.0.0.1:5432:5432        ← accesible desde el anfitrión y de nadie más

2. no publicar en absoluto
   los contenedores de la misma red se alcanzan por nombre y puerto interno;
   publicar solo lo que de verdad entra desde fuera

3. reglas de cortafuegos en la cadena correcta
   la que el motor consulta, no la de usuario
```

La segunda es la correcta en la mayoría de los casos y la que menos se usa. En una aplicación con base de datos, caché y servicio web, lo único que debería publicarse es el puerto del servicio web. Todo lo demás se habla por la red

interna.

Y la comprobación, que es una prueba negativa en el sentido de las clases 046 y 058:

```
$ ss -ltnp | grep 5432
LISTEN 0 4096 127.0.0.1:5432   ← solo bucle local                    ✓
$ nmap -p 5432 $IP_PUBLICA
5432/tcp filtered                                                    ✓
```

Sobre la salida, el mecanismo es la traducción de origen: el contenedor sale con la dirección del anfitrión. Eso trae dos consecuencias conocidas de las partes anteriores:

```
el destino ve la IP del anfitrión, no la del contenedor
  → las listas de permitidos de terceros se hacen con la del anfitrión
el número de conexiones simultáneas al mismo destino tiene un tope
  → es el agotamiento de puertos de traducción de las clases 039, 043 y 051
```

Cuarta aparición del mismo problema, con un cuarto mecanismo. Y la misma corrección de siempre en la aplicación: reutilizar el cliente en vez de abrir una conexión por petición.

Y un conflicto de direcciones que aparece en redes corporativas: el motor elige rangos privados por defecto para sus puentes, y esos rangos pueden coincidir con los de la red de la empresa. El síntoma es que algunos destinos internos dejan de ser alcanzables desde los contenedores y otros no, según qué rango colisione. Se corrige declarando los rangos:

```
$ cat /etc/docker/daemon.json
{"default-address-pools": [{"base": "10.240.0.0/12", "size": 24}]}
```

Es la misma planificación de direcciones de las clases 027, 039 y 051, aplicada al anfitrión.

3. La unidad máxima de transmisión: lo pequeño funciona y lo grande se cuelga

Merece una sección propia porque es el fallo de red más difícil de atribuir, y su síntoma engaña a todo el mundo.

Si el contenedor cree que puede enviar paquetes de 1.500 bytes y el camino real solo admite 1.400 —porque hay una red superpuesta, un túnel o una red privada virtual por medio—, los paquetes grandes se descartan. Y como las peticiones pequeñas caben, todo parece funcionar:

```
funciona     conexión, negociación de cifrado, peticiones pequeñas,
             comprobaciones de estado, `curl` de una página
se cuelga    subidas, respuestas grandes, consultas con mucho resultado,
             clonar un repositorio
```

El diagnóstico habitual va a la aplicación, al servidor, al tiempo de espera. Y la comprobación que lo identifica en un minuto es esta:

```
# paquete de 1472 + 28 de cabeceras = 1500, sin fragmentar
$ docker exec app ping -M do -s 1472 -c1 destino.interno
ping: local error: message too long, mtu=1400

$ docker exec app ping -M do -s 1372 -c1 destino.interno
64 bytes from …: seq=0 ttl=63 time=1.21 ms          ← 1400 sí cabe
```

Y el ajuste, en la red del contenedor:

```
$ docker network create --opt com.docker.network.driver.mtu=1400 cls
$ docker exec app ip link show eth0 | head -1
2: eth0@if14: <BROADCAST,MULTICAST,UP> mtu 1400
```

La causa profunda es que el mecanismo que debería resolverlo solo —el descubrimiento de la unidad máxima del camino— depende de mensajes de control que muchos cortafuegos bloquean por costumbre. Cuando esos mensajes no llegan, el emisor nunca se entera de que sus paquetes son demasiado grandes y sencillamente reintenta hasta agotar el tiempo.

Dos lugares donde esto aparece casi siempre:

```
redes superpuestas entre nodos    encapsulan y restan bytes a cada paquete
redes privadas virtuales          lo mismo, con otro encabezado
```

Y una regla práctica que evita la clase entera de incidentes: al montar cualquier red que encapsule, se comprueba la unidad máxima efectiva antes de desplegar nada, con la orden de arriba. Cuesta un minuto y ahorra los días que cuesta encontrarlo después, cuando el síntoma es «las subidas grandes fallan a veces».

4. Las direcciones cambian y la aplicación no se entera

La resolución de nombres integrada resuelve el nombre a una dirección en el momento de preguntar. Un contenedor recreado obtiene otra dirección, y ahí empieza el problema.

```
$ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' bd
172.19.0.3
$ docker compose up -d --force-recreate bd
$ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' bd
172.19.0.7                    ← otra
```

Si el cliente resolvió al arrancar y guardó la dirección, sigue intentando la antigua. El síntoma es característico: el servicio funciona hasta que se reinicia su dependencia, y entonces falla hasta que se reinicia él también. Y como reiniciarlo lo arregla, se archiva como «cosa rara» en vez de corregirse.

Los culpables habituales:

```
tiempos de ejecución que cachean para siempre por defecto
  ciertas máquinas virtuales guardan la resolución sin caducidad
  salvo que se les indique lo contrario
grupos de conexiones que resuelven una vez al crearse
clientes que guardan la dirección en lugar del nombre
```

Las correcciones, por orden:

```
1. que el cliente resuelva por NOMBRE en cada conexión nueva
2. acotar la caché de resolución a segundos, no a infinito
3. que el grupo de conexiones cierre y recree conexiones periódicamente
4. reintentar ante un fallo de conexión en lugar de propagarlo
```

La cuarta es la que convierte el incidente en invisible, y es la misma conclusión que este programa ha alcanzado ya cuatro veces con las conmutaciones de bases de datos gestionadas: una dependencia que cambia de sitio es funcionamiento normal, y el cliente tiene que sobrevivir a ello.

Y un detalle de la resolución interna que conviene conocer para no perseguirlo: la opción de puntos que el motor escribe en la configuración del resolutor hace que ciertos nombres se intenten primero con sufijos de búsqueda. En entornos con muchos sufijos, eso multiplica las consultas y añade latencia a cada resolución nueva. Se nota en aplicaciones que abren muchas conexiones a nombres externos, y se corrige poniendo el nombre completo con punto final o ajustando la opción.

```
$ time docker exec app getent hosts api.proveedor.com
real    0m0.312s          ← cinco consultas fallidas antes de la buena
$ time docker exec app getent hosts api.proveedor.com.
real    0m0.021s
```

5. Los cuatro modos de red y cuándo compensa renunciar al aislamiento

```
puente (por defecto)   espacio de red propio, traducción, publicación de puertos
anfitrión              comparte la pila de red de la máquina
ninguna                sin red: solo bucle local
compartida             usa el espacio de red de OTRO contenedor
```

El modo anfitrión elimina la traducción y el par de interfaces, y con ellos su coste. Es la respuesta correcta para cargas con muchísimo tráfico o muy sensibles a la latencia, y hay que aceptar lo que se pierde:

```
se gana   sin traducción: menos latencia y más rendimiento por núcleo
          la aplicación ve la IP de origen real de los clientes
se pierde aislamiento de red: el contenedor puede escuchar en cualquier puerto
          del anfitrión y ver todo su tráfico
          no hay publicación: los puertos son los del anfitrión, con sus conflictos
```

La penúltima línea es la que decide: en una máquina compartida por varios equipos, el modo anfitrión reparte el espacio de puertos entre todos y un servicio puede ocupar el puerto de otro.

El modo compartido es el más interesante conceptualmente, porque explica dos cosas del resto del programa:

```
$ docker run --rm -it --network container:app --pid container:app \
    nicolaka/netshoot ss -ltnp
```

Eso arranca un contenedor con herramientas de diagnóstico dentro de la red y los procesos del contenedor que se investiga, lo que resuelve el problema que la clase 062 dejó abierto: cómo depurar una imagen sin intérprete de órdenes. Es la técnica que la clase 070 desarrolla.

Y es exactamente el mecanismo con el que funciona la unidad de despliegue de la parte 06: varios contenedores que comparten espacio de red se ven entre sí en el bucle local y comparten una única dirección. Conocerlo aquí hace que aquello no parezca magia:

```
contenedores que comparten espacio de red
  → se llaman por 127.0.0.1
  → comparten los puertos: dos no pueden escuchar en el mismo
  → una sola dirección para el conjunto
```

Y el modo sin red tiene un uso real que se olvida: un proceso por lotes que solo lee de un volumen y escribe en él no necesita red, y quitársela elimina una superficie entera. Es la aplicación del privilegio mínimo a la conectividad, y es gratis.

Para terminar, el criterio de elección resumido:

```
puente con red propia   valor por defecto para todo
anfitrión               rendimiento medido que lo justifique, máquina dedicada
ninguna                 procesos por lotes sin dependencias de red
compartida              diagnóstico, y como modelo mental de la parte 06
```

Y la comprobación que cierra la clase, que debería estar en el guion de verificación de la 072:

```
# ningún servicio interno publicado hacia fuera
$ docker ps --format '{{.Names}}\t{{.Ports}}' | grep -v '127.0.0.1' | grep '0.0.0.0'
(vacío salvo el frontal)                                                    ✓
```

Ejemplo trabajado

CloudShop conecta sus contenedores. La aplicación funciona en el portátil de todo el mundo y produce cinco incidentes en cuanto sale de ahí — dos de conectividad, uno de seguridad y dos que tardaron días porque el síntoma no señalaba a la red.

Incidente 1 — el servicio no encuentra la base de datos.

```
Error: getaddrinfo ENOTFOUND bd
```

Ambos contenedores estaban en la red por defecto, que no resuelve nombres. La solución de urgencia fue usar la dirección; la correcta, una red propia.

```
red                                     por defecto      una por aplicación
resolución entre contenedores               no               sí
referencia en la configuración          172.17.0.3          bd:5432
al recrear la base de datos            hay que reconfigurar   sigue funcionando
```

Incidente 2 — la base de datos estaba en internet.

Una revisión externa encuentra el puerto 5432 accesible desde fuera del centro de datos, con el cortafuegos del anfitrión activo y denegando todo salvo el 22.

```
$ docker ps --format '{{.Names}}\t{{.Ports}}'
bd    0.0.0.0:5432->5432/tcp
$ sudo iptables -t nat -L DOCKER -n | grep 5432
DNAT  tcp -- 0.0.0.0/0  0.0.0.0/0  tcp dpt:5432 to:172.19.0.3:5432
```

La regla de traducción se evaluaba antes que las del cortafuegos de usuario. La base había estado alcanzable once días.

```
publicación de la base de datos        0.0.0.0:5432       no se publica
publicación de la caché                0.0.0.0:6379       no se publica
puertos publicados en total                  4                 1
contraseña de la base de datos          la de siempre    rotada el mismo día
prueba negativa desde fuera               ninguna       en el guion de verificación
```

El orden fue el de todas las exposiciones de este programa: rotar primero, corregir después. Cuarta vez.

Incidente 3 — las subidas grandes se quedaban colgadas.

Durante tres semanas, las subidas de facturas de más de unos 200 KB fallaban «a veces». Las pequeñas siempre funcionaban, la comprobación de estado siempre en verde, y el proveedor de la red privada virtual decía que todo estaba bien.

```
$ docker exec app ping -M do -s 1472 -c1 almacen.interno
ping: local error: message too long, mtu=1400
```

La red privada virtual entre el centro de datos y el proveedor restaba 100 bytes por paquete, y la red del contenedor seguía anunciando 1.500.

```
unidad máxima de la red del contenedor      1.500            1.400
subidas grandes fallidas                   ~18 %             0 %
días hasta encontrar la causa                21               —
comprobación al montar una red que encapsula  ninguna    obligatoria, documentada
```

Veintiún días para una comprobación de un minuto. La medida que evita la repetición no es el ajuste: es haber puesto la comprobación en la lista de puesta en marcha de cualquier red que encapsule.

Incidente 4 — el servicio falla cada vez que se reinicia la base de datos.

```
patrón   se reinicia la base de datos → el servicio empieza a fallar
         se reinicia el servicio → todo vuelve a funcionar
```

El cliente resolvía el nombre al crear el grupo de conexiones y la máquina virtual del lenguaje guardaba esa resolución sin caducidad, que era su valor por defecto.

```
caducidad de la caché de resolución      indefinida           30 s
resolución                              una vez al arrancar  por conexión nueva
renovación del grupo de conexiones         nunca          cada 30 min
reintento ante fallo de conexión             no               sí
reinicios del servicio por este motivo    9 en 2 meses          0
```

Incidente 5 — algunos destinos internos dejaron de ser alcanzables.

Desde los contenedores de un nodo concreto, tres sistemas internos no respondían y el resto sí.

```
$ docker network inspect cls -f '{{range .IPAM.Config}}{{.Subnet}}{{end}}'
172.20.0.0/16
$ ip route | grep 172.20
172.20.0.0/16 dev br-4f2a91c        ← y la empresa usa 172.20.4.0/24 para nóminas
```

El puente había elegido un rango que colisionaba con una red corporativa, así que el tráfico hacia ella se quedaba en el puente local.

```
rangos del motor                        elegidos solos    declarados: 10.240.0.0/12
destinos internos inalcanzables               3                 0
coordinación con el plan de direcciones      ninguna     el mismo de la clase 051
```

Resumen de la red:

```
redes definidas por el usuario                    0             3
puertos publicados hacia todas las interfaces     4             1
días de la base de datos expuesta                11             0
subidas grandes fallidas                        ~18 %          0 %
reinicios por caché de resolución             9 en 2 meses       0
destinos internos inalcanzables                   3             0
comprobaciones de red en la verificación          0             4
```

La lección que esta clase traslada al resto de la parte 05: tres de los cinco incidentes no parecían de red. La subida colgada parecía de la aplicación, el fallo tras reiniciar la dependencia parecía «cosa rara» y la exposición de la base de datos no parecía nada porque el cortafuegos decía que estaba cerrada. Los tres se diagnostican con una orden cada uno, y las tres órdenes caben en un guion de verificación. La red de contenedores no es difícil: es invisible, y lo que la hace manejable es tener escritas las cuatro comprobaciones antes de necesitarlas.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/065-redes-bridge-dns-interno-y-publicacion-de-puertos/
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 red-contenedores 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 red-contenedores 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 contenedor no resuelve el nombre de otro	Ambos están en la red por defecto, que no tiene resolución de nombres integrada	Crea una red propia por aplicación; nunca uses la red por defecto.
Un servicio interno es alcanzable desde internet con el cortafuegos activo	La publicación de puertos inserta una regla de traducción que se evalúa antes que el cortafuegos de usuario	No publiques lo interno; si hace falta, publica solo en bucle local y verifica desde fuera con una prueba negativa.
Las peticiones pequeñas funcionan y las grandes se quedan colgadas	La unidad máxima de transmisión del contenedor es mayor que la del camino real	Comprueba con ping -M do -s y ajusta la unidad máxima de la red; hazlo siempre al montar una red que encapsule.
El servicio falla al reiniciar su dependencia y se arregla reiniciándolo	La aplicación cacheó la dirección resuelta sin caducidad	Resuelve por nombre en cada conexión nueva, acota la caché a segundos y reintenta ante fallo de conexión.
Algunos destinos internos son inalcanzables desde los contenedores	El rango del puente colisiona con una red corporativa	Declara los rangos del motor dentro del plan de direcciones de la organización.
Se agotan las conexiones salientes hacia un destino concreto	Traducción de origen con un tope de puertos, cuarta aparición del mismo problema	Reutiliza el cliente en la aplicación; el mecanismo cambia y la corrección es siempre la misma.

■ Seguridad, ética y costo

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

■ Preguntas de comprobación

1. ¿Por qué dos contenedores en la red por defecto no se encuentran por nombre, y cuál es la corrección?
2. ¿Qué inserta exactamente -p 5432:5432 y por qué puede eludir el cortafuegos del anfitrión?
3. Describe el síntoma de una unidad máxima mal ajustada y la orden que lo confirma en un minuto.

4. ¿Por qué un servicio falla al reiniciar su base de datos y se arregla reiniciándolo?
5. ¿Qué se gana y qué se pierde con el modo de red del anfitrión, y qué explica el modo compartido sobre la parte 06?

Referencias

- Docker (2025). Networking overview — puentes, redes definidas por el usuario y resolución de nombres. <https://docs.docker.com/engine/network/>
- Docker (2025). Packet filtering and firewalls — cadenas de traducción y su relación con el cortafuegos del anfitrión. <https://docs.docker.com/engine/network/packet-filtering-firewalls/>
- Docker (2025). Network drivers and options — modo anfitrión, ninguno, compartido y opciones de unidad máxima. <https://docs.docker.com/engine/network/drivers/>
- Linux (2025). veth(4) — pares de interfaces virtuales. <https://man7.org/linux/man-pages/man4/veth.4.html>
- Cloudflare (2024). Path MTU discovery in practice — por qué falla el descubrimiento y cómo se manifiesta. <https://blog.cloudflare.com/path-mtu-discovery-in-practice/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 05 en PDF · Manual integral

Anterior	Índice	Siguiente
← 064 · Volúmenes, bind mounts y persistencia	Parte 05 · Programa	066 · Docker Compose y aplicaciones multiservicio →

Evaluación — 065 Redes bridge, DNS interno y publicación de puertos

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 red-contenedores 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.

066 — Docker Compose y aplicaciones multiservicio

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 4 Laboratorio: orchestration · Estado: EXECUTABLECORE

Propósito

Definir una aplicación de varios servicios con Compose, que resuelve muy bien un problema —que todo el equipo levante lo mismo con una orden— y resuelve muy mal otro que se le pide constantemente: ser la plataforma de producción. La clase enseña las dos cosas: cómo se escribe un fichero que funciona igual en todas las máquinas, y dónde está exactamente la frontera a partir de la cual seguir usándolo es una decisión que hay que justificar por escrito.

Resultados de aprendizaje

Al finalizar podrás:

1. Declarar dependencias que esperen a que el servicio esté listo, no a que haya arrancado.
2. Separar lo que es de la imagen de lo que es del entorno, sin reconstruir nada.
3. Publicar únicamente lo que entra desde fuera y comprobarlo.
4. Enumerar qué garantiza Compose y qué no, con la consecuencia operativa de cada carencia.
5. Traducir un fichero de Compose a un despliegue de plataforma señalando qué se conserva y qué no.

Conceptos centrales

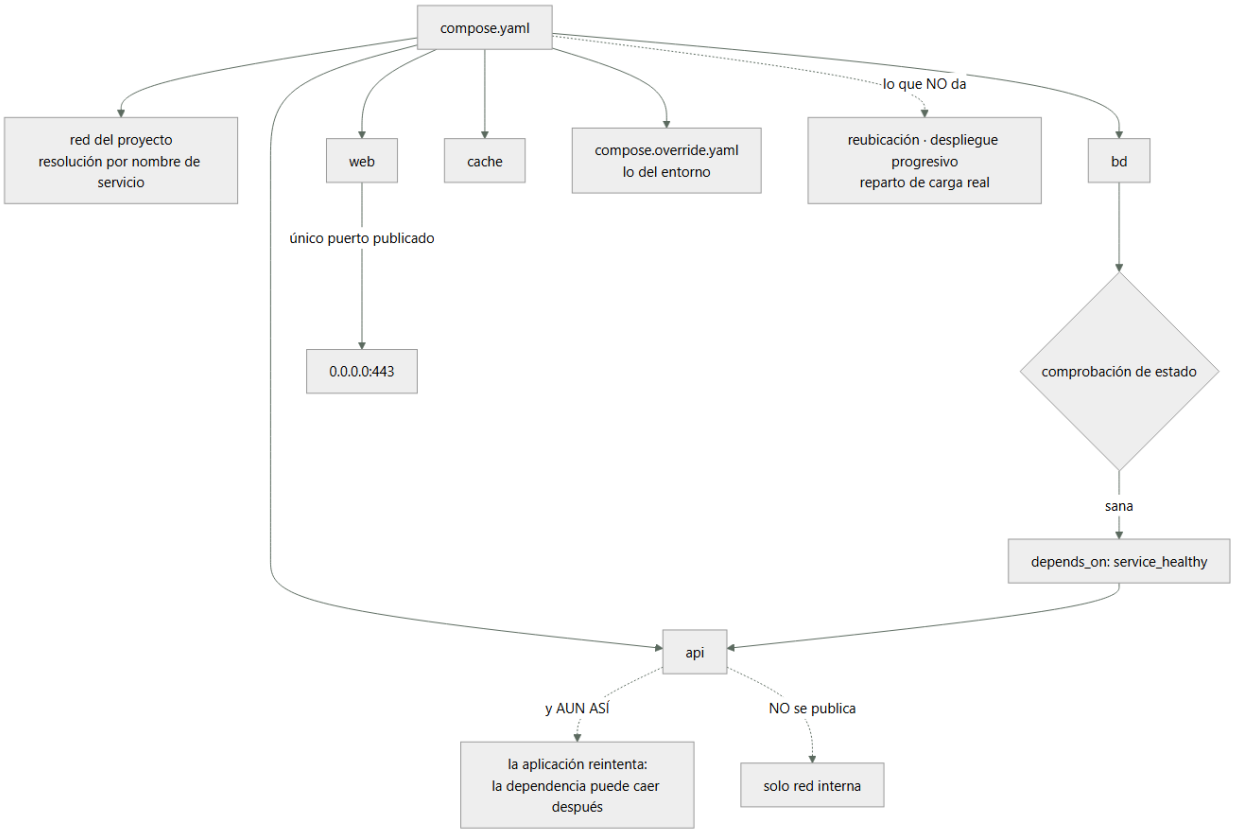
Concepto	Comprensión verificable
dependson simple	Solo espera a que el contenedor arranque, no a que el servicio esté listo. Es la causa del fallo de arranque más repetido con Compose.
condición de salud	condition: servicehealthy combinado con una comprobación de estado. Es lo único que convierte dependson en una espera útil.
perfil	Etiqueta que hace opcional a un servicio. Permite un fichero único para desarrollo, pruebas y herramientas puntuales sin levantarlo todo siempre.
superposición de ficheros	Compose combina compose.yaml con otros por capas. Es el mecanismo para que la misma definición valga en varios entornos sin duplicarla.
red del proyecto	Red propia que Compose crea, con resolución de nombres por nombre de servicio. Es una red definida por el usuario de la clase 065, con sus mismas propiedades.
política de reinicio	Instrucción para volver a arrancar un contenedor caído en la misma máquina. No es alta disponibilidad: no reubica nada si la máquina desaparece.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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. Esperar a que arranque no es esperar a que esté listo

Es el error más repetido con Compose y produce el mismo síntoma en todos los equipos: la primera vez que alguien levanta la aplicación, falla; la segunda, funciona.

```
services:
  api:
    image: registro/api@sha256:9f2c…
    depends_on: [bd]        # ← solo espera a que el CONTENEDOR arranque
```

Un motor de base de datos tarda varios segundos en aceptar conexiones después de que su contenedor exista. dependson en su forma simple no lo sabe: arranca api en cuanto bd existe, y api falla al conectar.

La corrección tiene dos mitades y hacen falta las dos.

La primera: una comprobación de estado real y una condición.

```
services:
  bd:
    image: postgres:16@sha256:1a2b…
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/bd_password
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U app -d pedidos"]
      interval: 5s
      timeout: 3s
      retries: 10
      start_period: 30s

  api:
    image: registro/api@sha256:9f2c…
    depends_on:
      bd:
        condition: service_healthy
```

El startperiod merece atención: durante ese plazo inicial, los fallos no cuentan para marcar el servicio como enfermo. Sin él, un motor que tarda cuarenta segundos en abrir se marca como enfermo antes de estar listo y el conjunto no

levanta nunca. Es el mismo retardo inicial que la reparación automática de la clase 052 necesitaba, con el mismo razonamiento.

La segunda: la aplicación reintenta igualmente.

La condición de salud resuelve el arranque y no resuelve la vida. La base de datos puede reiniciarse a las tres horas, y entonces no hay ningún dependson que ayude. Es la quinta vez que este programa llega a la misma conclusión —las conmutaciones gestionadas de las clases 048, 054 y 060, la resolución de nombres de la 065 y ahora esta—, así que ya se puede enunciar sin matices:

> Una dependencia que no está disponible es funcionamiento normal, no un caso excepcional. El cliente reintenta con retroceso o el sistema no es correcto.

Y conviene saber qué hace la comprobación de estado en Compose y qué no: marca el servicio como sano o enfermo, y no lo reinicia. Para eso está la política de reinicio, que es otra cosa y se combina con ella.

Un detalle práctico que ahorra tiempo en canalizaciones: --wait bloquea hasta que todos los servicios con comprobación estén sanos, lo que convierte el arranque en algo verificable:

```
$ docker compose up -d --wait --wait-timeout 120
$ echo $?      # 0 solo si todo llegó a sano
```

Eso es exactamente lo que hace falta en una prueba de integración: no «arrancó», sino «está listo».

2. Lo de la imagen, lo del entorno y lo que no debe estar

La clase 062 estableció el principio: se construye una vez y se promueve; lo que cambia entre entornos es la configuración inyectada al ejecutar. Compose es donde ese principio se materializa en local.

La estructura que lo consigue:

```
# compose.yaml — lo común, válido en cualquier entorno
services:
  api:
    image: registro/api@sha256:9f2c…
    environment:
      LOG_LEVEL: ${LOG_LEVEL:-info}
      BD_HOST: bd
    secrets: [bd_password]
    read_only: true
    tmpfs: [/tmp:size=64m,noexec,nosuid]
    user: "10001:10001"

secrets:
  bd_password:
    file: ./.secretos/bd_password
```

```
# compose.override.yaml — solo desarrollo; Compose lo aplica encima solo
services:
  api:
    build: .
    environment:
      LOG_LEVEL: debug
    develop:
      watch:
        - action: sync
          path: ./src
          target: /app/src
```

Dos cosas que hace bien esa separación. La primera: el fichero base no construye, usa una huella, así que en pruebas y en preproducción se ejecuta el mismo artefacto que en producción. La segunda: la sincronización de código para desarrollo está en la capa que no viaja, en vez de resolverse con un montaje del anfitrión que después alguien copia a un entorno real.

Y los perfiles evitan el fichero que levanta doce servicios cuando hacen falta tres:

```
services:
  panel-bd:
    image: adminer
    profiles: [herramientas]
```

```
$ docker compose up -d                        # sin panel-bd
```

```
$ docker compose --profile herramientas up -d # con él
```

Sobre los secretos, la regla es la de las clases 061 y 062, y Compose tiene un mecanismo propio que evita la vía cómoda:

```
mal   environment: BD_PASSWORD=…       queda en la configuración del contenedor,
                                        legible con inspect por cualquiera
      un .env con credenciales reales   acaba en el repositorio

bien  secrets: montados como fichero    fuera de la configuración,
                                        y en producción los sustituye
                                        el gestor de la plataforma
```

Y una advertencia sobre .env que se paga en casi todos los equipos: Compose lo lee automáticamente y su nombre invita a poner credenciales. Debe estar en .gitignore desde el primer commit, contener solo valores no sensibles, y tener un \.env.example versionado que documente las claves sin los valores.

Y la red, que es la de la clase 065 con otro nombre: Compose crea una red propia del proyecto, con resolución por nombre de servicio, así que BDHOST: bd funciona sin ninguna dirección. De ahí se sigue la regla de publicación:

```
services:
  web:
    ports: ["127.0.0.1:8080:8080"]   # lo único que entra desde fuera
  api: {}                             # sin ports: solo red interna
  bd:  {}                             # sin ports
```

Publicar la base de datos «para poder conectarme con mi cliente» es exactamente el incidente de la clase 065. Si hace falta, se publica en bucle local y se documenta.

3. Dónde está la frontera, dicho sin rodeos

Compose se acaba usando en producción por una progresión razonable: funciona, el equipo lo conoce y montar otra cosa cuesta. Conviene tener la lista de lo que no hace, porque cada carencia tiene una consecuencia operativa concreta.

```
no reubica              si la máquina cae, la aplicación no se mueve a ninguna parte
                        `restart: always` solo reinicia en LA MISMA máquina
no despliega progresivamente
                        `up -d` para y arranca: hay corte, de segundos a minutos
no reparte carga        `--scale 3` da tres contenedores y la resolución interna
                        devuelve las tres direcciones; el reparto depende del cliente
no gestiona secretos    los lee de ficheros locales
no controla capacidad   nada impide que dos aplicaciones del mismo nodo compitan
no tiene salud del conjunto
                        cada servicio tiene la suya; nadie decide si el conjunto sirve
```

La tercera línea produce un fallo instructivo que enlaza con la clase 065: al escalar, la resolución interna devuelve varias direcciones, y un cliente que resuelve una vez y guarda la primera envía todo el tráfico a un solo contenedor. Los otros dos existen, están sanos y no reciben nada. Es un reparto de carga solo si el cliente colabora.

Y dicho todo eso, hay un caso en el que usar Compose en producción es una decisión defendible, y conviene reconocerlo en lugar de fingir que no existe:

```
una sola máquina, sin requisito de alta disponibilidad
corte de despliegue aceptado y documentado en el acuerdo de servicio
copias de seguridad y restauración probadas (clase 064)
vigilancia externa que detecte que la máquina no responde
y la decisión escrita, con la condición que obligaría a revisarla
```

Eso describe muchas herramientas internas, entornos de demostración y sistemas de bajo tráfico. Lo que no es defendible es llegar ahí sin haberlo decidido. La señal de alarma es concreta: si el acuerdo de servicio promete algo que la lista de arriba no puede dar, hay una discrepancia que alguien va a descubrir durante un incidente.

Y la traducción a una plataforma real, para cuando llegue el momento, con lo que se conserva y lo que no:

```
se conserva casi tal cual
  la imagen por huella
  las variables de entorno
  la comprobación de estado (cambia la sintaxis, no la idea)
  los nombres de servicio como nombres de red
  el usuario no privilegiado y la raíz de solo lectura

hay que rehacerlo
```

```
la publicación de puertos → objeto de servicio y entrada
los volúmenes → reclamaciones de almacenamiento (clase 064)
los secretos → gestor de la plataforma (clases 046, 058)
`restart: always` → el orquestador lo hace por diseño
`depends_on` → NO existe: cada servicio arranca cuando puede
```

La última línea es la más importante de la traducción y la que más sorprende: en una plataforma real no hay orden de arranque garantizado. La condición de salud de Compose se convierte en «el cliente reintenta», que ya era obligatorio. Un equipo que se apoyó en dependson en vez de en el reintento descubre en la migración que su aplicación nunca supo arrancar sin ayuda.

4. Un fichero que sirve de contrato del entorno

El valor real de Compose no es técnico: es que el entorno de desarrollo deja de ser conocimiento tácito. Un fichero versionado con las dependencias exactas sustituye a una página de instrucciones que envejece.

El fichero completo de una aplicación pequeña, con todo lo de las clases anteriores aplicado:

```
name: cloudshop

services:
  web:
    image: registro/web@sha256:3c4d…
    ports: ["127.0.0.1:8080:8080"]
    environment:
      API_URL: http://api:9000
    depends_on:
      api: {condition: service_healthy}
    read_only: true
    tmpfs: ["/tmp:size=32m,noexec,nosuid"]
    user: "10001:10001"
    cap_drop: [ALL]
    security_opt: ["no-new-privileges:true"]
    healthcheck:
      test: ["CMD", "/bin/web", "-healthcheck"]
      interval: 10s
      start_period: 20s
    deploy:
      resources:
        limits: {cpus: "1.0", memory: 512M}

  api:
    image: registro/api@sha256:9f2c…
    environment:
      BD_HOST: bd
      CACHE_HOST: cache
      GOMAXPROCS: "2"                    # clase 063
    secrets: [bd_password]
    depends_on:
      bd:    {condition: service_healthy}
      cache: {condition: service_started}
    read_only: true
    tmpfs: ["/tmp:size=64m,noexec,nosuid"]
    user: "10001:10001"
    cap_drop: [ALL]
    security_opt: ["no-new-privileges:true"]
    healthcheck:
      test: ["CMD", "/bin/api", "-readyz"]
      interval: 10s
      start_period: 30s
    deploy:
      resources:
        limits: {cpus: "2.0", memory: 1G}

  bd:
    image: postgres:16@sha256:1a2b…
    environment:
      POSTGRES_DB: pedidos
      POSTGRES_USER: app
      POSTGRES_PASSWORD_FILE: /run/secrets/bd_password
    secrets: [bd_password]
    volumes: ["datos-bd:/var/lib/postgresql/data"]
```

```
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U app -d pedidos"]
      interval: 5s
      start_period: 30s

  cache:
    image: redis:7-alpine@sha256:5e6f…
    command: ["redis-server", "--maxmemory", "256mb", "--maxmemory-policy", "allkeys-lru"]

volumes:
  datos-bd:
    labels: {responsable: "equipo-pedidos"}

secrets:
  bd_password:
    file: ./.secretos/bd_password
```

Cada bloque viene de una clase anterior y merece señalarse, porque el fichero es donde todo se junta:

```
imagen por huella                       clase 061
GOMAXPROCS acorde al límite             clase 063
límites de CPU y memoria siempre        clase 063
volumen con etiqueta de responsable     clase 064
un solo puerto publicado, en bucle local clase 065
raíz de solo lectura y montaje en memoria acotado  clases 064, 069
sin capacidades y sin elevación         clase 069
comprobación de estado con retardo inicial  clases 052, 068
política de expulsión de la caché       clase 042
```

Y dos comprobaciones que conviene tener automatizadas sobre el propio fichero, porque detectan casi todos los errores de esta clase:

```
# la definición es válida y resuelve todas las variables
$ docker compose config --quiet

# nada publicado hacia todas las interfaces salvo el frontal
$ docker compose config --format json \
  | jq -r '.services | to_entries[] | select(.value.ports)
           | "\(.key): \(.value.ports[].published)"'
```

La primera detecta variables sin definir, que es como se cuela una configuración vacía en producción. La segunda es la prueba negativa de la clase 065, ejecutada sobre la declaración en vez de sobre el sistema en marcha — más barata y más temprana.

5. Operar con Compose sin sorpresas

Cuatro órdenes concentran casi todos los accidentes, y merece la pena conocer exactamente qué hacen.

```
docker compose up -d        crea o recrea lo que cambió; PARA y arranca: hay corte
docker compose down         para y elimina contenedores y redes
docker compose down -v      …y BORRA LOS VOLÚMENES  ← el destructor
docker compose restart      reinicia sin releer la configuración
```

La tercera es la que apareció en la clase 064 y conviene repetir porque está en muchos guiones de «empezar de cero». En una máquina compartida, un down -v se lleva los datos de todo el proyecto.

La cuarta esconde una trampa distinta: restart no relee el fichero. Un cambio de variable de entorno seguido de restart no tiene ningún efecto, y el equipo concluye que la variable no funciona. Lo que aplica cambios es up -d.

Y una nota sobre el corte: up -d recrea el contenedor cambiado, con lo que hay una ventana sin servicio. Se puede reducir, no eliminar:

```
$ docker compose up -d --no-deps --wait api
```

--no-deps evita recrear lo que no cambió y --wait espera a que el nuevo esté sano. Sigue habiendo corte porque el viejo se detiene antes de que el nuevo levante, y esa es precisamente la carencia estructural: el despliegue sin corte necesita un orquestador que sepa tener dos versiones a la vez, que es lo que la parte 06 introduce.

Para diagnosticar, cuatro órdenes que responden las preguntas habituales:

```
$ docker compose ps --format 'table {{.Service}}\t{{.Status}}\t{{.Health}}'
$ docker compose logs -f --tail 100 api
$ docker compose exec api sh -c 'cat /sys/fs/cgroup/memory.max'   # clase 063
$ docker compose events --json | jq -r '"\(.time) \(.service) \(.action)"'
```

La última es poco conocida y muy útil: muestra el flujo de sucesos —creaciones, muertes, reinicios— y responde de un vistazo a «qué se está reiniciando en bucle», que en un fichero con ocho servicios cuesta ver de otra forma.

Y el cierre que enlaza con el resto del programa: un fichero de Compose bien escrito es, de hecho, la especificación de lo que hay que pedirle a cualquier plataforma. Contiene la imagen exacta, los límites, la comprobación de salud, las dependencias, los secretos y qué se publica. Migrar a una plataforma consiste en traducir ese fichero, y las traducciones difíciles —volúmenes, publicación, arranque sin orden— son exactamente las cuatro fugas que la clase 060 predijo. Que la traducción sea mecánica en todo lo demás es la confirmación de la primera mitad de la hipótesis.

Ejemplo trabajado

CloudShop define su aplicación en Compose. Funciona el primer día, se convierte en el entorno de desarrollo de todo el equipo, y acaba en una máquina de producción sin que nadie lo decidiera. Cinco incidentes ordenan las dos cosas.

Incidente 1 — «la primera vez nunca levanta».

```
api | Error: connect ECONNREFUSED 172.19.0.3:5432
api exited with code 1
```

Y funcionaba al segundo intento, así que la instrucción de puesta en marcha del equipo decía literalmente «si falla, vuelve a ejecutarlo».

```
depends_on                              forma simple     condition: service_healthy
comprobación de estado de la base           ninguna        pg_isready con
                                                          start_period de 30 s
reintento en la aplicación                    no        con retroceso, 5 intentos
arranques fallidos en frío                  100 %             0 %
instrucción "vuelve a ejecutarlo"              sí            eliminada
```

Las dos correcciones a la vez: la condición resolvió el arranque y el reintento resolvió los reinicios de la base de datos a mitad del día, que la condición no cubre.

Incidente 2 — un .env con credenciales de producción.

```
$ git log --oneline -S 'BD_PASSWORD' -- .env | wc -l
1
$ git show <commit>:.env | grep BD_PASSWORD
BD_PASSWORD=Pr0d-2026-…
```

El fichero se había versionado once meses atrás. Y algo peor: alguien había levantado el entorno local con él, así que su máquina se conectó a la base de datos real durante una tarde de pruebas.

```
.env en el repositorio                        sí               no
secretos                               variables de entorno   ficheros montados
                                                              como secretos
.env.example versionado                       no               sí
credencial rotada                              —           la misma tarde
escaneo de secretos en el repositorio      ninguno       en cada pull request
```

Incidente 3 — down -v en la máquina compartida.

Un guion de «reiniciar el entorno» ejecutaba docker compose down -v. En la máquina de integración compartida, se llevó los datos de prueba de tres equipos, incluida una base de datos con casos reproducidos durante semanas.

```
orden en el guion                       down -v          down (sin -v)
reinicio de datos                       implícito        objetivo aparte,
                                                          con confirmación
copia de la base de integración          ninguna         diaria, restaurada
                                                          y verificada una vez al mes
etiquetas de responsable en volúmenes       no               sí
```

Incidente 4 — producción en Compose, descubierta durante un corte.

La máquina de producción se reinició por mantenimiento del proveedor. Al volver, la aplicación estaba a medias:

```
web    up      (restart: always)
api    up      (restart: always)
bd     exited  (sin política de reinicio)
cache  up      (restart: always)
```

Y el análisis posterior encontró más de lo esperado:

```
reubicación si la máquina cae            ninguna
corte en cada despliegue                 38 s medidos
```

```
acuerdo de servicio prometido            99,9 %
disponibilidad alcanzable con este montaje   ~99,2 % con suerte
copias de seguridad                      existían, nunca restauradas
vigilancia externa                       ninguna
```

La discrepancia entre lo prometido y lo alcanzable llevaba dieciocho meses sin que nadie la hubiera escrito. Se tomaron dos decisiones, y lo importante es que se tomaron:

```
servicios internos de baja criticidad   se quedan en Compose, con la decisión
                                        documentada y el acuerdo corregido
tienda y api                            migran a plataforma en la parte 06
todos                                   política de reinicio uniforme,
                                        vigilancia externa y restauración probada
```

Incidente 5 — tres réplicas y una que recibía todo.

```
$ docker compose up -d --scale api=3
$ docker compose exec web sh -c 'for i in $(seq 20); do curl -s api:9000/quien; done' | sort | uniq -c
     20 api-1
```

Las tres estaban sanas. El cliente resolvía el nombre una vez, guardaba la primera dirección y la reutilizaba — el mismo problema de caché de resolución de la clase 065, con otra consecuencia.

```
réplicas                                    3                3
réplicas que recibían tráfico               1                3
resolución en el cliente            una vez, cacheada    por conexión, con TTL corto
reparto real                             ninguno        proporcional
```

Y la conclusión que se anotó: escalar con Compose reparte contenedores, no tráfico. El reparto depende del cliente, y por eso deja de ser suficiente en cuanto el servicio importa.

Resumen:

```
arranques en frío fallidos                  100 %            0 %
secretos en el repositorio                     1              0
volúmenes borrados por un guion                3              0
puertos publicados hacia todas las interfaces  4              1
réplicas que reciben tráfico                1 de 3         3 de 3
servicios en producción sobre Compose          4              2, documentados
corte por despliegue                        38 s        38 s (aceptado por escrito)
```

La lección que esta clase traslada al resto de la parte 05: Compose resuelve bien el problema del entorno reproducible y su fichero acaba siendo la especificación de lo que hay que pedirle a cualquier plataforma. Los cinco incidentes se reparten entre los dos usos: tres eran de higiene del fichero y dos eran de haber cruzado la frontera sin decidirlo. La frontera no es un problema técnico —Compose no promete nada que incumpla— sino de expectativa: el acuerdo de servicio prometía una disponibilidad que ninguna configuración de una sola máquina puede dar, y eso no se descubre hasta que la máquina se reinicia.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/066-docker-compose-y-aplicaciones-multiservicio/lab.py
```

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

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

Evidencia esperada

El artefacto principal es servicios coordinados con health checks y apagado limpio. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye stack-compose 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 aplicación falla la primera vez que se levanta y funciona al reintentar	dependson simple espera a que el contenedor arranque, no a que el servicio acepte conexiones	Añade comprobación de estado con startperiod y condition: servicehealthy, y reintenta también en la aplicación.
Un cambio de variable de entorno no tiene efecto	Se usó restart, que no relee la configuración	Aplica cambios con up -d; restart solo reinicia el proceso.
Un guion de reinicio borra datos	down -v elimina los volúmenes del proyecto	Nunca pongas -v en un guion rutinario; separa el reinicio de datos en un objetivo explícito con confirmación.
Credenciales reales en el repositorio	.env se lee automáticamente y su nombre invita a poner secretos	Ignóralo desde el primer commit, versiona un .env.example y usa secretos montados como fichero.
Se escala a tres réplicas y una recibe todo el tráfico	El reparto depende del cliente y este cachea la primera dirección resuelta	Resuelve por conexión con caché corta; para reparto real hace falta un balanceador o un orquestador.
Tras un reinicio de la máquina, la aplicación queda a medias	Las políticas de reinicio eran distintas por servicio y no hay reubicación	Unifica la política, añade vigilancia externa y decide por escrito si esa máquina puede sostener el acuerdo de servicio.

■ 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 espera exactamente dependson en su forma simple y qué hace falta para que espere a que el servicio esté listo?
2. ¿Por qué la condición de salud no elimina la necesidad de reintentar en la aplicación?
3. ¿Qué se conserva y qué hay que rehacer al traducir un fichero de Compose a una plataforma real?
4. Enumera cuatro cosas que Compose no hace y la consecuencia operativa de cada una.
5. ¿Por qué escalar con Compose no reparte tráfico, y con qué clase anterior se relaciona el motivo?

Referencias

- Docker (2025). Compose file reference — servicios, dependencias, perfiles y secretos. <https://docs.docker.com/reference/compose-file/>
- Docker (2025). Control startup order — dependson, condiciones y comprobaciones de estado. <https://docs.docker.com/compose/how-tos/startup-order/>
- Docker (2025). Merge Compose files — superposición, override y variables de entorno. <https://docs.docker.com/compose/how-tos/multiple-compose-files/merge/>
- Docker (2025). Compose Develop specification — sincronización de código para desarrollo. <https://docs.docker.com/compose/how-tos/file-watch/>
- Docker (2025). Use secrets in Compose — secretos como ficheros en vez de variables de entorno. <https://docs.docker.com/compose/how-tos/use-secrets/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 05 en PDF · Manual integral

Anterior	Índice	Siguiente
← 065 · Redes bridge, DNS interno y publicación de puertos	Parte 05 · Programa	067 · Registros, SBOM, firma y procedencia de imágenes →

Evaluación — 066 Docker Compose y aplicaciones multiservicio

Preguntas

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

Reto verificable

Entrega stack-compose 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.

067 — Registros, SBOM, firma y procedencia de imágenes

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 4 Laboratorio: supply-chain · Estado: EXECUTABLECORE

Propósito

Responder tres preguntas sobre una imagen que está a punto de ejecutarse en producción: qué lleva dentro, quién dice que es suya y cómo se construyó. Cada una tiene un artefacto —inventario de componentes, firma y declaración de procedencia— que se guarda junto a la imagen en el registro. Y la clase insiste en la disciplina que este programa ya ha aplicado tres veces: firmar sin verificar es teatro, exactamente igual que una directiva sin prueba negativa o una cola de fallidos sin un mensaje envenenado.

Resultados de aprendizaje

Al finalizar podrás:

1. Generar un inventario de componentes y usarlo para responder «¿nos afecta?» en minutos en vez de en días.
2. Firmar imágenes sin gestionar ninguna clave y verificar la firma en el momento de desplegar.
3. Exigir procedencia para que solo se ejecute lo que salió de la canalización declarada.
4. Definir una puerta de vulnerabilidades que el equipo no acabe desactivando.
5. Separar las identidades de publicación y de descarga en el registro, con su prueba negativa.

Conceptos centrales

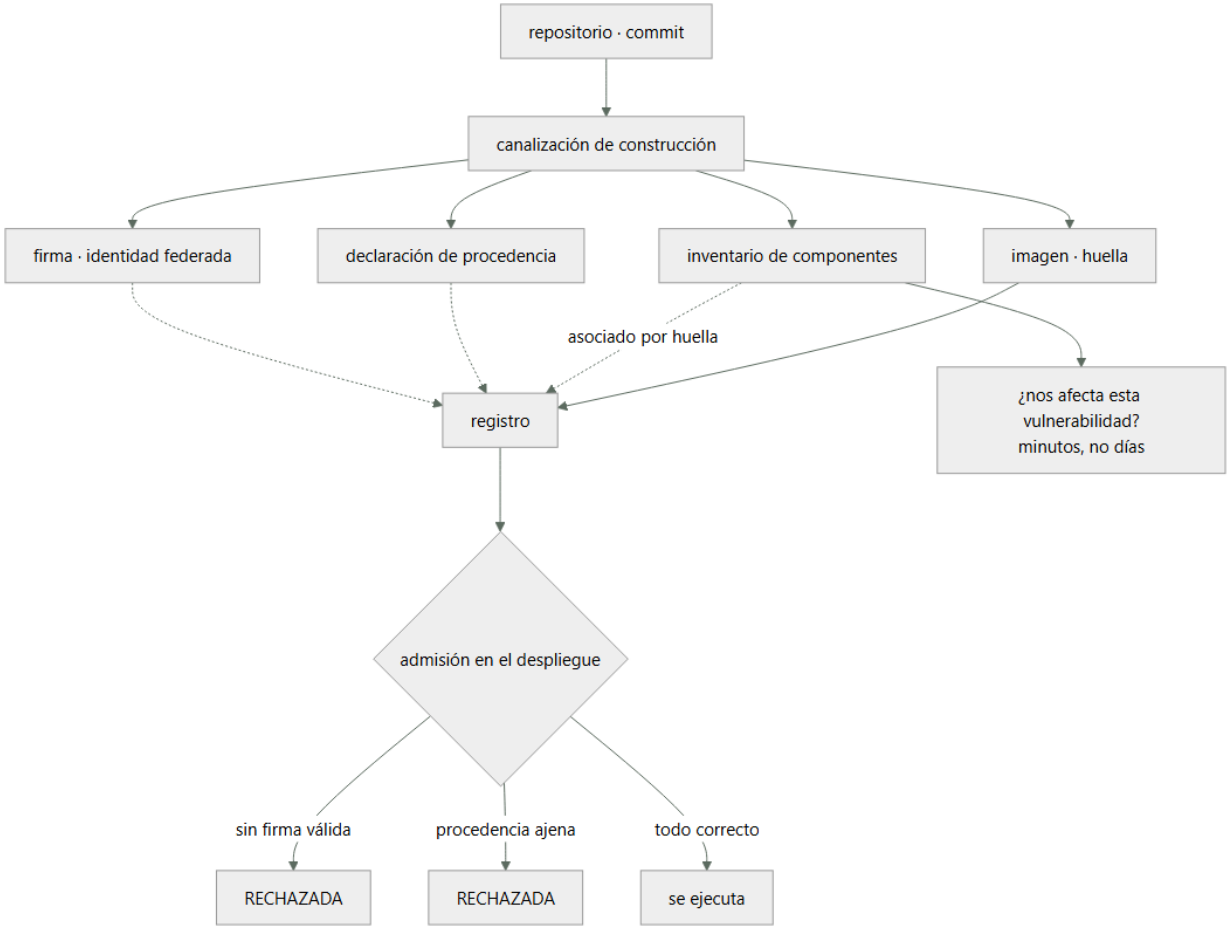
Concepto	Comprensión verificable
inventario de componentes	Lista de todo lo que contiene una imagen con sus versiones. Su valor no es documental: es responder en minutos si una vulnerabilidad recién publicada afecta a algo tuyo.
firma sin claves	Firma emitida a partir de una identidad federada, sin material criptográfico que custodiar. Es la aplicación directa de la federación de las clases 026, 038 y 050.
declaración de procedencia	Documento verificable que dice de qué repositorio, qué commit y qué constructor salió una imagen. Impide que algo publicado desde un portátil pase por artefacto oficial.
artefacto referente	Objeto que el registro asocia a una imagen por su huella. Es lo que permite guardar inventario, firma y procedencia junto a la imagen sin cambiarla.
puerta de vulnerabilidades	Regla que detiene la publicación. Si bloquea por todo, el equipo la desactiva; si bloquea por lo crítico y corregible, se respeta.
excepción con caducidad	Permiso temporal para publicar algo que incumple, con motivo, responsable y fecha. Es la misma disciplina de las clases 046 y 049 aplicada a la cadena de suministro.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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

■ Flujo de razonamiento



Desarrollo

1. Tres preguntas, tres artefactos

La clase 061 dejó que una imagen se identifica por su huella y que la huella es verificable. Eso responde a «¿es exactamente este contenido?» y no responde a nada más. Las tres preguntas que quedan, con el artefacto que contesta cada una:

```
¿qué lleva dentro?        inventario de componentes
¿quién dice que es suya?  firma
¿cómo se construyó?       declaración de procedencia
```

Y los tres se guardan en el registro, asociados a la huella de la imagen, sin modificarla. Esa es la parte del estándar de distribución que la clase 061 mencionó y que aquí se cobra: un registro puede alojar objetos que apuntan a otro objeto, y un cliente puede pedir «todo lo que hace referencia a esta huella».

```
$ cosign tree registro/tienda@sha256:9f2c4a1b…
■ Supply Chain Security Related artifacts
■■■ ■ Signatures
■■■ ■ SBOMs
■■■ ■ Attestations
```

La propiedad importante es que la asociación es por huella, no por etiqueta. Una firma vale para un contenido exacto; si la etiqueta se mueve, la firma no la sigue. Es la misma razón por la que se despliega por huella, con una consecuencia adicional: sin desplegar por huella, la verificación de firma no significa gran cosa.

El orden en el que conviene adoptarlos no es el orden en que se enumeran, y merece decirlo porque adoptar el tercero primero es un error común:

```
1. inventario   valor inmediato, sin cambiar el despliegue,
                y responde a la pregunta que llega de urgencia
2. firma + verificación   valor real solo si se verifica
3. procedencia + política  valor máximo, y exige que 1 y 2 funcionen
```

Y una advertencia que se aplica a los tres: generarlos sin usarlos no aporta nada. Un inventario que nadie consulta, una firma que nadie verifica y una procedencia que ninguna política exige son tres artefactos que ocupan espacio en el registro y producen la sensación de estar protegido. Es la misma familia de fallos que la clase 060 identificó como la más cara — un mecanismo que parece estar haciendo algo y no lo está — y aquí aparece por quinta vez.

2. El inventario: responder «¿nos afecta?» en minutos

Cuando se publica una vulnerabilidad grave en una biblioteca muy usada, la pregunta llega a la vez desde dirección, desde clientes y desde seguridad, y tiene una forma concreta:

```
¿qué servicios nuestros incluyen esa biblioteca, en qué versión,
 y desde cuándo están desplegados?
```

Sin inventario, la respuesta se construye a mano: buscar en repositorios, revisar ficheros de dependencias, comprobar qué versión se desplegó de verdad. En una organización mediana eso son días, y mientras tanto no se puede decir nada.

Con inventario, la respuesta es una consulta:

```
$ syft registro/tienda@sha256:9f2c… -o spdx-json > sbom.json
$ cosign attach sbom --sbom sbom.json registro/tienda@sha256:9f2c…

# y cuando llega la pregunta:
$ for img in $(kubectl get pods -A -o jsonpath='{..imageID}' | tr ' ' '\n' | sort -u); do
    cosign download sbom "$img" 2>/dev/null \
      | jq -r --arg i "$img" '.packages[] | select(.name=="biblioteca-x") | "\($i) \(.versionInfo)"'
  done
```

Dos formatos conviven —uno más orientado a licencias y cumplimiento, otro más orientado a seguridad— y ambos sirven. Lo que importa es elegir uno y generarlo siempre.

Y hay una diferencia entre generarlo durante la construcción y escaneando la imagen terminada que conviene conocer, porque ninguna de las dos es completa sola:

```
durante la construcción
  ve el grafo de dependencias real, con las transitivas y sus versiones exactas
  no ve lo que se instaló por otros medios

escaneando la imagen
  ve los paquetes del sistema y lo que hay en el sistema de ficheros
  no distingue una dependencia de una copia suelta ni ve lo eliminado
```

La práctica sensata es generar el inventario en la construcción, donde la información es mejor, y escanear la imagen publicada como comprobación independiente. Cuando los dos discrepan, la discrepancia es en sí misma un hallazgo: significa que algo entró en la imagen por un camino que la construcción no declara.

Y el inventario tiene un segundo uso que se descubre tarde y compensa el esfuerzo: la comparación entre versiones. Un cambio de dependencias inesperado en un despliegue rutinario se ve de un vistazo:

```
$ diff <(cosign download sbom $ANTERIOR | jq -r '.packages[].name' | sort) \
       <(cosign download sbom $NUEVA    | jq -r '.packages[].name' | sort)
> paquete-que-nadie-pidio
```

Esa línea de más es la señal de una dependencia transitiva nueva, que es como entran la mayoría de los problemas de cadena de suministro.

3. Firmar sin claves, y verificar de verdad

La firma responde a quién publicó una imagen, y el modelo moderno evita el problema que este programa ha señalado tres veces: no hay ninguna clave que custodiar.

```
$ cosign sign --yes registro/tienda@sha256:9f2c4a1b…
```

Lo que ocurre por debajo es la federación de las clases 026, 038 y 050 aplicada a la firma: la canalización demuestra su identidad ante un servicio que emite un certificado de corta duración, se firma con él, y la firma queda registrada en un registro público de transparencia. No hay clave privada que rotar, filtrar ni perder.

La firma vincula tres cosas: la huella de la imagen, la identidad que firmó y el momento. Y la verificación comprueba la identidad, no solo la validez:

```
$ cosign verify registro/tienda@sha256:9f2c4a1b… \
```

```
    --certificate-identity-regexp '^https://github\.com/cloudshop/tienda/\.github/workflows/publicar\.
yml@refs/heads/main$' \
    --certificate-oidc-issuer https://token.actions.githubusercontent.com
```

Esa expresión es la pieza crítica y es exactamente el mismo campo que en las tres federaciones anteriores: sin acotar la identidad, cualquiera con una identidad válida puede firmar y la verificación pasa. Verificar «que está firmada» sin comprobar por quién es equivalente a aceptar cualquier pasaporte sin mirar el nombre. Cuarta vez que este programa señala el mismo campo.

Y aquí está el punto que da sentido a toda la clase. Firmar es la parte fácil y la que se adopta primero; verificar es la que aporta, y se hace en el momento de admitir el despliegue:

```
# política de admisión: solo imágenes firmadas por nuestra canalización
apiVersion: policy.sigstore.dev/v1beta1
kind: ClusterImagePolicy
spec:
  images:
    - glob: "registro.interno/cloudshop/**"
  authorities:
    - keyless:
        url: https://fulcio.sigstore.dev
        identities:
          - issuer: https://token.actions.githubusercontent.com
            subjectRegExp: "^https://github\\.com/cloudshop/.*/\\.github/workflows/publicar\\.
yml@refs/heads/main$"
```

Sin esa política, la firma es un adorno. Y la prueba negativa correspondiente es la misma que este programa ha exigido en cada control:

```
$ kubectl run prueba --image=registro.interno/cloudshop/tienda@sha256:sinfirma…
Error from server: admission webhook denied the request:
  no matching signatures                                                ✓
```

La procedencia va un paso más allá y responde cómo se construyó:

```
$ cosign verify-attestation --type slsaprovenance registro/tienda@sha256:9f2c… \
    --certificate-identity-regexp '…' --certificate-oidc-issuer '…' \
  | jq -r '.payload' | base64 -d \
  | jq '{repo: .predicate.invocation.configSource.uri,
         commit: .predicate.invocation.configSource.digest.sha1,
         constructor: .predicate.builder.id}'
{
  "repo": "git+https://github.com/cloudshop/tienda@refs/heads/main",
  "commit": "a1b2c3d4e5f6…",
  "constructor": "https://github.com/actions/runner"
}
```

Eso cierra la cadena completa que la clase 061 empezó: del contenedor en ejecución al commit exacto, de forma verificable y sin preguntar a nadie. Y lo que impide es concreto: que una imagen construida en el portátil de alguien —con dependencias distintas, con código sin revisar— se despliegue como si fuera oficial. Sin procedencia exigida, nada lo distingue: las dos imágenes tienen huella válida y las dos pueden estar firmadas por una identidad de la organización.

4. Una puerta de vulnerabilidades que el equipo no desactive

El escaneo de vulnerabilidades falla casi siempre por el mismo motivo: se configura para bloquear por todo, produce cientos de hallazgos, el equipo no puede avanzar y alguien lo desactiva. Después figura como implantado.

```
$ trivy image --severity CRITICAL,HIGH registro/tienda@sha256:9f2c…
Total: 412 (HIGH: 371, CRITICAL: 41)
```

Cuatrocientos doce hallazgos en una imagen recién construida no significan cuatrocientos doce problemas. Significan tres cosas mezcladas:

```
1. vulnerabilidades en paquetes del sistema base, la mayoría
2. vulnerabilidades sin corrección disponible todavía
3. vulnerabilidades en código que la aplicación nunca ejecuta
```

La puerta que funciona distingue esas categorías:

```
$ trivy image --exit-code 1 \
    --severity CRITICAL,HIGH \
    --ignore-unfixed \
```

```
    --ignorefile .trivyignore \
    registro/tienda@sha256:9f2c…
```

--ignore-unfixed es la opción que cambia el resultado: bloquear por algo que no se puede arreglar solo enseña a saltarse la puerta. Lo que queda es lo crítico y corregible, que es accionable por definición.

Y la palanca con más rendimiento no es corregir hallazgos uno a uno: es actualizar la base. La mayoría de los hallazgos vienen de ahí, así que una cadencia de actualización de la imagen base resuelve en bloque lo que de otro modo se persigue individualmente:

```
sin cadencia de actualización de base    412 hallazgos, creciendo cada mes
con actualización mensual de la base      41 tras la primera, 3-8 en régimen
```

Eso conecta con la decisión de la clase 062 de fijar la base por huella: fijarla congela también sus vulnerabilidades, así que fijar exige un proceso que proponga la actualización. Un robot que abre un cambio con la huella nueva y deja que la canalización decida es suficiente, y es lo que convierte la fijación en algo sostenible en vez de en deuda.

Para lo que no se puede corregir ahora, la excepción con caducidad, que es la misma disciplina de las clases 046 y 049:

```
# .trivyignore
# CVE-2026-1234 — biblioteca X, sin corrección; el código afectado no se
# alcanza desde ninguna ruta de la aplicación.
# Responsable: equipo-tienda   Revisar: 2026-10-01
CVE-2026-1234 exp:2026-10-01
```

Tres campos obligatorios: por qué, quién y hasta cuándo. Una lista de exclusiones sin fechas es una lista que crece para siempre y que nadie vuelve a mirar.

Y una precisión honesta sobre lo que un escáner puede y no puede decir: informa de que una versión vulnerable está presente, no de que sea alcanzable desde tu aplicación. La diferencia es grande —muchos hallazgos afectan a rutas de código que nunca se ejecutan— y existen formatos para declarar esa evaluación de forma verificable. Sin ellos, la alternativa es la lista de exclusiones documentada, que es peor pero funciona si se revisa.

5. El registro como control, no como almacén

El registro es donde todo lo anterior vive, y merece tratarse como un componente de seguridad y no como un disco.

Identidades separadas para publicar y para descargar. Es el error de configuración más frecuente:

```
identidad de la canalización     puede PUBLICAR en su repositorio, y solo en él
identidad de los nodos           puede DESCARGAR, y nada más
personas                         descargan; publicar es de la canalización
```

Si los nodos pueden publicar, un contenedor comprometido puede sustituir la imagen del siguiente despliegue. Y si las personas pueden publicar, la procedencia deja de significar nada porque siempre habrá un camino alternativo. La prueba negativa:

```
$ docker push registro.interno/cloudshop/tienda:prueba   # con credenciales de nodo
denied: requested access to the resource is denied                          ✓
```

Etiquetas inmutables y retención, de la clase 061, más una precisión sobre el borrado: una política de retención por antigüedad puede eliminar la huella que producción está usando si ese despliegue lleva meses sin cambiar. La retención se define por referencias, no solo por fecha:

```
conservar siempre   toda huella referenciada por algún despliegue activo
conservar N meses   el resto
nunca borrar        lo que tenga una firma o una procedencia asociadas
                    y siga referenciado
```

Espejo de lo externo, que la clase 061 ya justificó por los límites de descarga y que aquí tiene una segunda razón más importante: una imagen base externa copiada al registro propio es un artefacto controlado, con su inventario y su escaneo, en vez de una descarga que cambia cuando el tercero decide.

Y el flujo completo de la cadena, que es el entregable de esta clase:

```
en la construcción
  1. construir con base fijada por huella          (062)
  2. generar el inventario de componentes
  3. escanear: bloquear crítico y corregible
  4. publicar por huella
  5. firmar con identidad federada
```

```
  6. adjuntar inventario y procedencia

en el despliegue
  7. resolver a huella y desplegar por huella        (061)
  8. admisión: verificar firma e identidad del firmante
  9. admisión: verificar procedencia del repositorio esperado

en operación
 10. reescanear las huellas EN PRODUCCIÓN periódicamente
     una imagen no cambia; las vulnerabilidades conocidas sí
```

El paso 10 se olvida siempre y es el que detecta el problema real: una imagen escaneada limpia hace tres meses puede tener hoy una vulnerabilidad crítica publicada la semana pasada. El escaneo en la construcción es una foto; lo que hace falta es reescanear lo que está en ejecución, con una consulta que empiece por el inventario de huellas desplegadas.

```
$ kubectl get pods -A -o jsonpath='{..imageID}' | tr ' ' '\n' | sort -u \
  | while read img; do trivy image --severity CRITICAL --ignore-unfixed -q "$img"; done
```

Ejemplo trabajado

CloudShop endurece su cadena de suministro. El detonante es una pregunta que llega un viernes y que nadie puede responder; los cuatro cambios posteriores se ordenan por lo que aporta cada uno, no por lo que suena mejor.

El detonante — «¿nos afecta?» sin poder contestar.

Se publica una vulnerabilidad crítica en una biblioteca de uso muy extendido. Dirección pregunta si CloudShop está expuesta.

```
tiempo hasta una respuesta con confianza      6 días
método                          revisar repositorios uno a uno,
                                comprobar qué versión se desplegó de verdad,
                                y aun así con dudas sobre dependencias transitivas
respuesta final                 2 servicios afectados de 14
```

Seis días sin poder decir nada. Con inventario adjunto a cada imagen, el mismo ejercicio repetido tres meses después con otra vulnerabilidad:

```
tiempo hasta la respuesta                    11 minutos
servicios revisados                          14 de 14, por huella desplegada
afectados                                    1, con la versión exacta
```

Cambio 1 — inventario en cada construcción.

```
inventario de componentes                   ninguno      generado y adjunto
comparación entre versiones              no era posible   automática en el pull request
dependencias transitivas nuevas detectadas       —          3 en el primer trimestre
```

Una de esas tres era un paquete que nadie había pedido, incorporado por una actualización menor de otra dependencia. Se detectó por la comparación de inventarios, no por una alerta de seguridad.

Cambio 2 — la puerta de vulnerabilidades que se había desactivado.

```
$ git log --oneline -S 'continue-on-error: true' .github/workflows/publicar.yml
a4f8e21  "desbloquear la canalización"
```

Ocho meses atrás alguien había desactivado el escáner porque bloqueaba por 412 hallazgos, la mayoría sin corrección disponible. Desde entonces figuraba como implantado.

```
criterio de bloqueo              crítico y alto, todo    crítico y CORREGIBLE
hallazgos que bloqueaban                  412                  41
tras actualizar la base                    —                    3
cadencia de actualización de la base     ninguna         mensual, por robot
excepciones                          sin fecha ni motivo   3, con responsable
                                                          y fecha de revisión
escáner activo                        desactivado           activo
```

La cifra de 41 a 3 se consiguió actualizando la base, no corrigiendo hallazgos uno a uno. Es la palanca que este programa ya identificó en las clases 046 y 058: corregir por bloque siempre gana a corregir por elemento.

Cambio 3 — firmado durante dos meses sin verificar nada.

La firma se adoptó pronto porque era fácil. La verificación no, porque exigía tocar la admisión del clúster. Una revisión encontró lo previsible:

```
$ kubectl get pods -A -o jsonpath='{..imageID}' | tr ' ' '\n' | sort -u \
  | while read i; do cosign verify "$i" --certificate-identity-regexp '…' \
      --certificate-oidc-issuer '…' >/dev/null 2>&1 || echo "SIN FIRMA VÁLIDA: $i"; done
SIN FIRMA VÁLIDA: registro.interno/cloudshop/informes@sha256:c74e…
SIN FIRMA VÁLIDA: registro.interno/cloudshop/migrador@sha256:81ab…
```

Dos imágenes en producción sin firma válida. Una se había publicado desde un portátil durante una urgencia; la otra la había construido una canalización antigua que nunca se migró.

```
imágenes firmadas                          12 de 14         14 de 14
verificación en la admisión                 ninguna         política obligatoria
identidad del firmante comprobada              —        expresión acotada al
                                                        repositorio y a la rama
prueba negativa                             ninguna     imagen sin firma rechazada
imágenes sin trazar a un commit                2                0
```

La quinta aparición de la misma familia de fallos del programa: un mecanismo que parecía estar protegiendo y no estaba conectado a nada.

Cambio 4 — procedencia exigida, y lo que destapó.

Al exigir que la procedencia declarara un repositorio de la organización y la rama principal, tres despliegues empezaron a fallar:

```
dos       construidos desde ramas de trabajo, en pruebas: correcto que fallen
uno       construido por una canalización de un repositorio archivado
          que seguía publicando a producción cada noche
```

El tercero llevaba catorce meses ejecutándose y nadie sabía que existía esa canalización.

```
canalizaciones que publican a producción      4 (una desconocida)   3
procedencia exigida en la admisión           no                sí
despliegues sin origen verificable             1                 0
```

Y el paso que faltaba: reescanear lo que está en ejecución.

```
escaneo                            solo en la construcción   también semanal
                                                             sobre las huellas
                                                             desplegadas
vulnerabilidades críticas descubiertas
  en imágenes ya desplegadas               no se detectaban    2 el primer mes
tiempo desde la publicación de la
  vulnerabilidad hasta detectarla                —            < 7 días
```

Las dos del primer mes correspondían a imágenes que habían pasado limpias en su construcción. La imagen no había cambiado; lo que cambió fue lo que se sabe de ella.

Resumen de la cadena de suministro:

```
tiempo para responder "¿nos afecta?"          6 días         11 min
escáner activo en la canalización            desactivado      activo
hallazgos críticos y corregibles                41              3
imágenes con firma verificada                 0 de 14        14 de 14
imágenes sin trazar a un commit                 2              0
canalizaciones que publican a producción      4 (una ignorada)  3
reescaneo de lo desplegado                    ninguno        semanal
excepciones con responsable y fecha           0 de 3          3 de 3
```

La lección que esta clase traslada al resto de la parte 05: los cuatro artefactos —inventario, firma, procedencia y escaneo— solo valen conectados a algo que decida. El inventario valió porque alguien lo consultó, la firma no valía nada hasta que la admisión la verificó, la procedencia destapó una canalización fantasma en cuanto se exigió, y el escáner llevaba ocho meses desactivado porque bloqueaba por lo que no se podía arreglar. Generar evidencia es barato; usarla es lo que cuesta y lo único que protege.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/067-registros-sbom-firma-y-procedencia-de-imagenes/lab.py
```

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

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

Evidencia esperada

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

Reto verificable

Construye cadena-suministro-imagen 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
Nadie puede decir si una vulnerabilidad recién publicada afecta a la organización	No hay inventario de componentes asociado a las imágenes desplegadas	Genera el inventario en la construcción, adjúntalo a la huella y consulta por las huellas en ejecución.
El escáner de vulnerabilidades acaba desactivado	Bloquea por hallazgos sin corrección disponible, así que impide avanzar sin aportar	Bloquea solo por crítico y corregible, actualiza la base con cadencia y registra excepciones con motivo, responsable y fecha.
Las imágenes están firmadas y se ejecutan imágenes sin firmar	Se adoptó la firma y no la verificación en la admisión	Añade la política de admisión, acota la identidad del firmante y comprueba con una imagen sin firma que el rechazo ocurre.
Una imagen en producción no se puede trazar a ningún commit	Se publicó fuera de la canalización y nada lo impedía	Exige procedencia en la admisión y separa las identidades de publicación y de descarga en el registro.
Una imagen escaneada limpia resulta vulnerable meses después	El escaneo de construcción es una foto y las vulnerabilidades conocidas cambian	Reescanea periódicamente las huellas que están en ejecución, no solo las que se construyen.
Una política de retención borra una imagen que producción está usando	La retención se definió por antigüedad y ese despliegue llevaba meses sin cambiar	Define la retención por referencias: conserva siempre lo que algún despliegue activo referencia.

■ Seguridad, ética y costo

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

■ Preguntas de comprobación

1. ¿Qué tres preguntas responde cada uno de los artefactos de la cadena de suministro, y en qué orden conviene adoptarlos?
2. ¿Por qué la asociación por huella hace que desplegar por etiqueta debilite la verificación de firma?
3. ¿Qué campo hay que acotar al verificar una firma, y con qué tres clases anteriores se relaciona?
4. ¿Qué opción del escáner evita que el equipo acabe desactivándolo, y por qué?
5. ¿Por qué hay que reescanear las imágenes que están en ejecución si la imagen no cambia?

Referencias

• Sigstore (2025). Keyless signing and verification with cosign — identidad federada y registro de transparencia. <https://docs.sigstore.dev/cosign/signing/overview/>

• SLSA (2025). Provenance and build levels — qué declara la procedencia y qué ataques cubre cada nivel. <https://slsa.dev/spec/v1.0/provenance>

• OCI (2025). Referrers API in the distribution specification — artefactos asociados a una imagen por huella. <https://github.com/opencontainers/distribution-spec/blob/main/spec.md#listing-referrers>

• CISA (2024). Minimum elements for a Software Bill of Materials — contenido mínimo y usos previstos. <https://www.cisa.gov/sbom>

• Aqua Security (2025). Trivy: filtering and ignore files — severidad, hallazgos sin corrección y excepciones con caducidad. <https://trivy.dev/latest/docs/configuration/filtering/>

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

Descargar: Parte 05 en PDF · Manual integral

Anterior	Índice	Siguiente
← 066 · Docker Compose y aplicaciones multiservicio	Parte 05 · Programa	068 · Límites, health checks y apagado ordenado →

Evaluación — 067 Registros, SBOM, firma y procedencia de imágenes

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 cadena-suministro-imagen 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.

068 — Límites, health checks y apagado ordenado

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 4 Laboratorio: reliability · Estado: EXECUTABLECORE

Propósito

Hacer que un contenedor arranque, sirva y se apague sin perder peticiones, que es lo que ocurre entre el momento en que la plataforma decide algo y el momento en que el proceso se entera. Dos hechos explican casi todos los errores durante los despliegues: las tres comprobaciones de estado responden preguntas distintas y confundirlas convierte un problema ajeno en una caída propia; y la señal de parada y la retirada del tráfico ocurren a la vez, así que hay una ventana en la que siguen llegando peticiones a un proceso que ya está cerrando.

Resultados de aprendizaje

Al finalizar podrás:

1. Distinguir las tres comprobaciones de estado por la pregunta que responden y por lo que provocan al fallar.
2. Describir la secuencia completa de un apagado y localizar la ventana en la que se pierden peticiones.
3. Calcular el plazo de gracia a partir de la petición más larga y del retardo de retirada.
4. Decidir con criterio si poner límite de CPU, sabiendo qué protege y qué empeora.
5. Demostrar con una prueba de carga que un despliegue no produce errores.

Conceptos centrales

Concepto	Comprensión verificable
comprobación de vivacidad	¿El proceso está atascado? Al fallar, se reinicia el contenedor. Por eso no debe consultar dependencias: un fallo ajeno se convertiría en una caída propia.
comprobación de disponibilidad	¿Puede servir ahora? Al fallar, se retira del reparto de tráfico sin reiniciar nada. Aquí sí se consultan las dependencias.
comprobación de arranque	¿Sigue iniciándose? Mientras responde que sí, los fallos de las otras dos no cuentan. Es lo que evita el reinicio en bucle de un proceso lento.
ventana de carrera del apagado	Intervalo entre que la plataforma envía la señal de parada y que deja de enviar tráfico. Ocurren en paralelo, y ahí se pierden las peticiones de un despliegue.
plazo de gracia	Tiempo entre la señal de parada y la de matar. Debe ser mayor que el retardo de retirada más la petición más larga más el cierre de recursos.
solicitud frente a límite	La solicitud decide dónde cabe el contenedor; el límite decide qué ocurre cuando se pasa. Son dos decisiones distintas y se confunden constantemente.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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

■ Flujo de razonamiento

Desarrollo

1. Tres comprobaciones, tres preguntas, tres consecuencias

Las tres se escriben igual y hacen cosas muy distintas. Confundirlas es la causa del incidente más caro de esta clase.

	Pregunta	Al fallar	¿Consulta dependencias?
Vivacidad	¿Está atascado?	Reinicia el contenedor	No
Disponibilidad	¿Puede servir ahora?	Lo retira del reparto	Sí
Arranque	¿Sigue iniciándose?	Espera; las otras no cuentan	No

La columna de la derecha es la que decide, y merece un ejemplo concreto de lo que pasa si se pone mal:

```
vivacidad que consulta la base de datos
  la base conmuta durante 40 segundos
  TODAS las réplicas fallan la comprobación a la vez
  la plataforma las reinicia TODAS a la vez
  las nuevas tardan 90 s en arrancar y vuelven a fallar
```

```
  → un fallo ajeno de 40 s se convierte en una caída propia de varios minutos
```

Es el mismo incidente que la clase 052 midió con la reparación automática, y la razón es idéntica: el mecanismo no retira, destruye. Por eso la comprobación de vivacidad tiene que ser lo más tonta posible:

```
vivacidad correcta      ¿responde el proceso? ¿está el bucle principal vivo?
                        sin red hacia fuera, sin base de datos, sin caché
disponibilidad correcta ¿tengo conexión a la base? ¿está la caché? ¿he cargado
                        la configuración? ¿estoy cerrando?
```

Y la de arranque resuelve un problema distinto que se manifiesta como un bucle infinito. Un proceso que tarda dos minutos en cargar su estado inicial falla la comprobación de vivacidad mucho antes de estar listo, así que lo reinician, y vuelve a empezar:

```
sin comprobación de arranque, con vivacidad cada 10 s y 3 fallos permitidos
  a los 30 s se reinicia; el proceso necesitaba 120
  → nunca llega a estar listo, y el registro solo muestra arranques
```

La alternativa antigua —un retardo inicial largo en la comprobación de vivacidad— funciona y tiene un coste: ese retardo se aplica también después del arranque, así que un proceso atascado tarda ese mismo tiempo en detectarse. La comprobación de arranque separa los dos casos: tolerante al principio, exigente después.

```
startupProbe:   {httpGet: {path: /healthz}, periodSeconds: 5,  failureThreshold: 30}
livenessProbe:  {httpGet: {path: /healthz}, periodSeconds: 10, failureThreshold: 3}
readinessProbe: {httpGet: {path: /readyz},  periodSeconds: 5,  failureThreshold: 2}
```

Con esos números, el arranque tolera hasta 150 segundos y, una vez listo, un atasco se detecta en 30. Y hay una aritmética que conviene comprobar siempre: el tiempo de espera de la comprobación tiene que caber en su periodo. Una comprobación cada 5 segundos con un tiempo de espera de 10 se solapa consigo misma y produce fallos bajo carga que no corresponden a nada.

Un último caso que aparece en aplicaciones con estado interno: una comprobación de disponibilidad debe poder decir «no» por decisión propia. Un proceso que está reconstruyendo un índice, drenando una cola o cerrando debe retirarse del reparto sin que nadie se lo pida. Esa es la mitad de la solución del apartado siguiente.

2. La ventana en la que se pierden las peticiones

Aquí está el hecho que explica los errores que aparecen en cada despliegue y que casi siempre se archivan como «normal al desplegar».

Cuando la plataforma decide parar un contenedor ocurren dos cosas a la vez:

```
1. envía la señal de parada al proceso
2. empieza a retirarlo de la lista de destinos del reparto de tráfico
```

La segunda es asíncrona y tarda: hay que propagar el cambio a los balanceadores, a las tablas de rutas y a los clientes. Mientras tanto, siguen llegando peticiones. Si el proceso cierra su escuchador al recibir la señal —que es lo que hace la implementación intuitiva—, esas peticiones se rechazan.

```
t+0,00  señal de parada       el proceso cierra el escuchador
t+0,00  empieza la retirada
t+0,05  llega una petición    → conexión rechazada
t+0,30  llega otra            → conexión rechazada
t+2,10  la retirada se completa
```

Dos segundos de errores por contenedor y por despliegue. Con veinte contenedores y ocho despliegues al día, es una cifra que aparece en el presupuesto de error de la clase 057 sin que nadie sepa de dónde sale.

La corrección es contraintuitiva y es la clave de la clase: al recibir la señal de parada, lo primero es esperar.

```
secuencia correcta
  1. recibir la señal
  2. marcar la disponibilidad como "no": la plataforma acelera la retirada
  3. ESPERAR el tiempo de propagación (2-5 s típicos) SIN dejar de servir
  4. cerrar el escuchador: no se aceptan conexiones nuevas
  5. terminar las peticiones en curso
  6. cerrar recursos: base de datos, caché, productores de mensajes
  7. salir con código 0
```

El paso 3 es el que falta en casi todas las implementaciones. Se puede hacer en el código o con un gancho previo a la parada que la plataforma ejecuta antes de la señal:

```
lifecycle:
```

```
  preStop:
    exec: {command: ["/bin/sleep", "5"]}
terminationGracePeriodSeconds: 60
```

Y en el código, la forma que además sirve para el paso 2:

```
sig := make(chan os.Signal, 1)
signal.Notify(sig, syscall.SIGTERM, syscall.SIGINT)
<-sig

listo.Store(false)              // 2: /readyz empieza a devolver 503
time.Sleep(5 * time.Second)     // 3: se sigue SIRVIENDO durante la espera

ctx, cancel := context.WithTimeout(context.Background(), 45*time.Second)
defer cancel()
srv.Shutdown(ctx)               // 4 y 5: sin conexiones nuevas, termina las vivas
cerrarRecursos()                // 6
```

Y un detalle que hace que todo lo anterior no baste: las conexiones persistentes. Un balanceador que mantiene conexiones abiertas puede seguir enviando peticiones por una conexión ya establecida aunque el destino se haya retirado de la lista. La solución es que el servidor deje de reutilizar esas conexiones en cuanto empieza a cerrar:

```
durante el apagado, responder con la cabecera que indica cierre de conexión
→ el cliente abre una conexión nueva, que ya irá a otro destino
```

Sin eso, un cliente con una conexión persistente puede seguir enviando peticiones al contenedor que está cerrando durante todo el plazo de gracia.

3. La aritmética del plazo de gracia

El plazo de gracia es el tiempo entre la señal de parada y la señal que mata sin remedio. Y no es un número redondo que se elige: se calcula.

```
plazo de gracia  >  retardo de propagación
                  + petición más larga que se admite
                  + cierre de recursos
                  + margen
```

Con cifras reales de un servicio:

```
retardo de propagación de la retirada        5 s
percentil 99,9 de la duración de la petición 12 s
cierre de conexiones y volcado de telemetría  3 s
margen                                        10 s
                                             ■■■■■■
plazo de gracia mínimo                        30 s
```

Y los dos errores que produce equivocarse en cada dirección:

```
plazo demasiado CORTO
  la señal de matar llega con peticiones en curso
  → se cortan a mitad, y el cliente ve una conexión cerrada sin respuesta
  → en un proceso por lotes, trabajo a medias sin marcar

plazo demasiado LARGO
  cada despliegue tarda ese plazo por contenedor si el proceso no sale antes
  → un despliegue de veinte contenedores con plazo de 300 s puede durar
    una hora si el proceso nunca sale por su cuenta
```

El segundo aparece cuando el proceso no maneja la señal (clase 063): el plazo entero se consume siempre, en cada contenedor. La comprobación es directa:

```
$ time docker stop demo
real    0m10,4s      ← plazo completo: nadie escuchó

$ time docker stop demo
real    0m1,2s       ← salió por su cuenta: correcto
```

Esa medición debería estar en la lista de verificación de cualquier servicio. Un apagado que tarda exactamente el plazo configurado es siempre un fallo, aunque no produzca errores visibles.

Y hay un caso que exige más cuidado: los consumidores de mensajes. Cerrar un consumidor a mitad de un mensaje no confirmado hace que se reentregue —lo cual es correcto y para eso está la idempotencia de las clases 044 y 056—, pero un apagado ordenado lo hace mejor:

```
1. dejar de pedir mensajes nuevos
2. terminar los que se están procesando y confirmarlos
3. cerrar el cliente
```

Con eso, un despliegue no genera reentregas. Sin eso, cada despliegue produce una tanda de duplicados que la idempotencia absorbe pero que aparece en las métricas y confunde durante un incidente.

Y para los trabajos por lotes, el plazo de gracia rara vez alcanza: un proceso de dos horas no cabe en ningún plazo razonable. La respuesta correcta no es alargar el plazo sino que el trabajo sea reanudable: puntos de control periódicos, y al recibir la señal, marcar el progreso y salir. Es exactamente el mismo diseño que exigían los avisos de desalojo de 30 segundos de las clases 040 y 052.

4. Solicitudes y límites: dos decisiones que se confunden

La clase 063 explicó el mecanismo; aquí está la decisión.

```
solicitud   lo que la plataforma RESERVA para decidir dónde cabe
            no limita nada: es planificación
límite      lo que el grupo de control impone
            memoria: mata · CPU: frena
```

Y de ahí salen los cuatro casos posibles, con lo que produce cada uno:

```
sin solicitud, sin límite    el contenedor cabe en cualquier sitio y puede
                             consumirlo todo: tumba a sus vecinos
solicitud sin límite         se planifica bien y sigue pudiendo consumir de más
límite sin solicitud         se planifica mal: la plataforma cree que no consume
ambos                        lo normal
```

Sobre la memoria, no hay debate y conviene decirlo así: el límite es obligatorio. Sin él, un contenedor con una fuga agota la memoria del nodo y el núcleo elige una víctima entre todos los procesos de la máquina, que puede ser cualquiera. Con límite, se mata el culpable. Es la conclusión de la clase 063.

Sobre la CPU sí hay un debate real y merece presentarse con honestidad, porque la respuesta no es la misma en todos los casos:

```
a favor del límite de CPU
  impide que un contenedor con un bucle descontrolado consuma el nodo
  hace el rendimiento predecible y reproducible entre entornos
  en plataformas multiinquilino, es un requisito

en contra
  la limitación frena aunque el nodo tenga CPU libre
  un pico corto y legítimo se convierte en latencia
  con un límite ajustado, el percentil alto empeora sin que falte capacidad
```

La posición defendible, y la que conviene escribir en la línea base:

```
memoria      solicitud y límite, siempre, y el límite ajustado al uso real
CPU          solicitud siempre; límite sí cuando el nodo se comparte entre
             equipos o cuando la reproducibilidad importa más que el pico,
             y NO cuando el servicio es sensible a la latencia y el nodo
             es del propio equipo
en cualquier caso  vigilar la proporción de periodos limitados (clase 063):
                   si sube del 5 %, el límite está haciendo daño
```

La última línea convierte el debate en una medición. No hace falta decidir en abstracto: se pone el límite, se mide la limitación y se ajusta con el dato delante.

Y el reinicio en bucle, que es donde límites y comprobaciones se cruzan. Un contenedor que muere se reinicia con espera creciente, y esa espera es la que salva al nodo:

```
reinicio 1   inmediato
reinicio 2   10 s
reinicio 3   20 s … hasta un tope de varios minutos
```

El caso que rompe esto es una comprobación de vivacidad mal ajustada: el contenedor no muere, lo matan, y el ciclo puede ser mucho más rápido. El síntoma es un servicio que reinicia constantemente con el proceso aparentemente sano, y la causa está casi siempre en el apartado primero de esta clase.

5. Demostrarlo: un despliegue sin errores es una medición

Todo lo anterior se puede afirmar o se puede demostrar, y este programa ya ha establecido cuál de las dos cuenta.

La prueba es un despliegue con carga en curso:

```
# carga sostenida durante todo el ejercicio
$ hey -z 3m -c 50 -q 20 https://tienda.interno/api/pedidos > carga.txt &

# despliegue a mitad
$ sleep 60 && kubectl set image deploy/tienda tienda=registro/tienda@sha256:nueva…
$ kubectl rollout status deploy/tienda

$ grep -E 'Status code|error' carga.txt
  [200] 179880 responses
  [503]      0 responses
  errors     0                                                    ✓
```

Cero errores durante un despliegue completo es el criterio. Y conviene fijarlo como umbral en la canalización, porque es una propiedad que se rompe sola: un cambio en el plazo de gracia, una petición que se vuelve más lenta o una comprobación mal ajustada la degradan sin que nadie lo note.

La lista de verificación de esta clase, que se aplica servicio a servicio:

- las tres comprobaciones existen y responden preguntas distintas
- la de vivacidad NO consulta ninguna dependencia
- la de disponibilidad SÍ, y puede decir "no" por decisión propia
- el tiempo de espera de cada comprobación cabe en su periodo
- el proceso maneja la señal de parada: `docker stop` tarda < 2 s
- hay retardo antes de cerrar el escuchador, mayor que la propagación
- el plazo de gracia es mayor que retardo + petición más larga + cierre
- durante el apagado se dejan de reutilizar las conexiones persistentes
- los consumidores terminan y confirman lo que tienen en curso
- los trabajos largos son reanudables, no dependen del plazo
- memoria: solicitud y límite; CPU: solicitud, y límite con criterio
- un despliegue con carga produce cero errores

Y dos mediciones que conviene tener como métricas permanentes, porque detectan la degradación de esta lista sin que nadie la revise:

```
duración del apagado por contenedor    si se acerca al plazo, alguien dejó de
                                       manejar la señal
errores durante la ventana de despliegue  debe ser cero; cualquier otra cosa
                                       es una regresión de esta clase
```

Un cierre que conecta con el resto del programa: de las once leyes que la clase 060 enumeró, esta clase toca tres —la conmutación como funcionamiento normal, el reintento del cliente y el fallo silencioso—. Y añade una observación propia que la parte 06 va a necesitar: el contrato entre la aplicación y la plataforma no es solo la imagen. Es también qué señales entiende, qué responde en cada comprobación y cuánto tarda en irse. Esa parte del contrato no está en ninguna especificación OCI, y es exactamente la cuarta fuga que la clase 060 predijo: lo que ocurre antes del primer proceso y después de la señal de terminación.

Ejemplo trabajado

CloudShop despliega ocho veces al día y cada despliegue produce errores. Nadie lo trata como un incidente porque «es normal al desplegar». Cinco correcciones lo llevan a cero, y la última destapa un problema que llevaba meses.

Punto de partida:

```
una sola comprobación, /health, usada como vivacidad y disponibilidad
consulta la base de datos
plazo de gracia por defecto: 30 s
el proceso no maneja la señal de parada
errores por despliegue: entre 300 y 900
```

Corrección 1 — la comprobación que convertía un parpadeo ajeno en una caída propia.

Una conmutación de la base de datos de 38 segundos produjo 11 minutos de caída:

```
$ kubectl get events --field-selector reason=Unhealthy -o wide | head
… Liveness probe failed: HTTP probe failed with statuscode: 503   (× 14 pods)
$ kubectl get pods -l app=tienda -o jsonpath='{..restartCount}'
3 3 3 3 3 3 3 3 3 3 3 3 3
```

Las catorce réplicas fallaron la vivacidad a la vez y se reiniciaron a la vez; las nuevas tardaban 70 s en arrancar y volvían a fallar.

```
comprobaciones                          1 (/health)      3, separadas
vivacidad                         consulta la base   solo el bucle principal
disponibilidad                          —            base y caché
arranque                                —            hasta 150 s tolerados
duración del episodio equivalente      11 min            38 s
reinicios durante el episodio            42               0
```

Corrección 2 — los errores de cada despliegue tenían una causa concreta.

```
t+0,00  señal de parada · el proceso cierra el escuchador al instante
t+0,00  empieza la retirada de destinos
t+2,30  la retirada se completa
→ 2,3 s de conexiones rechazadas por contenedor
```

Con catorce contenedores y ocho despliegues diarios, la cuenta cuadraba con los errores observados.

```
retardo antes de cerrar el escuchador       0 s             5 s
disponibilidad marcada como "no" al recibir
  la señal                                    no             sí
errores por despliegue                     300-900          0
```

Cinco segundos de espera por contenedor eliminaron entre 2.400 y 7.200 errores diarios.

Corrección 3 — el plazo de gracia y las peticiones cortadas.

Aunque los errores de conexión desaparecieron, quedaba un residuo de respuestas incompletas.

```
plazo de gracia                            30 s
retardo de propagación                      5 s
percentil 99,9 de la petición              12 s
cierre de recursos                          3 s
→ 20 s necesarios, 30 disponibles: correcto
```

Pero la exportación de informes admitía peticiones de hasta 90 segundos:

```
plazo de gracia (servicio general)         30 s             30 s
plazo de gracia (exportación)              30 s            120 s
peticiones cortadas por despliegue          17               0
```

La lección anotada: el plazo se calcula por servicio, no por convención de la organización.

Corrección 4 — el proceso que nunca salía por su cuenta.

```
$ time kubectl delete pod tienda-7f4c9
real    0m30,2s        ← el plazo entero, en cada contenedor
```

El punto de entrada estaba en forma de cadena, así que el proceso número uno era un intérprete de órdenes que no reenviaba las señales — el fallo de las clases 062 y 063.

```
punto de entrada                       forma de cadena   forma de lista
duración del apagado                       30,2 s           1,1 s
duración de un despliegue completo        7 min 20 s      1 min 05 s
```

Seis minutos menos por despliegue, ocho veces al día.

Corrección 5 — el consumidor que duplicaba en cada despliegue.

Al revisar las métricas de mensajería apareció un patrón que llevaba meses:

```
mensajes reentregados      picos exactos en cada despliegue
cantidad por despliegue    entre 40 y 120
```

El consumidor cerraba su cliente al recibir la señal, dejando sin confirmar los mensajes en curso. La idempotencia de la clase 056 evitaba el daño, así que nadie lo había investigado.

```
apagado del consumidor          cierre inmediato     deja de pedir, termina
                                                     los que tiene, confirma
reentregas por despliegue              40-120              0
efectos duplicados                        0                0   ← ya era 0
```

Cero efectos duplicados antes y después: la idempotencia funcionaba. Lo que se corrigió fue el ruido, y con él la confusión que ese ruido causaba durante cualquier incidente coincidente con un despliegue.

La prueba final:

```
$ hey -z 3m -c 50 -q 20 https://tienda.interno/api/pedidos > carga.txt &
$ sleep 60 && kubectl set image deploy/tienda tienda=registro/tienda@sha256:nueva…
$ grep -E '\[503\]|errors' carga.txt
```

```
  [503]  0 responses
  errors 0                                                    ✓
```

Resumen:

```
comprobaciones de estado                          1            3
reinicios durante una conmutación de la base     42            0
duración de un episodio de 38 s ajeno         11 min         38 s
errores por despliegue                       300-900          0
peticiones cortadas por despliegue              17            0
duración del apagado por contenedor           30,2 s        1,1 s
duración de un despliegue completo          7 min 20 s    1 min 05 s
reentregas de mensajes por despliegue         40-120          0
```

La lección que esta clase traslada al resto de la parte 05: los cinco problemas ocurrían en cada despliegue, ocho veces al día, y ninguno se estaba tratando como un incidente. Se habían normalizado porque coincidían con una operación esperada, y esa normalización es lo que impidió verlos durante meses. El criterio que los destapó es una sola frase que ahora está en la línea base: un despliegue con carga en curso produce cero errores, y cualquier otra cosa es una regresión.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/068-limites-health-checks-y-apagado-ordenado/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 contenedor-operable 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 contenedor-operable 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 fallo breve de la base de datos provoca el reinicio de toda la flota	La comprobación de vivacidad consulta dependencias, y al fallar reinicia en vez de retirar	Vivacidad sin dependencias, disponibilidad con ellas; el reinicio es para procesos atascados, no para dependencias caídas.
Un proceso lento reinicia en bucle y nunca llega a estar listo	La vivacidad lo mata antes de terminar el arranque	Añade comprobación de arranque tolerante; una vez listo, la vivacidad puede ser exigente.
Cada despliegue produce un pico de errores de conexión	El proceso cierra el escuchador al recibir la señal mientras la retirada de tráfico aún se propaga	Marca la disponibilidad como no, espera el tiempo de propagación sirviendo, y solo entonces cierra.
El apagado tarda exactamente el plazo de gracia configurado	El proceso no maneja la señal, casi siempre por un punto de entrada en forma de cadena	Forma de lista y manejador declarado; mide que stop tarde menos de dos segundos.
Se cortan peticiones largas durante los despliegues	El plazo de gracia es menor que el retardo de propagación más la petición más larga	Calcula el plazo por servicio; para trabajos largos, hazlos reanudables en lugar de alargar el plazo.
Cada despliegue genera una tanda de mensajes reentregados	El consumidor cierra sin terminar ni confirmar lo que estaba procesando	Deja de pedir mensajes, termina y confirma los en curso, y después cierra 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 pregunta responde cada una de las tres comprobaciones y qué provoca el fallo de cada una?
2. Describe la ventana en la que se pierden peticiones durante un apagado y la corrección contraintuitiva que la elimina.
3. Calcula el plazo de gracia de un servicio con propagación de 5 s, percentil 99,9 de 12 s y cierre de 3 s.
4. ¿Cuándo pondrías límite de CPU y cuándo no, y qué medición resuelve el debate?
5. ¿Qué medición demuestra que un despliegue no pierde peticiones, y por qué debe ser un umbral de la canalización?

Referencias

• Kubernetes (2025). Configure liveness, readiness and startup probes — semántica y parámetros de las tres. <https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>

• Kubernetes (2025). Pod lifecycle: termination — orden de la señal, retirada de destinos y plazo de gracia. <https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-termination>

• Kubernetes (2025). Requests and limits — planificación frente a imposición, y efectos de cada una. <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>

• Docker (2025). docker stop and signal handling — plazo, señales y comportamiento del proceso número uno. <https://docs.docker.com/reference/cli/docker/container/stop/>

• Google (2018). The Site Reliability Workbook, cap. 4 — errores durante despliegues y presupuesto de error. <https://sre.google/workbook/implementing-slos/>

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

Descargar: Parte 05 en PDF · Manual integral

Evaluación — 068 Límites, health checks y apagado ordenado

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 contenedor-operable 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.

069 — Rootless, capabilities, seccomp y secretos

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 4 Laboratorio: security · Estado: EXECUTABLECORE

Propósito

Reducir lo que un contenedor puede hacer contra el anfitrión, partiendo de un hecho que la clase 063 dejó establecido y que casi nadie asume: el núcleo es uno solo, así que el usuario cero dentro del contenedor es el usuario cero del núcleo, con algunas capacidades retiradas. La clase ordena los cuatro mecanismos que acotan eso —usuario, capacidades, filtro de llamadas al sistema y control de acceso obligatorio—, señala cuál de ellos se desactiva más a menudo por copiar una respuesta de internet, y cierra el tratamiento de secretos en ejecución que las clases 058 y 066 dejaron a medias.

Resultados de aprendizaje

Al finalizar podrás:

1. Ejecutar sin privilegios y explicar qué cambia realmente respecto de ejecutar como usuario cero.
2. Retirar todas las capacidades y añadir solo las necesarias, con la prueba negativa correspondiente.
3. Reconocer las opciones que anulan las defensas y qué se pierde exactamente con cada una.
4. Sustituir el socket del motor por un constructor sin privilegios en las canalizaciones.
5. Inyectar secretos en ejecución sin que aparezcan en el entorno del proceso ni en un volcado.

Conceptos centrales

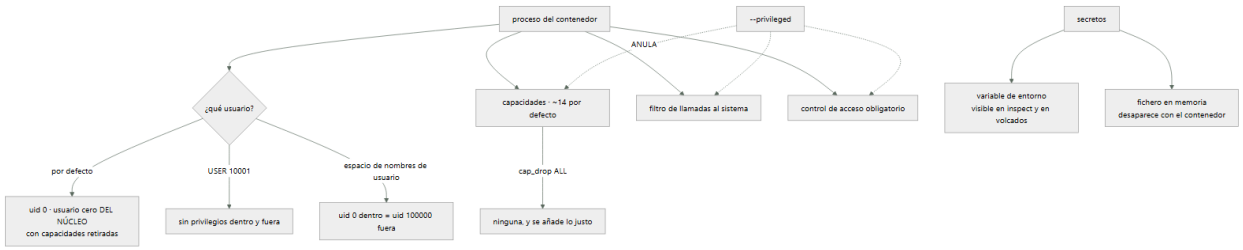
Concepto	Comprensión verificable
capacidad	Una de las cuarenta piezas en que está dividido el poder del usuario cero. Un contenedor no privilegiado conserva unas catorce por defecto, y varias sorprenden.
modo privilegiado	Todas las capacidades, todos los dispositivos y ninguna política de filtrado. Es equivalente a ser administrador del anfitrión, no una versión algo más permisiva.
sin nuevos privilegios	Impide que un proceso gane permisos al ejecutar un binario con bit de elevación. Cuesta una línea y corta una familia entera de escaladas.
filtro de llamadas al sistema	Lista de llamadas permitidas. El perfil por defecto bloquea unas decenas; desactivarlo es lo que sugieren muchas respuestas de internet para arreglar un fallo.
espacio de nombres de usuario	Traducción de identificadores: el usuario cero dentro corresponde a un usuario sin privilegios fuera. Es la única de las cuatro capas que cambia el resultado de una fuga.
secreto en variable de entorno	Valor legible en la configuración del contenedor, en el entorno del proceso y en cualquier volcado de fallo. Es el mecanismo más cómodo y el peor.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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 usuario cero del contenedor es el del núcleo

La clase 063 estableció que el núcleo es uno solo. La consecuencia para la seguridad es directa y se enuncia mal muy a menudo:

```
contenedor sin opciones especiales
  el proceso corre como uid 0
  el núcleo lo VE como uid 0
  lo que lo separa del anfitrión son los espacios de nombres
  y un conjunto reducido de capacidades
```

Es decir: no hay una capa de traducción. Si aparece una vulnerabilidad del núcleo que permita salir del aislamiento, se sale como administrador de la máquina. Y hay caminos más prosaicos: un volumen montado con datos del anfitrión, un dispositivo accesible, un socket del motor.

Lo primero, entonces, es no ser el usuario cero:

```
USER 10001:10001
```

Y conviene declararlo con número, por lo que la clase 061 explicó: las políticas de plataforma comprueban el número, y un nombre que hay que resolver contra el fichero de usuarios de la imagen no siempre se puede verificar.

Dos objeciones habituales, con su respuesta:

```
"necesito el puerto 80"
  → escucha en 8080 y publica el 80 desde fuera (clase 065)
     abrir puertos bajos exige una capacidad que casi nunca hace falta

"necesito escribir en el sistema de ficheros"
  → declara dónde: un volumen o un montaje en memoria (clase 064)
     y deja la raíz de solo lectura
```

La segunda lleva a una configuración que conviene adoptar como valor por defecto y que corta muchas cadenas de explotación de golpe:

```
$ docker run \
    --user 10001:10001 \
    --read-only --tmpfs /tmp:rw,noexec,nosuid,size=64m \
    --cap-drop ALL \
    --security-opt no-new-privileges:true \
    registro/tienda@sha256:9f2c…
```

Cuatro opciones. Y la tercera merece detallarse porque el conjunto por defecto sorprende.

Las capacidades son las piezas en que está dividido el poder del usuario cero. Un contenedor no privilegiado conserva unas catorce, y estas tres explican por qué conviene retirarlas todas:

```
ignorar permisos de fichero   permite leer cualquier fichero del sistema
                              de ficheros del contenedor, incluidos los montados,
                              sin importar sus permisos
cambiar de identidad          permite convertirse en otro usuario
cambiar propietarios          permite reasignar ficheros
```

La primera es la que convierte una ejecución de código en lectura de todo lo montado. Retirarlas y añadir solo lo necesario es una línea:

```
--cap-drop ALL --cap-add NET_BIND_SERVICE      # solo si de verdad hace falta
```

Y la prueba negativa, que se puede automatizar:

```
$ docker run --rm --cap-drop ALL --user 10001 imagen \
```

```
    sh -c 'cat /etc/shadow' 2>&1 | head -1
cat: /etc/shadow: Permission denied                                    ✓

$ docker inspect contenedor --format '{{.HostConfig.CapAdd}} {{.HostConfig.CapDrop}}'
[NET_BIND_SERVICE] [ALL]                                               ✓
```

2. Las tres opciones que anulan las defensas

Hay tres opciones que aparecen constantemente en respuestas de internet como solución a un fallo, y las tres desactivan protecciones. Conviene saber exactamente qué hace cada una para poder decir que no con argumentos.

Modo privilegiado. Se describe a veces como «un poco más de permisos». No lo es:

```
--privileged
  todas las capacidades
  todos los dispositivos del anfitrión accesibles
  el filtro de llamadas al sistema, desactivado
  el control de acceso obligatorio, desactivado
  las rutas del núcleo enmascaradas, visibles
```

Con eso, montar el disco raíz del anfitrión y cambiar la raíz a él es cuestión de dos órdenes. Es administrador de la máquina, sin matices. Si un contenedor lo necesita, la pregunta correcta no es cómo concederlo con cuidado, sino qué hace ese contenedor y si puede hacerlo de otra forma.

Filtro de llamadas al sistema desactivado. Es la sugerencia habitual cuando algo falla con un error de operación no permitida:

```
--security-opt seccomp=unconfined      # ← "así funciona"
```

El perfil por defecto bloquea varias decenas de llamadas que casi ninguna aplicación necesita y que sí necesitan muchas cadenas de explotación. Desactivarlo funciona, y funciona porque quita el filtro entero. Las alternativas, en orden:

```
1. actualizar el tiempo de ejecución o la biblioteca del sistema
   la mayoría de estos fallos son incompatibilidades ya corregidas
2. permitir la llamada concreta con un perfil derivado del de por defecto
3. desactivarlo, documentado, temporal y con fecha de revisión
```

La tercera es una excepción con caducidad, en el sentido de las clases 046, 049 y 067. Lo que no es aceptable es que aparezca en un fichero sin explicación.

Montar el socket del motor. Ya apareció en la clase 064 y merece repetirse porque es la más frecuente en canalizaciones:

```
quien habla con el socket del motor puede arrancar un contenedor privilegiado
→ montar el socket es conceder administrador del anfitrión
```

Y la alternativa existe y es madura: construir imágenes sin demonio y sin privilegios.

```
# constructor sin privilegios, sin socket y sin modo privilegiado
$ buildctl-daemonless.sh build \
    --frontend dockerfile.v0 --local context=. --local dockerfile=. \
    --output type=image,name=registro/tienda:v9,push=true
```

Con eso, un agente de canalización comprometido no obtiene el anfitrión. Es la corrección con más rendimiento de toda la clase, porque los agentes de construcción ejecutan código de cualquiera que abra un cambio.

Y la cuarta opción, que sí conviene activar y casi nunca está:

```
--security-opt no-new-privileges:true
```

Impide que un proceso gane permisos al ejecutar un binario con bit de elevación. Corta la escalada clásica dentro del contenedor —de usuario sin privilegios a usuario cero del contenedor— y no rompe nada salvo que la aplicación dependa precisamente de eso, que es rarísimo.

3. Espacios de nombres de usuario: la capa que cambia el resultado

Las tres capas anteriores reducen lo que el contenedor puede hacer. Esta cambia qué es desde el punto de vista del núcleo, y es la única que altera el resultado de una fuga.

```
sin espacio de nombres de usuario
  uid 0 dentro  →  uid 0 para el núcleo

con espacio de nombres de usuario
```

```
uid 0 dentro  →  uid 100000 para el núcleo, sin ningún privilegio
```

Es decir: un proceso que consiga salir del aislamiento aparece fuera como un usuario que no puede hacer nada. Lo que dentro parece administración total, fuera es un usuario cualquiera.

Hay dos formas de adoptarlo:

```
el motor lo aplica a los contenedores
  el demonio sigue siendo privilegiado; los contenedores no
modo sin privilegios completo
  el propio demonio corre como usuario normal
  → el anfitrión no tiene ningún proceso de contenedores como usuario cero
```

La segunda es la correcta para máquinas de desarrollo y especialmente para agentes de canalización, por el motivo del apartado anterior: ahí se ejecuta código que no ha revisado nadie.

Y hay que decir lo que cuesta, porque no es gratis y presentarlo como tal lleva a adoptarlo y revertirlo:

```
puertos por debajo de 1024        no se pueden abrir sin configuración extra
algunos controladores de
  almacenamiento                  rendimiento distinto según el sistema de ficheros
montajes del anfitrión            los identificadores se traducen: hay que
                                  recalcular la propiedad (clase 064)
algunas cargas específicas        depuradores, herramientas de red y ciertos
                                  perfiles de rendimiento necesitan ajustes
```

La tercera fila es la que produce la sorpresa inmediata: un directorio del anfitrión que funcionaba con identificador 10001 deja de funcionar, porque ese identificador ahora se traduce a otro. Es el mismo problema de la clase 064 con una capa más, y la corrección es la misma: hacer coincidir los números, ahora contando la traducción.

```
$ cat /etc/subuid
usuario:100000:65536

# uid 10001 dentro  →  110000 fuera
$ sudo chown -R 110000:110000 /datos/subidas
```

El control de acceso obligatorio completa el conjunto y casi nunca hay que tocarlo: el perfil por defecto del motor impide escrituras en rutas sensibles del núcleo y montajes arbitrarios. Lo importante es no desactivarlo, que es lo que hace el modo privilegiado.

Y la jerarquía de las cuatro capas, para saber qué aporta cada una:

```
usuario no privilegiado     lo más barato; evita la mayoría de los accidentes
capacidades retiradas       corta caminos concretos dentro del contenedor
filtro y control obligatorio reduce la superficie del núcleo alcanzable
espacios de usuario         cambia lo que ocurre SI todo lo demás falla
```

Las cuatro son acumulativas y las tres primeras cuestan una línea cada una. La cuarta cuesta trabajo y es la única que responde a la pregunta que importa: qué pasa el día que haya una vulnerabilidad del núcleo.

4. Secretos en ejecución: dónde acaban de verdad

Las clases 058 y 066 dejaron establecido que un secreto en una variable de entorno es un secreto publicado. Conviene ver dónde acaba exactamente, porque la lista es más larga de lo que parece:

```
en la configuración del contenedor      docker inspect lo muestra
en el entorno del proceso               /proc/<pid>/environ, legible por
                                        cualquier proceso del mismo usuario
en los procesos hijos                   se hereda a todo lo que se lance
en los volcados de fallo                muchos recolectores de errores envían
                                        el entorno completo con la excepción
en los registros de arranque            si alguien imprime la configuración
en la plataforma                        en la definición del despliegue,
                                        legible por quien pueda leerla
```

La cuarta es la que produce las filtraciones más embarazosas: un servicio de seguimiento de errores externo recibe la excepción con todas las variables de entorno adjuntas, y ahí van las credenciales.

La alternativa es un fichero en un sistema de ficheros en memoria:

```
no aparece en la configuración del contenedor
no se hereda a los hijos
no lo recoge un volcado de excepción
no toca el disco ni queda en una instantánea del volumen
```

```
se puede releer tras una rotación sin redesplegar (clase 058)
```

```
$ docker run \
    --mount type=tmpfs,destination=/run/secretos,tmpfs-size=1m \
    --env BD_PASSWORD_FILE=/run/secretos/bd \
    registro/tienda@sha256:9f2c…
```

Y la convención de la variable con sufijo de fichero merece adoptarse en toda la organización, porque hace explícito que el valor no está en el entorno:

```
BD_PASSWORD        ← el valor: mal
BD_PASSWORD_FILE   ← la ruta: bien
```

Muchas imágenes oficiales ya la soportan, y para las propias es una línea de código.

Dos comprobaciones que conviene tener automatizadas:

```
# ningún secreto en la configuración del contenedor
$ docker inspect contenedor --format '{{json .Config.Env}}' \
  | grep -Eic 'password|secret|token|key' || echo "sin secretos en el entorno   ✓"

# ningún secreto en el entorno del proceso
$ docker exec contenedor sh -c 'tr "\0" "\n" < /proc/1/environ' \
  | grep -Eic 'password|secret|token' || echo "entorno limpio                ✓"
```

Y una precisión honesta sobre el alcance: nada de esto protege frente a un atacante que ya ejecuta código dentro del contenedor con el mismo usuario que la aplicación. Ese puede leer el fichero igual que lo lee la aplicación. Lo que se consigue es eliminar las vías indirectas —la configuración, los volcados, los registros, los procesos hijos, la definición del despliegue—, que son por donde salen casi todas las filtraciones reales.

La protección contra el primer caso es distinta y ya se trató: privilegio mínimo del secreto (clase 058), duración corta y rotación posible sin corte.

5. La lista de endurecimiento, y sus pruebas

Todo lo anterior cabe en una lista de verificación por contenedor, y su valor está en poder ejecutarla automáticamente:

- usuario numérico distinto de cero
- raíz de solo lectura, con los puntos de escritura declarados
- todas las capacidades retiradas; las añadidas, justificadas una a una
- sin nuevos privilegios activado
- filtro de llamadas al sistema y control obligatorio en su perfil por defecto
- sin modo privilegiado
- sin socket del motor montado
- montajes del anfitrión: los mínimos, y de solo lectura
- límites de memoria y de procesos (clases 063, 068)
- secretos como fichero en memoria, nunca en variables
- imagen sin intérprete de órdenes cuando sea posible (clase 062)

Y la comprobación sobre lo que está en ejecución, que es la que detecta la desviación:

```
$ docker ps -q | while read c; do
    docker inspect "$c" --format '{{.Name}} priv={{.HostConfig.Privileged}} \
user={{.Config.User}} ro={{.HostConfig.ReadonlyRootfs}} caps={{.HostConfig.CapDrop}}'
  done | grep -E 'priv=true|user= |ro=false'
```

Cualquier línea que aparezca es un contenedor que se salta la línea base. Y en una plataforma, esto mismo se expresa como política de admisión, que es la forma sostenible: la lista se comprueba antes de ejecutar, no después.

Las pruebas negativas de esta clase, en el sentido exacto de las clases 046, 058 y 067 —el éxito es que fallen—:

```
# 1. no se puede escribir en la raíz
$ docker exec c touch /prueba
touch: /prueba: Read-only file system                              ✓

# 2. no se puede escalar con un binario de elevación
$ docker exec c sh -c '/usr/bin/newgrp root'
setgid: Operation not permitted                                    ✓

# 3. no se pueden crear sockets sin procesar
$ docker exec c ping -c1 10.0.0.1
ping: permission denied (are you root?)                            ✓

# 4. no hay intérprete de órdenes en la imagen final
$ docker run --rm --entrypoint sh registro/tienda@sha256:9f2c… -c id
```

```
executable file not found                                                    ✓

# 5. no hay secretos en el entorno
$ docker exec c sh -c 'tr "\0" "\n" < /proc/1/environ' | grep -c PASSWORD
0                                                                            ✓
```

Y un cierre que sitúa esta clase en el conjunto del programa. La seguridad de un contenedor tiene tres niveles y conviene no confundirlos:

```
qué contiene la imagen        clases 061, 062, 067
qué puede hacer el proceso    esta clase
qué puede alcanzar por red    clases 065, y las de red de las partes 02 a 04
```

Los tres son necesarios y ninguno sustituye a otro. Una imagen impecable ejecutándose en modo privilegiado es un anfitrión comprometido; un contenedor perfectamente acotado con una imagen que lleva una biblioteca vulnerable y acceso de red a todo sigue siendo un problema. Lo que hace manejable el conjunto es que las tres listas son cortas, están escritas y se comprueban solas.

Ejemplo trabajado

CloudShop endurece sus contenedores después de un incidente real: una vulnerabilidad en una dependencia permitió ejecutar código dentro de un servicio. El daño fue mucho mayor de lo que debería, y el análisis explica por qué.

El incidente y por qué escaló.

```
vulnerabilidad          ejecución de código en el servicio de informes
lo que debería haber
  conseguido el atacante  el contenido de ese contenedor
lo que consiguió        credenciales de la base de datos, el fichero de
                        configuración de otro servicio y la clave de la
                        canalización de despliegue
```

Los tres pasos que lo permitieron:

```
# 1. el proceso corría como usuario cero
$ docker inspect informes --format '{{.Config.User}}'
(vacío)

# 2. con la capacidad de ignorar permisos de fichero, leyó ficheros
#    del volumen compartido que no le pertenecían
$ docker inspect informes --format '{{.HostConfig.CapDrop}}'
[]

# 3. las credenciales estaban en variables de entorno
$ docker inspect informes --format '{{json .Config.Env}}' | grep -c PASSWORD
2
```

Corrección 1 — usuario, capacidades y raíz de solo lectura.

```
usuario                                  0 (root)          10001
capacidades                             14 por defecto     ninguna
raíz                                     escribible      solo lectura + tmpfs
sin nuevos privilegios                       no              sí
ficheros del volumen legibles por el proceso  todos     solo los suyos
```

Repetido el mismo ataque en un entorno controlado con la configuración nueva:

```
lectura de ficheros ajenos del volumen      denegada
escritura de un binario en la raíz          denegada
escalada a usuario cero                     denegada
lo conseguido                               el contenido de ese contenedor
```

Exactamente el alcance que debía tener.

Corrección 2 — el agente de canalización en modo privilegiado.

```
$ grep -rn 'privileged' .gitlab-ci.yml .github/workflows/
.github/workflows/publicar.yml:  options: --privileged
```

Estaba ahí desde el primer día porque la construcción de imágenes «lo necesitaba». El agente ejecuta el código de cualquiera que abra un cambio, incluidos colaboradores externos.

```
modo del agente                          privilegiado    sin privilegios
construcción de imágenes            socket del motor     constructor sin demonio
acceso al anfitrión desde el agente      completo           ninguno
```

```
duración de la construcción              1 min 30 s       1 min 40 s
```

Diez segundos más por eliminar el camino más directo del repositorio al anfitrión.

Corrección 3 — el filtro desactivado por una respuesta de internet.

```
$ grep -rn 'seccomp=unconfined' compose.yaml manifiestos/
compose.yaml:  security_opt: ["seccomp=unconfined"]
```

El comentario adyacente decía «necesario, si no falla al arrancar». El fallo era una incompatibilidad entre la biblioteca del sistema de la imagen y la versión del núcleo, corregida en versiones posteriores de ambas.

```
filtro de llamadas al sistema          desactivado       perfil por defecto
causa real del fallo original      incompatibilidad     imagen base actualizada
contenedores con el filtro desactivado      3                0
excepciones documentadas con fecha           0             1, con revisión
```

La excepción restante corresponde a una herramienta de diagnóstico que sí necesita llamadas del sistema poco habituales, y tiene motivo, responsable y fecha.

Corrección 4 — las credenciales que viajaron a un servicio externo.

Al revisar el recolector de errores apareció lo peor del incidente:

```
excepciones enviadas con el entorno completo adjunto     11.400 en 8 meses
credenciales visibles en ellas                            2 distintas
servicio externo con acceso a esos datos                  sí
```

Las credenciales llevaban ocho meses saliendo de la organización en cada excepción, con independencia del ataque.

```
secretos                             variables de entorno   ficheros en memoria
convención                              BD_PASSWORD         BD_PASSWORD_FILE
envío del entorno al recolector            activado          desactivado
credenciales rotadas                           —             las 2, ese día
comprobación automática del entorno       ninguna      en la admisión
```

Corrección 5 — espacios de nombres de usuario en los agentes.

Se adoptó donde más aporta y no en todas partes, con los costes aceptados por escrito:

```
espacios de nombres de usuario                    sí              no, por ahora
motivo                                      ejecutan código      la plataforma no lo
                                            no revisado          soporta todavía
coste asumido                          ajuste de propiedad de     —
                                       los directorios de caché
resultado de una fuga hipotética   usuario sin privilegios  usuario cero
```

La segunda columna quedó como riesgo residual declarado, con la condición de revisión: cuando la plataforma lo soporte.

Resumen del endurecimiento:

```
contenedores como usuario cero                 9 de 11        0 de 11
contenedores con todas las capacidades         11 de 11       0 de 11
raíz de solo lectura                            0 de 11       11 de 11
modo privilegiado                                 1              0
socket del motor montado                          2              0
filtro de llamadas desactivado                    3         1, documentado
secretos en variables de entorno                  7              0
credenciales enviadas a servicios externos     11.400            0
pruebas negativas de endurecimiento             0 de 5         5 de 5
```

La lección que esta clase traslada al resto de la parte 05: el ataque no tuvo éxito por una vulnerabilidad excepcional, sino porque cada capa que debía acotarlo estaba desactivada por una razón razonable en su momento — el usuario cero por comodidad, el modo privilegiado para construir, el filtro por una respuesta de internet y las variables de entorno por ser lo fácil. Las cuatro se corrigen con cuatro líneas de configuración, y la única que costó trabajo real —los espacios de nombres de usuario— es también la única que cambia lo que ocurre el día que falle todo lo demás.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/069-rootless-capabilities-seccomp-y-secretos/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 contenedor-endurecido 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 contenedor-endurecido 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 ejecución de código en un servicio permite leer ficheros de otros	El proceso corre como usuario cero con la capacidad de ignorar permisos de fichero	Usuario numérico distinto de cero, todas las capacidades retiradas y raíz de solo lectura.
El agente de la canalización necesita modo privilegiado para construir imágenes	Se construye a través del socket del motor, que equivale a administrador del anfitrión	Usa un constructor sin demonio y sin privilegios; el coste son segundos de construcción.
Un contenedor solo arranca con el filtro de llamadas desactivado	Incompatibilidad entre la biblioteca del sistema de la imagen y el núcleo, casi siempre ya corregida	Actualiza la base o permite la llamada concreta; desactivarlo entero es una excepción con fecha, no una solución.
Aparecen credenciales en un servicio externo de seguimiento de errores	Los secretos están en variables de entorno y el recolector adjunta el entorno a cada excepción	Pásalos como ficheros en memoria con la convención de sufijo, desactiva el envío del entorno y rota lo expuesto.
Al activar espacios de nombres de usuario dejan de funcionar los montajes del anfitrión	Los identificadores se traducen, así que la propiedad de los directorios ya no coincide	Recalcula la propiedad sumando el desplazamiento de la traducción, como en la clase 064.
Un contenedor con la imagen endurecida sigue alcanzando servicios que no le tocan	El endurecimiento del proceso no acota la red	Los tres niveles son necesarios: contenido de la imagen, permisos del proceso y alcance de red.

■ Seguridad, ética y costo

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

■ Preguntas de comprobación

1. ¿Por qué el usuario cero del contenedor es el usuario cero del núcleo, y qué cambia con espacios de nombres de usuario?
2. Nombra tres capacidades del conjunto por defecto y di qué permite cada una.
3. ¿Qué desactiva exactamente el modo privilegiado, y por qué montar el socket del motor es equivalente?
4. ¿Dónde acaba un secreto pasado como variable de entorno? Enumera al menos cuatro sitios.
5. Escribe tres pruebas negativas de endurecimiento y di qué demuestra cada una.

Referencias

• Linux (2025). capabilities(7) — las piezas del poder del usuario cero y su semántica. <https://man7.org/linux/man-pages/man7/capabilities.7.html>

• Docker (2025). Runtime privilege and Linux capabilities — conjunto por defecto y modo privilegiado. <https://docs.docker.com/engine/containers/run/#runtime-privilege-and-linux-capabilities>

• Docker (2025). Seccomp security profiles — perfil por defecto y consecuencias de desactivarlo. <https://docs.docker.com/engine/security/seccomp/>

• Docker (2025). Rootless mode — espacios de nombres de usuario, limitaciones y configuración. <https://docs.docker.com/engine/security/rootless/>

• NIST (2022). SP 800-190: Application Container Security Guide — modelo de amenazas y controles por capa. <https://csrc.nist.gov/pubs/sp/800/190/final>

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

Descargar: Parte 05 en PDF · Manual integral

Anterior	Índice	Siguiente
← 068 · Límites, health checks y apagado ordenado	Parte 05 · Programa	070 · Diagnóstico de CPU, memoria, red y filesystem →

Evaluación — 069 Rootless, capabilities, seccomp y secretos

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 contenedor-endurecido 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.

070 — Diagnóstico de CPU, memoria, red y filesystem

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 4 Laboratorio: observability · Estado: EXECUTABLECORE

Propósito

Diagnosticar un contenedor que va mal, con dos restricciones que las clases anteriores han creado a propósito: la imagen no tiene intérprete de órdenes y los límites, no el anfitrión, son el techo. De ahí sale un método distinto del de una máquina: se entra con un contenedor efímero que comparte los espacios de nombres, se leen los ficheros del grupo de control en vez del panel del anfitrión, y se usa una señal que casi nadie mira y que mide exactamente lo que importa: cuánto tiempo se pierde esperando por cada recurso.

Resultados de aprendizaje

Al finalizar podrás:

1. Investigar un contenedor sin intérprete de órdenes usando un contenedor efímero que comparte sus espacios de nombres.
2. Leer la información de presión por recurso y usarla como aviso antes del fallo.
3. Distinguir saturación de CPU, de memoria, de disco y de red con la señal propia de cada una.
4. Localizar el consumo real de un proceso en producción sin reiniciarlo ni modificar la imagen.
5. Recorrer la escalera de diagnóstico de red hasta el punto exacto donde se rompe.

Conceptos centrales

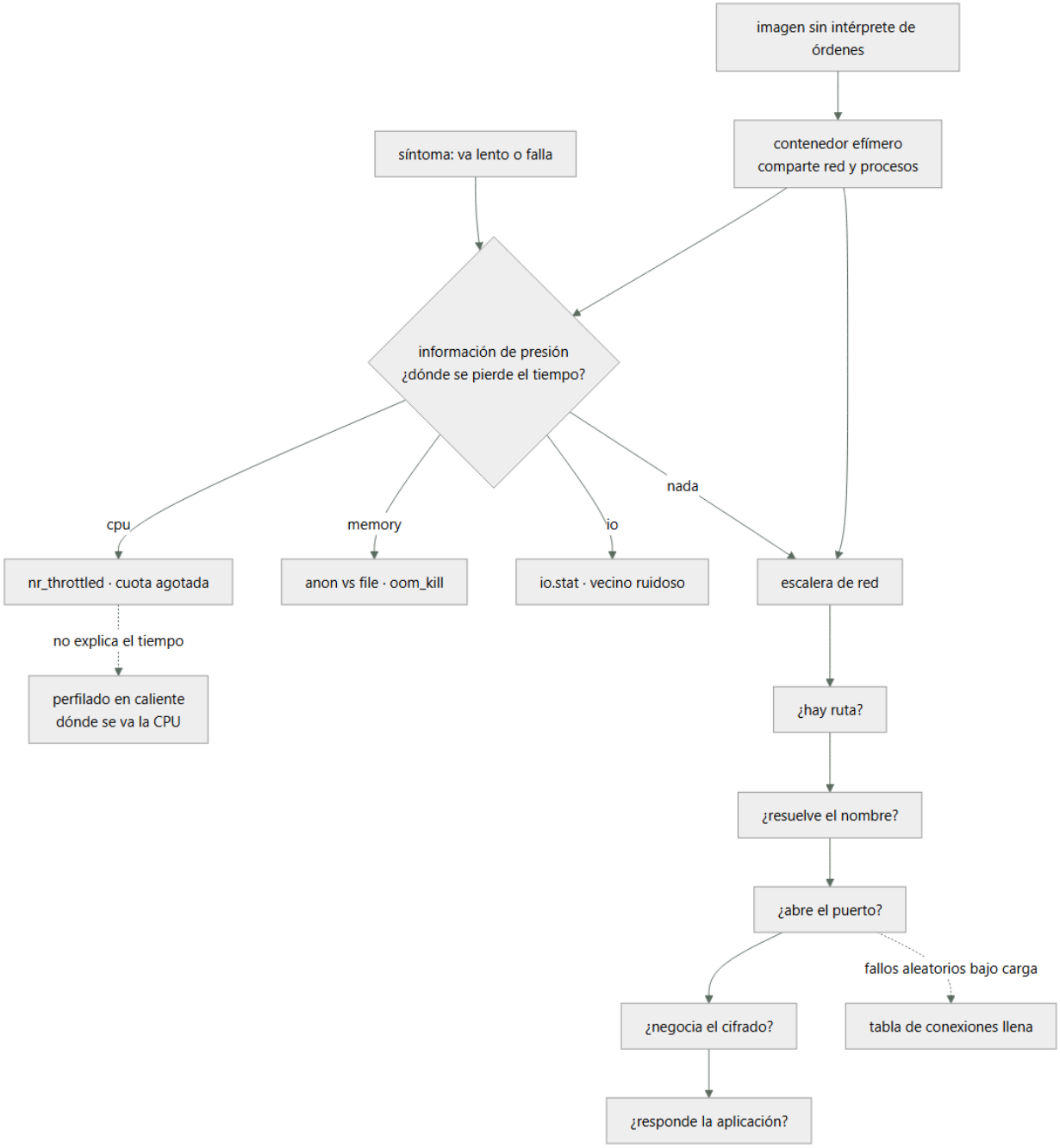
Concepto	Comprensión verificable
contenedor efímero de diagnóstico	Contenedor con herramientas que se une a los espacios de nombres de otro. Permite investigar una imagen mínima sin añadirle nada.
información de presión	Medida del tiempo perdido esperando por CPU, memoria o disco, disponible por grupo de control. Es la señal que se corresponde con lo que sufre el usuario.
saturación frente a utilización	La utilización dice cuánto se usa; la saturación, cuánto se espera. Un recurso al 60 % con cola es peor que uno al 95 % sin ella.
tabla de seguimiento de conexiones	Registro del núcleo con las conexiones traducidas. Al llenarse, se descartan paquetes sin ningún error en la aplicación.
perfilado en caliente	Obtener el consumo por función de un proceso en marcha, sin reiniciarlo ni cambiar la imagen. Responde lo que ninguna traza puede responder.
escalera de red	Secuencia fija —ruta, nombre, puerto, cifrado, aplicación— que localiza en qué peldaño se rompe una conexión en vez de adivinarlo.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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. Entrar sin entrar: el contenedor efímero

Las clases 062 y 069 dejaron imágenes sin intérprete de órdenes, sin gestor de paquetes y sin utilidades. Eso es lo correcto y crea un problema práctico:

```
$ docker exec -it tienda sh
OCI runtime exec failed: exec: "sh": executable file not found
```

La respuesta no es añadir un intérprete a la imagen de producción. Es traer las herramientas en otro contenedor que se una a los espacios de nombres del que se investiga — el modo compartido de la clase 065:

```
$ docker run --rm -it \
    --network container:tienda \
    --pid container:tienda \
    --cap-add SYS_PTRACE \
    nicolaka/netshoot
```

Dentro de ese contenedor se ve la red y los procesos del otro, con todas las herramientas:

```
# ss -ltnp                → los puertos que escucha el proceso investigado
# ps aux                  → sus procesos
# tcpdump -i any port 5432 → su tráfico
# cat /proc/1/environ     → su entorno (clase 069)
```

En una plataforma, el mecanismo tiene nombre propio y funciona igual:

```
$ kubectl debug -it tienda-7f4c9 --image=nicolaka/netshoot \
    --target=tienda --profile=general
```

--target es la parte importante: sin él, el contenedor comparte la red pero no los procesos, y la mitad de las comprobaciones no sirven.

Y desde el anfitrión, cuando ni siquiera eso está disponible, se entra en los espacios de nombres directamente:

```
$ PID=$(docker inspect -f '{{.State.Pid}}' tienda)
$ sudo nsenter -t $PID -n ss -ltnp        # solo la red
$ sudo ls -l /proc/$PID/root/etc/         # su sistema de ficheros, desde fuera
$ sudo cat /proc/$PID/limits
```

La segunda línea es especialmente útil: el sistema de ficheros del contenedor es accesible desde el anfitrión a través de la raíz del proceso, sin necesidad de ejecutar nada dentro. Se pueden leer ficheros de configuración y registros de una imagen que no tiene con qué mostrarlos.

Y una regla que conviene fijar antes de necesitarla: el contenedor de diagnóstico se documenta y se autoriza. Tiene herramientas de red y capacidad de inspeccionar procesos, así que es exactamente lo que un atacante querría. En producción debe estar sujeto a la misma política de admisión que el resto (clase 067) y su uso debe quedar registrado, igual que el acceso justo a tiempo de la clase 046.

2. La presión: la señal que mide lo que duele

Los paneles muestran utilización: porcentaje de CPU, memoria usada, operaciones de disco. Ninguna de esas cifras responde a la pregunta que importa, que es cuánto tiempo se está perdiendo esperando.

El núcleo la publica por grupo de control, y casi nadie la mira:

```
$ cat /sys/fs/cgroup/cpu.pressure
some avg10=34.21 avg60=28.90 avg300=19.44 total=418293991

$ cat /sys/fs/cgroup/memory.pressure
some avg10=0.00 avg60=0.00 avg300=0.00 total=0
full avg10=0.00 avg60=0.00 avg300=0.00 total=0

$ cat /sys/fs/cgroup/io.pressure
some avg10=2.11 avg60=1.80 avg300=1.02 total=88142991
```

Cómo se lee:

```
some   porcentaje de tiempo en que AL MENOS una tarea esperaba por ese recurso
full   porcentaje de tiempo en que TODAS esperaban: nadie avanzaba
avg10  media de los últimos 10 segundos
```

Y por qué es mejor señal que la utilización:

```
CPU al 95 % sin cola          el contenedor está aprovechando su cuota: sano
CPU al 40 % con presión 34 %  hay trabajo esperando: la cuota es el problema
memoria al 80 % sin presión   normal
memoria al 70 % con presión   el núcleo está desalojando caché sin parar:
                              la E/S se va a disparar antes que la memoria falle
```

La última fila es la más valiosa, porque es un aviso previo: la presión de memoria sube minutos antes de la terminación por falta de memoria de la clase 063. Un umbral sobre memory.pressure some avg60 > 10 avisa mientras todavía se puede actuar; el contador de terminaciones avisa después.

Los umbrales que funcionan como punto de partida, y que hay que ajustar por servicio:

```
cpu.pressure    some avg60 > 20 %   la cuota está frenando de forma sostenida
memory.pressure some avg60 > 10 %   el límite está cerca; revisar antes del fallo
io.pressure     some avg60 > 20 %   disco saturado, propio o de un vecino
io.pressure     full avg10 > 10 %   nadie avanza: incidente
```

Y las tres métricas de grupo de control que la clase 063 pedía, ampliadas con estas tres de presión, forman el panel mínimo de cualquier plataforma de contenedores:

```
oom_kill acumulado          cualquier valor > 0
proporción de periodos limitados
procesos frente al tope
presión de CPU, memoria y E/S
```

Seis señales. Con ellas, los incidentes de las clases 063, 064 y 068 se detectan antes de producir daño; sin ellas, se detectan por sus consecuencias.

3. Dónde se va el tiempo: perfilar sin reiniciar

Cuando la presión no explica nada —el contenedor tiene cuota de sobra, memoria libre y disco tranquilo— el problema está dentro del proceso, y ninguna traza distribuida lo va a mostrar porque ocurre dentro de un tramo.

La herramienta es el perfilado, y lo importante es que se puede hacer sobre el proceso en marcha, en producción, sin reiniciar y sin cambiar la imagen.

Si la aplicación expone un punto de perfilado —lo habitual en varios lenguajes— basta con consultarlo:

```
$ kubectl port-forward tienda-7f4c9 6060:6060 &
$ go tool pprof -http=:8080 http://localhost:6060/debug/pprof/profile?seconds=30
$ go tool pprof -http=:8081 http://localhost:6060/debug/pprof/heap
```

Y si no la expone, hay perfiladores que se adjuntan desde fuera al proceso en ejecución:

```
# desde un contenedor efímero que comparte los procesos
$ py-spy top --pid 1
$ py-spy dump --pid 1          # la pila de cada hilo, ahora mismo
```

La segunda orden merece atención porque resuelve un caso concreto que cuesta horas: un proceso colgado. Un volcado de pilas dice exactamente en qué línea está esperando cada hilo, y eso distingue de un vistazo entre un bloqueo mutuo, una espera en una llamada de red sin plazo y un bucle.

Desde el anfitrión, con herramientas del núcleo, funciona para cualquier lenguaje:

```
$ PID=$(docker inspect -f '{{.State.Pid}}' tienda)
$ sudo perf record -F 99 -p $PID -g -- sleep 30
$ sudo perf script | stackcollapse-perf.pl | flamegraph.pl > perfil.svg
```

Y los tres hallazgos típicos, que casi nunca están donde el equipo cree:

```
serialización y deserialización   con frecuencia el mayor consumo de un servicio
compilación repetida de expresiones regulares o de plantillas
criptografía y compresión         a veces correcta, a veces aplicada dos veces
manejo de fechas y zonas horarias  sorprendentemente caro en algunos entornos
```

Y el perfil de memoria, que responde a la pregunta que la clase 063 dejaba abierta cuando la terminación no venía de un montaje en memoria ni de una configuración mal dimensionada:

```
$ go tool pprof -http=:8081 http://localhost:6060/debug/pprof/heap
# o, para comparar dos momentos y ver qué crece
$ curl -s localhost:6060/debug/pprof/heap > t0.pprof
$ sleep 600; curl -s localhost:6060/debug/pprof/heap > t1.pprof
$ go tool pprof -base t0.pprof t1.pprof
```

La comparación entre dos instantes es lo que distingue una fuga de un uso alto pero estable, y es la única forma de demostrar cuál de las dos es. Un servicio que crece 40 MiB cada diez minutos tiene una fuga; uno que se estabiliza en 400 MiB tiene un límite mal puesto.

Y el perfilador continuo de la clase 057 hace todo esto sin que nadie lo pida, que es la diferencia entre encontrarlo durante el incidente y haberlo encontrado antes.

4. La escalera de red y las dos causas que no dan error

El diagnóstico de red se hace en orden fijo, porque cada peldaño depende del anterior y saltarse uno hace perder tiempo:

```
# 1. ¿hay ruta?
# ip route get 10.20.5.7

# 2. ¿resuelve el nombre?
# dig +short bd.interno
# getent hosts bd.interno

# 3. ¿abre el puerto?
```

```
# nc -zv bd.interno 5432

# 4. ¿negocia el cifrado?
# openssl s_client -connect api.proveedor.com:443 -servername api.proveedor.com </dev/null

# 5. ¿responde la aplicación?
# curl -v -m 5 http://bd.interno:8080/readyz
```

Y cuando el resultado es «a veces sí y a veces no», hay dos causas que no producen ningún error en la aplicación y que conviene descartar antes de nada.

La tabla de seguimiento de conexiones llena. El núcleo mantiene un registro de las conexiones traducidas, y ese registro tiene un tamaño. Al llenarse, descarta paquetes en silencio:

```
$ cat /proc/sys/net/netfilter/nf_conntrack_count /proc/sys/net/netfilter/nf_conntrack_max
262144
262144                  ← lleno
$ dmesg | grep -c 'nf_conntrack: table full'
1284
```

El síntoma es exactamente el que más despista: conexiones que fallan al azar bajo carga, en cualquier servicio del nodo, sin patrón. Y la causa suele ser un servicio que abre muchas conexiones cortas —el mismo defecto de las clases 039, 043, 051 y 065—, con lo que la corrección tiene dos partes: subir el tamaño de la tabla y reducir las conexiones en la aplicación.

Retransmisiones y colas del socket. Antes de culpar a la red conviene mirar si el problema es de la máquina:

```
$ ss -ti | grep -A1 ESTAB | grep -E 'retrans|rto' | head
$ ss -ltn '( sport = :8080 )'
Recv-Q Send-Q  Local Address:Port
   511    511        0.0.0.0:8080     ← la cola de aceptación, llena
$ nstat -az TcpExtListenOverflows TcpExtListenDrops
TcpExtListenOverflows           4127
```

Una cola de aceptación desbordada significa que el proceso no acepta conexiones tan rápido como llegan. No es un problema de red: es un problema de la aplicación, y se manifiesta como tiempos de espera del cliente. La corrección puede ser aumentar la cola, y casi siempre es que el proceso está bloqueado en otra cosa — lo que devuelve al perfilado del apartado anterior.

Y la captura de tráfico, que es el último recurso y hay que saber acotarla para que sirva:

```
# desde el contenedor efímero, que comparte la red del investigado
$ tcpdump -i any -nn -c 200 'host 10.20.5.7 and port 5432' -w /tmp/captura.pcap
```

Con el filtro puesto y un número máximo de paquetes. Una captura sin filtro en un servicio con tráfico produce gigabytes ilegibles y añade carga justo cuando el sistema ya está mal.

5. El método, y las dos preguntas que lo cierran

Con las señales y las herramientas anteriores, el diagnóstico de un contenedor sigue siempre el mismo orden:

```
1. ¿el usuario lo nota?        SLO y presupuesto de error (clase 057)
2. ¿qué recurso tiene presión? cpu, memory, io — por grupo de control
3. si hay presión de CPU       ¿es cuota (nr_throttled) o es trabajo real?
   si hay presión de memoria   ¿anónima o caché? ¿oom_kill > 0?
   si hay presión de E/S       ¿este contenedor o un vecino del nodo?
   si no hay presión           el tiempo se va dentro del proceso o en la red
4. dentro del proceso          perfil de CPU o volcado de pilas
   en la red                   escalera de cinco peldaños
5. ¿qué cambió?                despliegue, configuración, volumen de tráfico
```

El paso 3 con presión de E/S merece una nota porque es el único que apunta fuera del contenedor: el disco del nodo lo comparten todos sus contenedores, y un vecino que escribe mucho degrada a los demás sin superar ningún límite propio. Se confirma comparando la presión del contenedor con la del nodo:

```
$ cat /sys/fs/cgroup/io.pressure          # este contenedor
some avg10=18.40
$ cat /proc/pressure/io                   # el nodo entero
some avg10=61.22 full avg10=44.10         ← el problema no es de este contenedor
```

Y las dos preguntas que cierran cualquier diagnóstico y que este programa ha ido repitiendo:

¿Qué señal habría detectado esto antes? Si la respuesta es «ninguna», el resultado del incidente no es solo la corrección: es la métrica que falta. Los seis indicadores de grupo de control de esta clase salieron todos de esa pregunta.

¿Está escrita la consulta que lo encontró? Un diagnóstico que costó tres horas y no dejó una consulta guardada costará tres horas la próxima vez. Es el mismo criterio de las clases 045 y 057: un incidente no es el momento de aprender a consultar.

Y un cierre honesto sobre los límites de esta clase. Todo lo anterior diagnostica un contenedor. Cuando el problema está en la interacción entre varios —una cascada, una dependencia circular, un reparto de carga desequilibrado— hacen falta las trazas distribuidas de las clases 045 y 057 y los conceptos de sistemas distribuidos de la parte 12. La señal de que se ha cruzado esa frontera es concreta: cada componente parece sano por separado y el conjunto no funciona. Ahí, seguir mirando grupos de control no lleva a ninguna parte.

Ejemplo trabajado

CloudShop tiene tres incidentes en dos semanas y ninguno se diagnostica con las herramientas que el equipo usaba. Los tres se resuelven con el método de esta clase, y los tres dejan una métrica que no existía.

Incidente 1 — el percentil 99 se dispara y no hay nada saturado.

```
p99 del catálogo        112 ms → 890 ms, sin cambios de código ni de tráfico
CPU del contenedor      41 %
memoria                 38 % del límite
```

Con las herramientas anteriores, el diagnóstico se detenía ahí. Con la presión:

```
$ kubectl exec catalogo-7f4 -- cat /sys/fs/cgroup/io.pressure
some avg10=19.80 avg60=17.22
$ kubectl debug node/nodo-14 -it --image=busybox -- cat /proc/pressure/io
some avg10=64.10 full avg10=47.33
```

La presión de E/S del nodo era tres veces la del contenedor: el problema estaba fuera. Un trabajo de indexación en otro contenedor del mismo nodo escribía 400 MB/s.

```
límite de E/S en el trabajo de indexación   ninguno       100 MB/s declarado
presión de E/S del nodo                      64 %            9 %
p99 del catálogo                            890 ms          118 ms
métrica de presión en el panel                 no          las tres, por contenedor
```

Y se anotó la lección: un contenedor puede estar dentro de todos sus límites y sufrir por el vecino, y la única señal que lo muestra es la presión comparada.

Incidente 2 — conexiones que fallan al azar bajo carga.

```
síntoma    ~0,8 % de conexiones fallidas en el pico, en TODOS los servicios
           del nodo, sin patrón por destino ni por servicio
```

La escalera de red daba correcto en los cinco peldaños cuando se probaba a mano. La causa apareció en el núcleo:

```
$ kubectl debug node/nodo-14 -it --image=busybox -- \
    sh -c 'cat /proc/sys/net/netfilter/nf_conntrack_count /proc/sys/net/netfilter/nf_conntrack_max;
dmesg | tail -3'
262144
262144
nf_conntrack: table full, dropping packet
```

La tabla estaba llena. El origen: un servicio que abría una conexión por petición hacia la pasarela de pago — el mismo defecto que este programa ha encontrado en las clases 039, 043, 051 y 065, ahora con una quinta manifestación.

```
cliente HTTP                       uno por petición    uno por proceso
entradas en la tabla en el pico        262.144            41.200
tamaño de la tabla                     262.144           524.288
conexiones fallidas en el pico           0,8 %             0,0 %
alerta sobre ocupación de la tabla      ninguna       > 70 % → aviso
```

Las dos correcciones: la de la aplicación es la que resuelve, la del tamaño es el margen.

Incidente 3 — una fuga de memoria en una imagen sin intérprete de órdenes.

```
terminaciones por memoria    1 cada 6 horas
montaje en memoria           64 MiB, correcto (clase 064)
configuración del tiempo
```

```
  de ejecución              acorde al límite (clase 063)
```

Descartadas las dos causas conocidas, quedaba el proceso. Y la imagen no tenía con qué mirar dentro:

```
$ kubectl exec informes-3a1 -- sh
error: exec: "sh": executable file not found
```

Con un contenedor efímero que comparte los procesos:

```
$ kubectl debug -it informes-3a1 --image=nicolaka/netshoot --target=informes
# curl -s localhost:6060/debug/pprof/heap > /tmp/t0
# sleep 600 && curl -s localhost:6060/debug/pprof/heap > /tmp/t1
```

Y la comparación entre los dos instantes:

```
crecimiento en 10 minutos          +38 MiB
concentrado en                     una caché en memoria sin límite ni vencimiento
                                   de plantillas compiladas por cliente

caché de plantillas                sin límite         acotada, con vencimiento
crecimiento de memoria             +38 MiB / 10 min      estable
terminaciones por memoria           1 cada 6 h              0
tiempo hasta el diagnóstico            —              47 min con el método
```

No hizo falta cambiar la imagen, ni añadir un intérprete de órdenes, ni reiniciar el proceso investigado.

Y lo que quedó escrito.

Las tres investigaciones dejaron seis métricas y cuatro consultas que antes no existían:

```
métricas nuevas
  presión de CPU, memoria y E/S por contenedor
  presión de E/S del nodo
  ocupación de la tabla de conexiones
  desbordamientos de la cola de aceptación

consultas guardadas
  comparar presión de contenedor con la del nodo
  volcado de pilas de un proceso colgado
  comparación de dos perfiles de memoria
  escalera de red desde un contenedor efímero
```

Resumen:

```
p99 del catálogo en horas de indexación       890 ms         118 ms
conexiones fallidas en el pico                 0,8 %          0,0 %
terminaciones por memoria                  1 cada 6 h           0
métricas de presión en el panel                 0               4
tiempo medio hasta un diagnóstico de este tipo  horas       < 1 hora
consultas de diagnóstico guardadas              0               4
```

La lección que esta clase traslada al proyecto de la clase 072: los tres incidentes eran invisibles con las métricas de utilización, y los tres tenían una señal disponible en el núcleo que nadie estaba leyendo. La presión mide el tiempo perdido esperando, que es exactamente lo que sufre el usuario, y está publicada por grupo de control desde hace años sin coste. Y la restricción que las clases 062 y 069 crearon —una imagen sin herramientas— no dificultó ningún diagnóstico: las herramientas se traen, no se instalan, y esa es la práctica que hay que ensayar antes del incidente y no durante.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/070-diagnostico-de-cpu-memoria-red-y-filesystem/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 informe-diagnostico 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 informe-diagnostico 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 investigar un contenedor porque la imagen no tiene intérprete de órdenes	La imagen está endurecida a propósito, según las clases 062 y 069	Usa un contenedor efímero que comparta red y procesos del investigado; nunca añadas herramientas a la imagen de producción.
El servicio va lento con todos los recursos dentro de sus límites	La utilización no mide la espera; puede haber presión, propia o del nodo	Lee la información de presión por grupo de control y compárala con la del nodo para descartar un vecino ruidoso.
Conexiones que fallan al azar bajo carga en todos los servicios del nodo	La tabla de seguimiento de conexiones del núcleo está llena y descarta paquetes en silencio	Reduce las conexiones cortas en la aplicación, aumenta el tamaño de la tabla y alerta sobre su ocupación.
Se sospecha una fuga de memoria y no se puede demostrar	Una sola medición no distingue una fuga de un uso alto y estable	Compara dos perfiles de memoria separados en el tiempo; el crecimiento sostenido es la prueba.
Los clientes sufren tiempos de espera y la red está bien	La cola de aceptación del socket se desborda porque el proceso no acepta con la rapidez suficiente	Comprueba los desbordamientos de la cola y perfila el proceso: casi siempre está bloqueado en otra cosa.
Cada componente parece sano y el conjunto no funciona	El problema está en la interacción entre servicios, no dentro de uno	Cambia de herramienta: trazas distribuidas y análisis de dependencias; seguir mirando grupos de control no lleva a ninguna parte.

■ 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 investiga un contenedor sin intérprete de órdenes, y qué opción es imprescindible para ver sus procesos?

2. ¿Qué mide la información de presión que no mide la utilización, y qué umbral usarías como aviso previo a una terminación por memoria?
3. ¿Cómo distingues un problema de E/S propio de uno causado por un vecino del mismo nodo?
4. ¿Qué síntoma produce una tabla de seguimiento de conexiones llena y por qué es difícil de atribuir?
5. ¿Qué demuestra la comparación de dos perfiles de memoria que una sola medición no puede demostrar?

Referencias

• Linux (2025). Pressure Stall Information — some y full por recurso y por grupo de control. <https://docs.kernel.org/accounting/psi.html>

• Kubernetes (2025). Debug running pods with ephemeral containers — kubectl debug y --target. <https://kubernetes.io/docs/tasks/debug/debug-application/debug-running-pod/>

• Brendan Gregg (2020). Systems Performance, cap. 2 — método de utilización, saturación y errores. <https://www.brendangregg.com/systems-performance-2nd-edition-book.html>

• Linux (2025). nfconntrack sysctl — tamaño de la tabla, ocupación y descarte de paquetes. <https://www.kernel.org/doc/html/latest/networking/nfconntrack-sysctl.html>

• Go (2025). Profiling Go programs with pprof — perfiles de CPU y memoria, y comparación entre instantes. <https://go.dev/blog/pprof>

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

Descargar: Parte 05 en PDF · Manual integral

Anterior	Índice	Siguiente
← 069 · Rootless, capabilities, seccomp y secretos	Parte 05 · Programa	071 · Migración de una aplicación legacy a contenedores →

Evaluación — 070 Diagnóstico de CPU, memoria, red y filesystem

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 informe-diagnostico 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.

071 — Migración de una aplicación legacy a contenedores

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 4 Laboratorio: migration · Estado: EXECUTABLECORE

Propósito

Llevar a contenedores una aplicación que no se escribió para ellos, que es el caso real en la mayoría de las organizaciones y donde el error típico no es técnico sino de orden: se empieza escribiendo el Dockerfile. La clase invierte el orden —primero el inventario de supuestos de la aplicación, después la traducción de cada uno, y solo entonces el empaquetado— y presta atención especial al fallo que más daño hace en estas migraciones: una tarea programada dentro de la aplicación que, al haber tres réplicas, se ejecuta tres veces.

Resultados de aprendizaje

Al finalizar podrás:

1. Inventariar los supuestos de una aplicación antes de empaquetarla, con una tabla que sirva de plan.
2. Traducir los patrones heredados habituales —sesión en proceso, ficheros locales, tareas programadas, bloqueos— a su equivalente en contenedores.
3. Migrar por partes con desvío progresivo de tráfico en lugar de un cambio único.
4. Medir antes y después para demostrar que nada empeoró.
5. Decidir con criterio cuándo una aplicación no debe contenerizarse.

Conceptos centrales

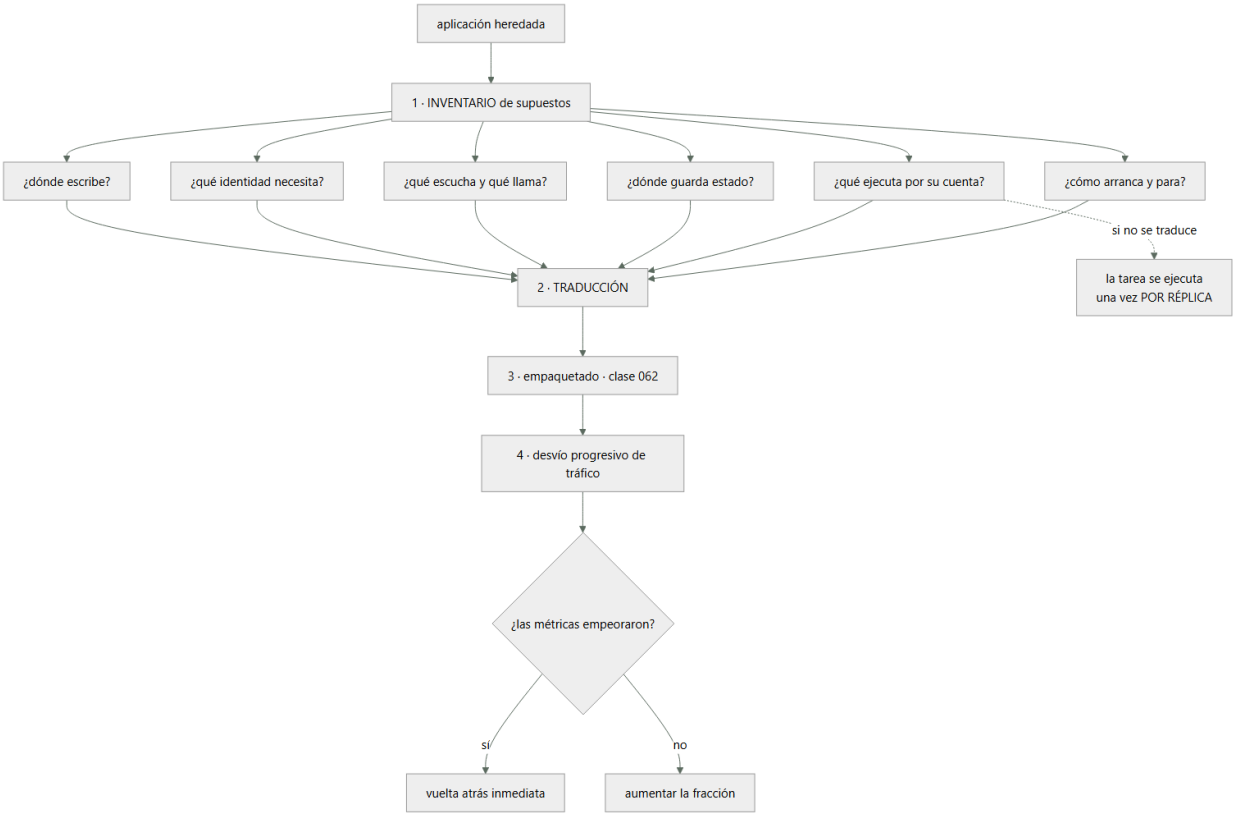
Concepto	Comprensión verificable
inventario de supuestos	Tabla de lo que la aplicación da por hecho: dónde escribe, con qué identidad, qué escucha, dónde guarda estado y qué ejecuta por su cuenta. Es el plan de migración real.
traslado literal	Meter la aplicación en un contenedor sin cambiar nada. Produce un contenedor que se comporta como una máquina y hereda todos sus problemas sin ninguna de sus ventajas.
estrangulamiento progresivo	Migrar por partes desviando una fracción del tráfico, con vuelta atrás inmediata, en lugar de sustituir todo de una vez.
tarea programada en proceso	Trabajo periódico que la aplicación ejecuta por su cuenta. Con varias réplicas se ejecuta una vez por réplica, y ese es el fallo más caro de estas migraciones.
estado adherido	Sesión, caché o bloqueo que vive en la memoria o el disco de una instancia concreta. Es lo que impide reemplazar instancias, que es la propiedad que se quiere ganar.
línea base comparable	Medición del sistema antes de migrar, con el mismo método que se usará después. Sin ella, no se puede afirmar que la migración no empeoró nada.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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 inventario va antes que el Dockerfile

La migración fallida tiene una firma reconocible: empieza escribiendo el Dockerfile, consigue que la aplicación arranque en un día, y a partir de ahí acumula problemas durante meses. Ninguno de esos problemas es del empaquetado.

El orden correcto empieza por seis preguntas, y la tabla que producen es el plan de migración:

Pregunta	Qué hay que averiguar	Clase
¿Dónde escribe?	Rutas, tamaño, si sobrevive a un reinicio	064
¿Con qué identidad?	Usuario, permisos, ficheros con propiedad concreta	069
¿Qué escucha y a qué llama?	Puertos, destinos, si depende de su propia dirección	065
¿Dónde guarda estado?	Sesiones, subidas, cachés, bloqueos	064
¿Qué ejecuta por su cuenta?	Tareas programadas, procesos en segundo plano	esta
¿Cómo arranca y para?	Orden, tiempo, qué hace al recibir la señal	068

Y se responden observando, no leyendo la documentación, que casi siempre está incompleta:

```
# dónde escribe, durante una hora de uso real
$ sudo inotifywait -mr /opt/app /var /tmp -e create,modify --format '%w%f' \
  | sort -u > escrituras.txt

# qué escucha y a qué llama
$ sudo ss -tanp | grep -E 'java|python|httpd'
$ sudo lsof -i -a -p $(pgrep -f app) -n

# qué ejecuta por su cuenta
$ crontab -l -u app; ls -l /etc/cron.d/ /etc/systemd/system/*.timer

# con qué identidad y qué ficheros necesita
$ ps -eo user,pid,cmd | grep app
$ sudo lsof -p $(pgrep -f app) | awk '{print $9}' | grep -v '^/proc' | sort -u
```

El resultado suele contener sorpresas que ningún miembro del equipo actual conocía: un directorio temporal compartido con otro sistema, una llamada a un servicio interno que nadie recordaba, un fichero de bloqueo que garantiza que solo hay una instancia.

Y hay una decisión de alcance que conviene tomar aquí y no después: la base de datos no se migra en el mismo paso. Contenerizar la aplicación y mover el almacén de datos a la vez multiplica las causas posibles de cualquier problema. La aplicación va a contenedores hablando con la misma base de datos de siempre; el motor se aborda después, si es que se aborda.

Y otra que ahorra semanas: contenerizar no es reescribir. La tentación de aprovechar para dividir el monolito, cambiar el marco de trabajo o reordenar el código convierte una migración medible en un proyecto sin final. Primero se mueve tal cual con los supuestos traducidos, se demuestra que nada empeoró, y después se decide qué se cambia.

2. Los seis patrones heredados y su traducción

Casi todo lo que aparece en el inventario cae en uno de seis patrones, y cada uno tiene una traducción conocida y una versión provisional aceptable.

1. Sesión en la memoria del proceso. Es el más común y el que impide reemplazar instancias.

```
traducción correcta   sesión en un almacén externo (clase 042/054), o testigo
                      firmado sin estado en el servidor
provisional aceptable afinidad de sesión en el balanceador, documentada,
                      con la limitación explícita: un reinicio pierde las sesiones
no aceptable          fingir que no existe y descubrirlo al escalar
```

2. Ficheros escritos en disco local. Subidas, informes generados, cachés.

```
subidas de usuario     almacenamiento de objetos (clases 030, 041, 053)
informes temporales    montaje en memoria, con tamaño acotado (clase 064)
cachés reconstruibles  se pierden y se reconstruyen: no necesitan volumen
ficheros compartidos   sistema de ficheros compartido, con la advertencia
                       de semántica de la clase 064
```

3. Tarea programada dentro de la aplicación. Este merece su propio apartado, más abajo, porque es el que produce daño real.

4. Bloqueo por fichero para garantizar una sola ejecución.

```
# el patrón heredado
if [ -f /var/run/proceso.lock ]; then exit 0; fi
touch /var/run/proceso.lock
```

Con varias réplicas, cada una tiene su propio sistema de ficheros, así que el bloqueo no bloquea nada.

```
traducción   bloqueo en un almacén compartido con vencimiento,
             o elección de líder que ofrezca la plataforma
simplificación  a menudo la mejor: hacer el proceso IDEMPOTENTE
                y dejar de necesitar el bloqueo (clases 033, 044, 056)
```

5. Registros en ficheros con rotación.

```
traducción   salida estándar; la plataforma recoge y rota
efecto secundario  desaparece el volumen de registros que se llenaba
trampa       si se deja el fichero Y se añade la salida estándar,
             se paga la ingesta dos veces (clases 045, 057)
```

6. Configuración por fichero con valores por entorno. La clase 062 ya dio la regla: se inyecta al ejecutar. La traducción típica es una plantilla que el punto de entrada rellena desde variables y secretos montados, sin reconstruir nada.

Y dos supuestos que no tienen traducción y hay que detectar pronto, porque cambian la decisión:

```
la aplicación depende de su nombre de máquina o de su dirección
  típico en licencias atadas al equipo y en clústeres antiguos
  → hay que resolverlo con el proveedor antes de seguir

la aplicación necesita módulos del núcleo o hardware concreto
  → contenerizarla exige privilegios que anulan el aislamiento (clase 069)
```

El segundo lleva a la pregunta del último apartado: si contenerizar exige --privileged, la ganancia se ha evaporado.

3. La tarea programada que se ejecuta tres veces

Es el fallo más caro de estas migraciones y aparece semanas después, cuando alguien escala el servicio.

```
en una máquina
  la aplicación arranca un hilo que a las 02:00 factura los pedidos del día
  hay UNA máquina: se ejecuta una vez

en contenedores con 3 réplicas
  cada réplica arranca su hilo
  a las 02:00 se ejecuta TRES veces
```

Y el daño depende de lo que haga la tarea:

```
generar un informe        tres informes idénticos: molesto
enviar correos            tres correos a cada cliente: incidente visible
facturar                  triple facturación: incidente grave
borrar datos antiguos     condición de carrera entre las tres
```

Lo peor es que funciona durante semanas mientras hay una sola réplica, y aparece el día que alguien escala — que suele ser un día de mucho tráfico, es decir, el peor.

Las tres traducciones, en orden de preferencia:

```
1. sacarla de la aplicación
   un recurso de trabajo programado de la plataforma, que ejecuta un contenedor
   aparte con la misma imagen y otro punto de entrada
   → una ejecución, con su registro, su reintento y su alerta

2. elección de líder
   las réplicas se coordinan y solo una ejecuta
   → más complejidad en la aplicación; se justifica si la tarea
     necesita el estado del proceso

3. hacerla idempotente y aceptar varias ejecuciones
   → válido para tareas de limpieza o de reconciliación,
     no para las que producen efectos hacia fuera
```

La primera es casi siempre la correcta, y tiene una ventaja adicional que se descubre después: la tarea pasa a ser visible. Deja de ser un hilo dentro de un proceso que nadie vigila y pasa a tener su propia ejecución, su duración, su código de salida y su alerta cuando falla.

```
# la misma imagen, otro punto de entrada, una sola ejecución
schedule: "0 2 * * *"
concurrencyPolicy: Forbid          # si la anterior sigue, no arranca otra
backoffLimit: 2
template:
  spec:
    containers:
      - image: registro/tienda@sha256:9f2c…
        command: ["/bin/tienda", "facturar-dia"]
```

La línea de la política de concurrencia importa más de lo que parece: sin ella, una ejecución que tarde más de un día se solapa consigo misma, que es la misma condición de carrera con otro origen.

Y la comprobación que hay que hacer antes de escalar, no después:

```
# ¿qué ejecuta la aplicación por su cuenta?
$ grep -rniE 'cron|scheduler|@Scheduled|setInterval|timer|quartz' src/ | head
```

Cualquier resultado de esa búsqueda es un candidato a ejecutarse una vez por réplica. Merece estar en la lista de comprobación de toda migración.

4. Migrar por partes, con vuelta atrás

Sustituir el sistema entero en una noche concentra todo el riesgo en el peor momento posible. La alternativa es desviar tráfico por fracciones, y exige tres condiciones que conviene comprobar antes:

```
1. las dos versiones pueden convivir
   → misma base de datos, mismo esquema, sin migraciones incompatibles
2. la sesión no ata al usuario a una versión
   → o hay afinidad, o la sesión es externa (patrón 1)
3. se puede volver atrás en segundos
   → el desvío es una configuración del balanceador, no un despliegue
```

Con eso, el recorrido es gradual y cada escalón se decide con datos:

```
fase 0   la versión en contenedor recibe 0 % y se comprueba a mano
fase 1   1 %   una hora · se comparan métricas con la línea base
fase 2   10 %  un día
fase 3   50 %  tres días · incluye un pico de tráfico real
fase 4   100 % una semana antes de retirar lo antiguo
fase 5   se retira la infraestructura anterior
```

La fase 5 se olvida sistemáticamente y tiene coste: máquinas encendidas, licencias, parches que alguien sigue aplicando. Debe tener fecha y responsable desde el principio.

Y el criterio para avanzar o retroceder tiene que estar escrito antes de empezar, porque durante la migración todo el mundo tiene prisa:

```
avanzar si, comparado con la línea base:
  tasa de error igual o menor
  percentil 95 y 99 dentro del 10 %
  ningún error nuevo en el agrupador de excepciones
  consumo de recursos explicable

retroceder si cualquiera de los cuatro falla, sin discusión
```

Y la línea base comparable es la parte que se salta con más frecuencia. Hay que medir el sistema antes de tocarlo, con el mismo método que se usará después:

```
p50, p95, p99 por operación, durante una semana completa
tasa de error por operación
coste mensual de la infraestructura actual
duración de un despliegue y de una vuelta atrás
tiempo medio de recuperación de los últimos incidentes
```

Sin esas cinco cifras, la afirmación «la migración salió bien» no se puede sostener ni rebatir. Con ellas, la conversación posterior es sobre números.

Y una precaución de datos que evita el incidente más caro de una migración por fases: mientras las dos versiones conviven, cualquier cambio de esquema tiene que ser compatible en ambos sentidos. Añadir columnas, sí; renombrarlas o eliminarlas, no, hasta que la versión antigua esté retirada. Es la disciplina de expandir y contraer, y la parte 08 la desarrolla.

5. Cuándo no contenerizar

Una clase de migración que no diga cuándo no hacerlo está incompleta. Hay cuatro casos en los que la respuesta honesta es no, y decirlo pronto ahorra meses.

1. La aplicación necesita privilegios que anulan el aislamiento. Módulos del núcleo, acceso directo a dispositivos, manipulación de la red del anfitrión. Si contenerizarla exige modo privilegiado (clase 069), se han conservado todos los inconvenientes y se ha perdido la principal ventaja. Existen mecanismos para casos concretos —dispositivos declarados, capacidades específicas—, y si la lista de excepciones es larga, la respuesta es una máquina.

2. La aplicación se retira en menos de un año. El coste de la migración no se amortiza. Lo defendible es mantenerla donde está, congelar los cambios y dedicar el esfuerzo a lo que la sustituye.

3. La licencia lo prohíbe o lo encarece. Algunas licencias comerciales se calculan por núcleos físicos del anfitrión, no por los asignados al contenedor, con lo que la densidad multiplica el coste en vez de reducirlo. Es una comprobación de una tarde que evita una sorpresa de seis cifras.

4. El equipo no va a poder operarla. Contenerizar mueve la complejidad de la instalación a la operación. Si no hay quien mantenga la plataforma, el sistema estará peor que en una máquina que alguien sabe reiniciar.

Y una quinta situación que no es un no rotundo pero exige una decisión explícita: una aplicación con estado grande y crítico. Un motor de base de datos autogestionado en contenedores es posible y trae los problemas de la clase 064 —un escritor por volumen, semántica de red, ventanas de despliegue—. En la mayoría de los casos, la comparación correcta no es contra la máquina actual sino contra el servicio gestionado que las partes 02, 03 y 04 describieron.

Y la lista de comprobación que cierra la clase y alimenta el proyecto de la clase 072:

- [ ] inventario de los seis supuestos, hecho por observación
- [ ] línea base medida antes de tocar nada, con las cinco cifras
- [ ] base de datos fuera del alcance de este paso

- tareas programadas sacadas de la aplicación
- estado adherido traducido o su limitación documentada
- registros por salida estándar, sin duplicar
- configuración inyectada al ejecutar
- apagado ordenado verificado (clase 068)
- endurecimiento aplicado (clase 069)
- criterios de avance y de vuelta atrás escritos ANTES de empezar
- fecha y responsable para retirar la infraestructura anterior

Once puntos, de los cuales solo dos son de empaquetado. Esa proporción es la lección de la clase: contenerizar una aplicación heredada es casi todo trabajo de análisis y de traducción, y muy poco de Dockerfile.

Ejemplo trabajado

CloudShop migra a contenedores su sistema de facturación, escrito hace nueve años y ejecutándose en dos máquinas. Es la aplicación de la que nadie quiere tocar nada, y el proyecto se planifica en tres fases con la línea base medida antes de empezar.

Fase 0 — el inventario, hecho observando.

```
¿dónde escribe?      /opt/fact/tmp        informes en curso, hasta 400 MB
                     /opt/fact/salida     PDF de facturas, 62 GB acumulados
                     /var/log/fact/       registros con rotación propia
                     /var/run/fact.lock   fichero de bloqueo
¿qué identidad?      usuario 1207, ficheros de salida con esa propiedad
¿qué escucha?        8080
¿a qué llama?        base de datos, pasarela de pago, y un servicio interno
                     de tipos de cambio que NADIE del equipo actual conocía
¿qué ejecuta solo?   un hilo a las 02:00 que factura el día
                     otro a las 03:30 que envía correos
¿cómo para?          no maneja la señal: lo matan
```

Dos hallazgos que no estaban en ninguna documentación: la llamada al servicio de tipos de cambio y los 62 GB de facturas en disco local, que resultaron ser la única copia de documentos con obligación de conservación de siete años.

Fase 0 — la línea base, medida durante una semana.

```
p50 / p95 / p99 de la emisión de factura     180 / 890 / 2.100 ms
tasa de error                                0,4 %
coste mensual de las dos máquinas            410 USD
duración de un despliegue                    45 min, con corte de 12
tiempo medio de recuperación (3 incidentes)  2 h 10 min
```

Fase 1 — traducción de los seis supuestos.

```
informes en curso                /opt/fact/tmp        montaje en memoria, 512 MB
PDF de facturas                  disco local, 62 GB   almacenamiento de objetos
                                                      con inmutabilidad (clase 053)
registros                        ficheros + rotación  salida estándar
fichero de bloqueo               /var/run/fact.lock   eliminado: la tarea sale
                                                      de la aplicación
identidad                        usuario 1207         usuario 1207 (se conserva
                                                      para no reasignar 62 GB)
tareas programadas               2 hilos internos     2 trabajos programados
apagado                          lo matan             señal manejada, 90 s de plazo
```

La migración de los 62 GB se hizo antes de contenerizar nada, con verificación de recuento y de suma de comprobación, y con la aplicación antigua leyendo ya del almacenamiento de objetos durante dos semanas. Mover los datos y cambiar la plataforma son dos cambios, y se hicieron por separado.

Fase 2 — el desvío progresivo, y el incidente que lo justificó.

```
fase 1 · 1 %    1 hora    métricas iguales
fase 2 · 10 %   1 día     p99 mejora a 1.850 ms
fase 3 · 50 %   3 días    ← aquí ocurrió
```

En la fase 3, con tres réplicas y a las 02:00:

```
facturas emitidas esa noche          3 veces las esperadas
clientes afectados                   1.204
tiempo hasta detectarlo              22 min (alerta de volumen de facturación)
tiempo hasta volver atrás             90 s
```

La causa: durante la traducción se sacaron las dos tareas a trabajos programados y nadie eliminó los hilos internos de la aplicación. Con una réplica en las fases 1 y 2 el efecto era una duplicación que pasó desapercibida entre el ruido; con tres réplicas se hizo evidente.

```
$ grep -rniE 'cron|scheduler|@Scheduled|Timer' src/ | wc -l
4
```

Cuatro coincidencias, de las que solo dos se habían tratado.

```
hilos programados dentro de la aplicación      4                0
trabajos programados externos                  2                4
política de concurrencia                    ninguna          Forbid
comprobación de tareas internas             no estaba     en la lista de migración
facturas duplicadas                          3.612               0
```

Las 3.612 facturas duplicadas se anularon en cuatro horas gracias a que la emisión era idempotente por número de pedido —una propiedad que existía por casualidad, no por diseño—. Se hizo obligatoria después.

Y la vuelta atrás, que funcionó porque estaba ensayada: noventa segundos, porque el desvío era una configuración del balanceador y no un despliegue. Ese número era uno de los criterios escritos antes de empezar.

Fase 3 — el resultado, con la misma medición.

```
p50 / p95 / p99 (ms)                 180/890/2.100   165/720/1.780
tasa de error                            0,4 %          0,3 %
coste mensual                          410 USD         288 USD
duración de un despliegue               45 min          3 min 20 s
corte por despliegue                    12 min             0
tiempo medio de recuperación          2 h 10 min       18 min
réplicas posibles                           1              6
facturas en disco local                   62 GB            0
```

La mejora de latencia no era el objetivo y merece explicarse para no atribuirla a los contenedores: vino de eliminar la escritura de PDF en disco local, que competía con el resto. El resto de mejoras sí son de la plataforma.

Y la fase 5, que estuvo a punto de no ocurrir. Las dos máquinas antiguas siguieron encendidas «por si acaso» durante siete semanas más de lo previsto. Tenían fecha y responsable desde el principio, y aun así hubo que reclamarlas dos veces.

```
coste de las máquinas antiguas durante esas 7 semanas   96 USD
parches aplicados a un sistema ya sin tráfico            3
riesgo real                          una configuración divergente que
                                     nadie habría notado hasta necesitarla
```

Resumen de la migración:

```
supuestos heredados sin traducir                 6             0
hilos programados dentro de la aplicación        4             0
estado en disco local                          62 GB           0
corte por despliegue                           12 min          0
duración de un despliegue                      45 min       3 min 20 s
coste mensual                                 410 USD        288 USD
tiempo medio de recuperación                2 h 10 min      18 min
vuelta atrás ensayada                          no         sí, 90 s medidos
```

La lección que esta clase traslada al proyecto de la clase 072: el único incidente serio de la migración —3.612 facturas duplicadas— no vino del empaquetado sino de un supuesto que el inventario había identificado y la traducción cubrió a medias. De los once puntos de la lista de comprobación, solo dos eran de Dockerfile, y el que falló fue de análisis. Contenerizar una aplicación heredada es un ejercicio de traducción de supuestos, y el Dockerfile es el último paso, no el primero.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/071-migracion-de-una-aplicacion-legacy-a-contenedores/
lab.py
```

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

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa plan-modernizacion 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 inventario, dependencias, riesgo y oleadas. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye plan-modernizacion 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 tarea periódica se ejecuta varias veces al escalar el servicio	La tarea vive dentro de la aplicación y cada réplica la arranca	Sácala a un trabajo programado con política de concurrencia, y busca en el código todas las apariciones antes de escalar.
La aplicación funciona con una réplica y falla con dos	Hay estado adherido: sesión en memoria, fichero de bloqueo o caché local	Traduce cada uno a su equivalente externo, o documenta la afinidad como limitación temporal con fecha.
Se descubre que el disco local contenía la única copia de documentos con obligación de conservación	El inventario se hizo leyendo documentación en vez de observando el sistema	Observa escrituras, conexiones y tareas durante un uso real antes de empaquetar nada.
Nadie puede afirmar si la migración mejoró o empeoró el sistema	No se midió la línea base antes de tocar nada	Mide latencias, errores, coste, duración de despliegue y tiempo de recuperación durante una semana previa.
Un cambio de esquema rompe la versión antigua durante la convivencia	Se renombró o eliminó una columna con las dos versiones activas	Solo cambios compatibles en ambos sentidos mientras convivan; expandir primero y contraer después de retirar la antigua.
Contenerizar exige modo privilegiado y accesos al anfitrión	La aplicación necesita módulos del núcleo o hardware concreto	Reconoce que no es candidata: se conservan los inconvenientes y se pierde el aislamiento, que era la ventaja.

■ Seguridad, ética y costo

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

■ Preguntas de comprobación

1. ¿Cuáles son las seis preguntas del inventario y por qué se responden observando en vez de leyendo documentación?
2. ¿Por qué una tarea programada dentro de la aplicación es el fallo más caro de estas migraciones?
3. ¿Qué tres condiciones hacen posible migrar desviando tráfico por fracciones?
4. ¿Qué cinco cifras forman una línea base comparable y por qué hay que medirlas antes?
5. Enumera cuatro situaciones en las que la respuesta correcta es no contenerizar.

Referencias

• Martin Fowler (2004). StranglerFigApplication — sustitución progresiva en lugar de reemplazo único. <https://martinfowler.com/bliki/StranglerFigApplication.html>

• Adam Wiggins (2017). The Twelve-Factor App — configuración, procesos sin estado, registros y procesos administrativos. <https://12factor.net/>

• Kubernetes (2025). CronJob concurrency policy — una sola ejecución y solapamiento. <https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/>

• Microsoft (2025). Migrate applications to containers: assessment — inventario de dependencias y criterios de idoneidad. <https://learn.microsoft.com/en-us/azure/migrate/tutorial-app-containerization-aspnet-kubernetes>

• Sam Newman (2019). Monolith to Microservices, cap. 3 — patrones de convivencia y cambios de esquema compatibles. <https://samnewman.io/books/monolith-to-microservices/>

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

Descargar: Parte 05 en PDF · Manual integral

Anterior	Índice	Siguiente
← 070 · Diagnóstico de CPU, memoria, red y filesystem	Parte 05 · Programa	072 · Proyecto: stack OCI endurecido y observable →

Evaluación — 071 Migración de una aplicación legacy a contenedores

Preguntas

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

Reto verificable

Entrega plan-modernizacion 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.

072 — Proyecto: stack OCI endurecido y observable

Parte: 05 — Contenedores, Docker y OCI Nivel: intermedio · Horas estimadas: 8 Laboratorio: capstone · Estado: EXECUTABLECORE

Propósito

Integrar las once clases anteriores en una pila de contenedores endurecida, observable y demostrable, y calificar la hipótesis que escribió la clase 060. Esta vez la predicción acertó en las tres afirmaciones, incluida la lista exacta de las cuatro fugas — y quedó corta en la última: no reapareció una de las diez leyes, reaparecieron cuatro. De ese exceso salen las dos leyes nuevas que esta parte añade al programa, y una de ellas explica de golpe cuatro incidentes de cuatro clases distintas.

Resultados de aprendizaje

Al finalizar podrás:

1. Trazar cada decisión de la pila hasta la clase que la tomó y la alternativa descartada.
2. Calificar la hipótesis de la clase 060 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 contenedores y medir detección, impacto y recuperación.
5. Entregar un guion de verificación que demuestre cada afirmación de la línea base.

Conceptos centrales

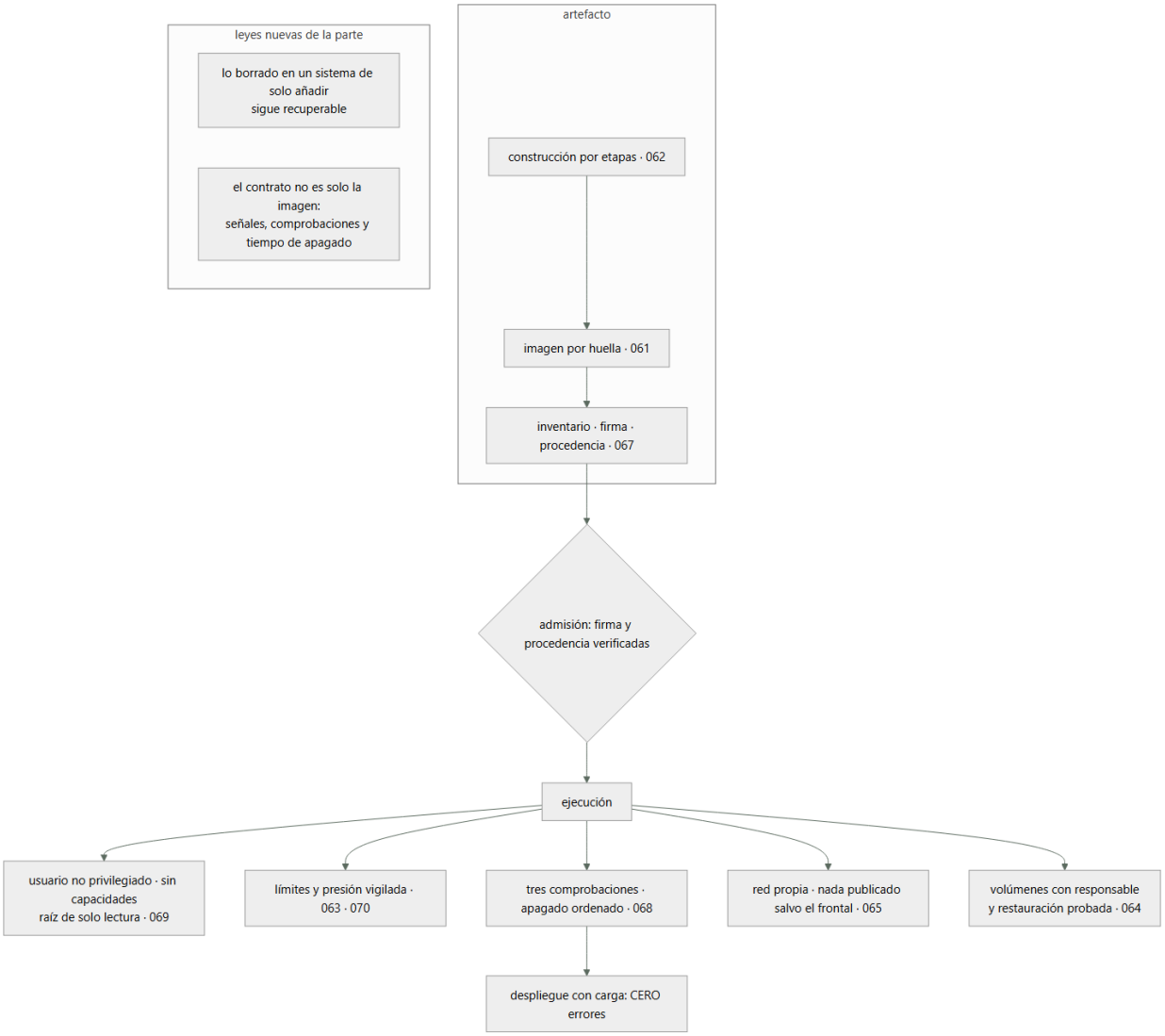
Concepto	Comprensión verificable
sistema de solo añadir	Almacén en el que escribir no sustituye: acumula. Lo «borrado» sigue siendo recuperable, y esa propiedad ha producido cuatro incidentes en cuatro clases distintas del programa.
contrato aplicación-plataforma	Lo que la plataforma espera del proceso más allá de la imagen: qué señales entiende, qué responde en cada comprobación y cuánto tarda en irse. No está en ninguna especificación OCI.
fuga del contrato	Punto donde la portabilidad del contenedor deja de valer. La clase 060 predijo cuatro y las cuatro produjeron incidentes: almacenamiento, red, identidad y ciclo de vida.
prueba negativa de endurecimiento	Comprobación cuyo éxito es que falle. Es la única evidencia aceptable de un control, y esta parte añade cinco a las doce de la parte 04.
presión por recurso	Tiempo perdido esperando, por grupo de control. Es la señal que detecta antes los tres incidentes de esta parte que la utilización no muestra.
hipótesis calificada	Predicción evaluada después con datos, incluida la parte errónea o incompleta. La de la clase 060 acertó y se quedó corta, y lo segundo es lo que aporta.

Modelo mental

Una imagen es un artefacto inmutable; un contenedor es un proceso aislado con límites y dependencias explícitas, no una máquina virtual pequeña.

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 pila entregada y de dónde sale cada decisión

Doce decisiones, cada una con su alternativa descartada y la trampa concreta que evita:

Decisión	Requisito	Alternativa descartada	Trampa que evita
Despliegue por huella	Saber qué se ejecuta (061)	Etiqueta v8	Dos versiones en producción sin desplegar
Índice para varias arquitecturas	Flota mixta (061)	Construir en el portátil	exec format error en media flota
Construcción por etapas	Superficie (062)	Una sola etapa	Intérprete y compilador disponibles para un atacante
Montaje de secreto	Sin rastro (062)	COPY y borrar	Token extraíble de la capa once semanas
Construir una vez y promover	Que lo probado sea lo desplegado (062)	Una imagen por entorno	Fallo que solo se da en producción
Límite de memoria siempre	Radio de impacto (063)	Sin límite	El núcleo elige víctima entre todos los vecinos
Paralelismo acorde a la cuota	Rendimiento (063)	Valor por defecto	32 % de periodos limitados con CPU al 38 %

Decisión	Requisito	Alternativa descartada	Trampa que evita
Volumen con responsable y restauración probada	Datos (064)	Copia sin ensayar	Copia que no restaura, descubierto en el peor momento
Nada publicado salvo el frontal	Exposición (065)	Publicar por comodidad	Base de datos en internet con el cortafuegos activo
Firma verificada en la admisión	Origen (067)	Solo firmar	Dos imágenes sin firma válida en producción
Tres comprobaciones separadas	Continuidad (068)	Una sola	Fallo ajeno de 38 s convertido en 11 min propios
Sin capacidades, sin privilegios, raíz de solo lectura	Contención (069)	Valores por defecto	Una ejecución de código convertida en acceso al anfitrión

Y tres decisiones tomadas en contra de lo que sugiere el hábito, que son las que un revisor cuestionará:

1. La imagen final no tiene intérprete de órdenes, aunque impida `exec`.
   La depuración se resuelve con un contenedor efímero (070), y a cambio
   desaparece una familia entera de cadenas de explotación (062, 069).

2. Al recibir la señal de parada, lo primero que hace el proceso es ESPERAR.
   Es contraintuitivo y es lo que elimina los errores de cada despliegue (068).

3. Compose se conserva en producción para dos servicios, por decisión escrita.
   La alternativa —migrarlo todo— no estaba justificada por su criticidad,
   y lo que sí se corrigió fue el acuerdo de servicio que prometía de más (066).

La tercera es la más incómoda de defender y la más honesta: no todo tiene que migrarse, y lo que no se migra tiene que estar decidido en vez de heredado.

2. La hipótesis de la clase 060, calificada

La clase 060 dejó escrito:

> El contrato del contenedor será portable de verdad y las fugas estarán en los bordes —almacenamiento, red, identidad, arranque y terminación—. Volverá a aparecer al menos una de las diez leyes, con un mecanismo nuevo.

Afirmación 1 — el contrato es portable. CIERTA, y comprobable.

La misma huella se ejecutó sin cambios en cinco entornos:

```
portátil con Docker                    ✓
agente de canalización sin privilegios  ✓
plataforma gestionada de AWS           ✓
Container Apps de Azure                ✓
Cloud Run de Google Cloud              ✓
```

Lo que se conservó intacto: la imagen, el punto de entrada, las variables, el puerto y las señales. Es exactamente la lista que la hipótesis nombró, y es la razón por la que las tres nubes convergieron sin coordinarse.

Afirmación 2 — las fugas están en los bordes. CIERTA, y la lista era exacta.

Las cuatro produjeron incidentes, una por clase:

```
almacenamiento   uid que no coincide, capa de escritura lenta,
                 montaje en memoria contra el límite           064
red              publicación que elude el cortafuegos,
                 unidad máxima, caché de resolución             065
identidad        usuario cero, capacidades, modo privilegiado,
                 secretos en el entorno                         069
ciclo de vida    proceso 1 que ignora señales, ventana de
                 retirada, plazo mal calculado                  068
```

Ninguna de las cuatro está en la especificación OCI, que es precisamente lo que la hipótesis predecía.

Afirmación 3 — reaparecerá al menos una de las diez leyes. CIERTA, y CORTA.

Reaparecieron cuatro, cada una con un mecanismo nuevo:

```
ley 3 · agotamiento de puertos de traducción
  quinta aparición: publicación de puertos y tabla de seguimiento
  de conexiones llena                                  065, 070

ley 4 · el caudal es el mínimo de dos límites
  el contenedor ve el hardware del anfitrión y su tiempo de
  ejecución se dimensiona con esa información                  063

ley 7 · la dependencia caída es funcionamiento normal
  quinta aparición: `depends_on` con condición de salud resuelve
  el arranque y no la vida                              066, 068

ley 2 · el gobierno guarda la puerta y no limpia la casa
  firmar sin verificar: dos imágenes sin firma en producción    067
```

Cuatro de diez, sin buscarlas. Que la predicción se quedara corta es el dato interesante: las leyes no reaparecen a veces, reaparecen sistemáticamente, porque describen propiedades del problema y no del proveedor ni de la tecnología.

3. Dos leyes nuevas, con sus apariciones

De los incidentes de esta parte salen dos comportamientos que cumplen el criterio del programa —observados de forma independiente en varias implementaciones— y que merecen añadirse a las diez de la clase 060.

Ley 11. Lo que un sistema de solo añadir «borra» sigue siendo recuperable.

Cuatro apariciones, en cuatro clases distintas y con cuatro mecanismos que no comparten nada:

```
historial de despliegues de ARM       un secreto en una salida queda
                                      en todas las entradas anteriores      047
estado de Terraform                   guarda en claro todo lo que pasó
                                      por un recurso                        059
capas de una imagen                   `rm` en una capa posterior tapa,
                                      no elimina                            061
historial de construcción             las órdenes ejecutadas viajan
                                      dentro de la configuración            062
```

Y la consecuencia operativa es siempre la misma, con el mismo orden obligatorio:

```
1. rotar el secreto: estuvo expuesto, punto
2. corregir el mecanismo
3. purgar el histórico si el sistema lo permite
```

Invertir el orden —corregir primero y rotar «cuando haya tiempo»— deja la credencial válida en un sitio recuperable. Este programa ha aplicado ese orden cuatro veces y merece estar escrito como norma.

Ley 12. El contrato entre aplicación y plataforma no es solo la imagen.

Incluye tres cosas que ninguna especificación normaliza:

```
qué señales entiende el proceso y qué hace con ellas
qué responde en cada una de las tres comprobaciones
cuánto tarda en irse, y qué hace mientras tanto
```

Apariciones, con tres mecanismos distintos:

```
el proceso 1 no recibe manejadores por defecto: la señal no hace nada   063
vivacidad y disponibilidad confundidas: un fallo ajeno reinicia todo    068
la señal y la retirada de tráfico ocurren en paralelo: hay ventana      068
Compose espera al arranque y no a la disponibilidad                     066
```

Esta ley es la que explica por qué una aplicación puede estar perfectamente contenerizada y aun así producir errores en cada despliegue: la imagen es portable y el comportamiento del proceso ante la plataforma es un contrato aparte, que hay que escribir, verificar y medir.

Y una observación sobre las doce leyes en conjunto que conviene llevarse a la parte 06: ninguna es un defecto de una tecnología. Todas son consecuencias de problemas reales —traducción de direcciones, límites de recursos, entrega no fiable, almacenamiento incremental, terminación de procesos— que cualquier implementación tiene que resolver de alguna manera. Por eso reaparecen, y por eso la lista es un cuestionario útil para la tecnología siguiente.

4. Tres fallos provocados y lo que enseñó cada uno

Fallo 1 — llenar el disco del nodo. Se escribe hasta agotar el sistema de ficheros del anfitrión.

```
detección por presión de E/S           40 s
contenedores afectados                 todos los del nodo
recuperación tras liberar espacio      2 min 10 s
```

Y el hallazgo: el contenedor que llenó el disco no fue el que dejó de funcionar primero. Los registros del motor no tenían límite de rotación, así que el servicio más locuaz consumió el espacio y el primero en fallar fue otro, que intentaba escribir su montaje en memoria — que también se contabiliza contra el nodo cuando este se queda sin memoria para respaldarlo.

```
síntoma observable   un servicio falla; el culpable sigue funcionando
causa                registros sin límite de rotación en el motor
corrección           límite de tamaño y número de ficheros por contenedor,
                     alerta sobre espacio del nodo al 75 %,
                     y presión de E/S del nodo en el panel (070)
```

Fallo 2 — desplegar bajo carga sin apagado ordenado. Se revierte temporalmente la corrección de la clase 068 para medir su valor.

```
errores durante el despliegue          412                  0
peticiones cortadas                     17                  0
duración del despliegue             7 min 20 s          1 min 05 s
```

Y el hallazgo: la medición se hizo sobre el camino HTTP y el equipo dio el resultado por bueno. Al repetirla mirando también el camino asíncrono aparecieron 94 mensajes reentregados por despliegue que la primera medición no veía, porque los consumidores no tienen peticiones que contar.

```
corrección   el criterio de "cero errores en despliegue" se amplía:
             cero errores HTTP y cero reentregas de mensajes
```

Es la misma lección que la clase 060 dejó con el SLI del camino asíncrono, con un mecanismo nuevo: lo que no tiene métrica propia no aparece en la verificación.

Fallo 3 — ejecutar código dentro de un contenedor. Se simula la consecuencia de una vulnerabilidad de dependencia, comparando la configuración anterior con la endurecida.

```
leer ficheros de otros en el volumen       posible          denegado
escribir un binario en la raíz             posible          denegado
escalar a usuario cero                     posible          denegado
leer secretos del entorno                  posible          denegado
alcanzar la base de datos por red          posible          posible  ←
```

Y el hallazgo: la última fila. El endurecimiento del proceso no acota la red, y el contenedor comprometido seguía pudiendo hablar con todo lo que su servicio hablaba. Con las credenciales montadas como fichero, un atacante dentro del proceso las lee igual que la aplicación — que es exactamente el límite que la clase 069 declaró y que el simulacro confirmó.

```
corrección   segmentación de red entre servicios, con la disciplina
             de las clases 039, 051 y 065
lo que NO se corrige aquí   un atacante con el mismo usuario que la
             aplicación lee lo que la aplicación puede leer;
             eso se acota con privilegio mínimo del secreto y
             duración corta (058), no con endurecimiento del proceso
```

Y un cuarto hallazgo transversal, que apareció al preparar los simulacros: el contenedor de diagnóstico de la clase 070 no estaba sujeto a ninguna política de admisión. Tenía herramientas de red, capacidad de inspeccionar procesos y ninguna restricción. Era el hueco más grande que quedaba, y lo había abierto la propia clase que enseñaba a diagnosticar.

```
corrección   imagen de diagnóstico firmada y en la lista de la política,
             uso registrado, y disponible solo mediante una concesión
             temporal como la de la clase 046
```

5. La entrega y la pregunta que abre la parte 06

La entrega, sin conocimiento tácito.

```
imágenes            construidas por etapas, con inventario, firma y procedencia
despliegue          por huella, con admisión que verifica ambas cosas
línea base          22 afirmaciones, cada una con su prueba negativa
verificar.sh        ejecuta las 22 y devuelve código de salida
Compose             fichero único con superposición por entorno
ADR                 12 decisiones con su alternativa descartada
riesgos residuales  4, con responsable y condición de revisión
```

```
procedimientos      3 fallos ensayados, con verificación posterior
panel               6 señales de grupo de control, incluidas las de presión
línea base medida   rendimiento, arranque, apagado y coste
```

Los cuatro riesgos residuales:

```
1. dos servicios permanecen en Compose en una sola máquina, por decisión escrita
2. sin espacios de nombres de usuario en producción: la plataforma no los soporta
3. un contenedor con el filtro de llamadas relajado, documentado y con revisión
4. un atacante con el usuario de la aplicación lee lo que ella lee:
   se acota con duración del secreto, no con endurecimiento
```

La comparación con el punto de partida, medida con el mismo método:

```
construcción en agente efímero              11 min 20 s   1 min 30 s
tamaño de la imagen                            1,2 GB       148 MB
vulnerabilidades críticas y corregibles           41            3
secretos extraíbles de las capas                   1            0
errores por despliegue                          300-900         0
duración del apagado por contenedor             30,2 s       1,1 s
contenedores como usuario cero                  9 de 11      0 de 11
puertos publicados hacia todas las interfaces      4            1
volúmenes con restauración probada              0 de 4       1 de 1
tiempo hasta un diagnóstico de recursos           horas      < 1 hora
pruebas negativas ejecutadas                     0 de 22     22 de 22
```

Y una cifra que resume la parte: de las once clases, nueve produjeron al menos un incidente que llevaba meses ocurriendo sin que nadie lo tratara como tal. Los errores de cada despliegue, la base de datos publicada, el escáner desactivado, el token en la capa, las credenciales en cada excepción. Ninguno era nuevo; lo nuevo fue tener una comprobación que los nombrara.

Y la pregunta que abre la parte 06.

Esta parte deja una pila que funciona en una máquina y que no resuelve lo que la clase 066 enumeró: no reubica, no despliega sin corte, no reparte carga y no decide si el conjunto sirve. Eso es lo que hace un orquestador, y la pregunta correcta no es qué hace sino qué se lleva y qué devuelve:

> De las cuatro fugas de esta parte —almacenamiento, red, identidad y ciclo de vida—, ¿cuántas resuelve Kubernetes y cuántas se limita a renombrar? ¿Y qué problema nuevo introduce el hecho de que su modelo sea declarativo y su consistencia, eventual?

La hipótesis que se escribe ahora, para poder equivocarse de forma comprobable:

> Kubernetes resolverá la reubicación, el despliegue progresivo y el reparto de carga, y no resolverá ninguna de las cuatro fugas: les dará un nombre propio —reclamación de volumen, servicio y política de red, cuenta de servicio, comprobaciones y ciclo de vida— y devolverá a la aplicación exactamente las mismas obligaciones. Y aparecerá una ley nueva propia de su modelo: que un objeto esté aceptado no significa que esté funcionando, y la distancia entre ambas cosas será la causa de sus incidentes característicos.

La parte 06 la califica.

Ejemplo trabajado

Entrega del capstone de la parte 05, con las cifras que se llevan a la parte 06.

Verificación completa. Las 22 afirmaciones de la línea base:

```
$ ./verificar.sh
✓ despliegue por huella, no por etiqueta      0 etiquetas en manifiestos
✓ índice con las dos arquitecturas            amd64 + arm64
✓ imagen sin intérprete de órdenes            executable file not found
✓ ningún secreto en las capas                 0 coincidencias
✓ ninguna variable de entorno con secreto     0 coincidencias
✓ una sola imagen por commit                  huella idéntica en 4 entornos
✓ firma verificada por identidad acotada      ok, y sin firma → rechazada
✓ procedencia del repositorio esperado        ok, y ajena → rechazada
✓ inventario de componentes adjunto           14 de 14 imágenes
✓ escáner activo: crítico y corregible        3 hallazgos, 3 con excepción
✓ usuario numérico distinto de cero           11 de 11
✓ todas las capacidades retiradas             11 de 11
✓ raíz de solo lectura                        11 de 11
```

```
✓ sin modo privilegiado                       0 contenedores
✓ sin socket del motor montado                0 contenedores
✓ límites de memoria en todos                 11 de 11
✓ paralelismo acorde a la cuota               nr_throttled < 1 %
✓ nada publicado salvo el frontal             1 puerto, en bucle local
✓ unidad máxima verificada en la red          1400, ping -M do correcto
✓ apagado en menos de 2 s                     1,1 s medidos
✓ despliegue con carga: cero errores          0 HTTP, 0 reentregas
✓ restauración de volumen verificada          1.284.391 filas · 2 h 40 min
22/22 correctas
```

Línea base medida:

```
rps 982,4 · p50 40,1 ms · p95 93,8 ms · p99 199,4 ms · errores 0
construcción en agente efímero        1 min 30 s
despliegue completo                   1 min 05 s
apagado por contenedor                1,1 s
vuelta atrás (cambio de huella)       22 s
costo mensual de la pila              791 USD
```

Los tres fallos, con lo aprendido en cada uno:

```
disco del nodo lleno            40 s      un servicio caído,      el culpable no
                                          y NO era el culpable    es el primero
                                                                  que falla

despliegue sin apagado          —         412 errores HTTP        lo que no tiene
  ordenado                                + 94 reentregas         métrica propia no
                                          que no se medían        aparece en la
                                                                  verificación

código ejecutado dentro         —         contenido del           el endurecimiento
  del contenedor                          contenedor, y el        del proceso no
                                          alcance de RED intacto  acota la red
```

El hallazgo que justificó el capstone. Al preparar los simulacros, la comprobación de la política de admisión dio un resultado inesperado:

```
$ kubectl auth can-i create pods --subresource=ephemeralcontainers --as=$USUARIO
yes
$ kubectl debug -it tienda-7f4c9 --image=cualquier/imagen --target=tienda
# … arranca sin pasar por la verificación de firma
```

El contenedor de diagnóstico de la clase 070 no pasaba por ninguna política. Podía ser cualquier imagen, de cualquier registro, con herramientas de red y capacidad de inspeccionar procesos, dentro del espacio de nombres de un servicio de producción.

```
síntoma observable   ninguno: todas las demás comprobaciones en verde
consecuencia real    el mecanismo de diagnóstico era el hueco más grande
causa                la política cubría los contenedores del despliegue
                     y no los efímeros
qué lo destapó       preparar el simulacro, no ejecutarlo
```

Corregido y verificado:

```
imágenes permitidas en diagnóstico       cualquiera      1, firmada y en la lista
quién puede usarlo                    9 personas        2, con concesión temporal
uso registrado                           no             sí, en la auditoría
prueba negativa                          ninguna    imagen ajena → rechazada  ✓
```

Es la sexta aparición en el programa de la familia de fallos que la clase 060 identificó como la más cara — un mecanismo que parece estar protegiendo y no cubre lo que se cree — y la primera en la que el hueco lo abrió una buena práctica de una clase anterior.

Se entrega a la parte 06 con:

```
doce leyes observadas, dos de ellas nuevas de esta parte
cuatro fugas del contrato del contenedor, nombradas y con incidente propio
22 afirmaciones y su guion de verificación
12 decisiones con alternativa descartada
4 riesgos residuales con responsable
línea base de rendimiento, construcción, despliegue y apagado
```

Y la hipótesis escrita para la parte 06:

> 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. Su ley nueva será que aceptado no es funcionando.

La lección que esta parte deja al programa: la hipótesis de la clase 060 acertó en todo y se quedó corta en la magnitud — predijo que reaparecería una ley y reaparecieron cuatro. Eso cambia cómo hay que usar la lista: no es un resumen de lo aprendido, es un cuestionario que se le hace a cada tecnología nueva, y su rendimiento es tan alto que conviene aplicarlo antes de tener incidentes en vez de después. De las once clases de esta parte, nueve destaparon un problema que llevaba meses ocurriendo, y ninguno era nuevo: lo nuevo fue tener una comprobación que lo nombrara.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-05-containers-docker-oci/072-proyecto-stack-oci-endurecido-y-observable/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-contenedores 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-contenedores 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 falla cuando el disco del nodo se llena, y no es el que lo llenó	Los registros del motor no tienen límite de rotación y el espacio es un recurso compartido del nodo	Límite de tamaño y número de ficheros por contenedor, alerta de espacio del nodo y presión de E/S en el panel.
La verificación del despliegue da cero errores y hay trabajo duplicado	Solo se mide el camino HTTP; el asíncrono no tiene peticiones que contar	Amplía el criterio: cero errores HTTP y cero reentregas de mensajes durante el despliegue.
Un contenedor endurecido sigue alcanzando servicios que no le corresponden	El endurecimiento acota el proceso, no la red	Segmenta la red entre servicios; los tres niveles —imagen, proceso y red— son necesarios y ninguno sustituye a otro.
El mecanismo de diagnóstico se salta la política de admisión	La política cubría los contenedores del despliegue y no los efímeros	Incluye la imagen de diagnóstico en la lista firmada, restringe quién puede usarla y registra su uso.
Un secreto corregido sigue siendo recuperable semanas después	Vivía en un sistema de solo añadir: historial, estado o capa	Rota primero, corrige después y purga el histórico si el sistema lo permite; el orden no es negociable.
La aplicación está bien contenerizada y cada despliegue produce errores	El contrato con la plataforma —señales, comprobaciones y tiempo de apagado— no está escrito ni verificado	Trátalo como parte del contrato: manejador declarado, tres comprobaciones separadas y despliegue con carga medido.

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 y en qué se quedó corta la hipótesis de la clase 060, y por qué lo segundo es más útil?
2. Enuncia la ley del sistema de solo añadir y cita sus cuatro apariciones con mecanismos distintos.
3. ¿Qué tres cosas incluye el contrato entre aplicación y plataforma que no están en ninguna especificación OCI?
4. Al llenar el disco del nodo falló un servicio que no era el culpable. ¿Por qué, y qué señal lo habría mostrado?
5. ¿Por qué el hueco de seguridad mayor que quedaba lo había abierto una buena práctica de una clase anterior?

Referencias

- Open Container Initiative (2025). Specifications overview — qué normaliza el conjunto y qué queda fuera. <https://opencontainers.org/release-notices/overview/>
- NIST (2022). SP 800-190: Application Container Security Guide — riesgos por capa y contramedidas. <https://csrc.nist.gov/pubs/sp/800/190/final>
- CNCF (2025). Software Supply Chain Best Practices — inventario, firma, procedencia y verificación. <https://github.com/cncf/tag-security/blob/main/community/resources/software-supply-chain-security/secure-software-factory/SecureSoftwareFactoryWhitepaper.pdf>
- Linux (2025). Pressure Stall Information — señales de saturación por recurso y por grupo de control. <https://docs.kernel.org/accounting/psi.html>
- Google (2018). The Site Reliability Workbook, cap. 15 — simulacros de fallo y lo que se aprende de cada uno. <https://sre.google/workbook/postmortem-culture/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Anterior	Índice	Siguiente
← 071 · Migración de una aplicación legacy a contenedores	Parte 05 · Programa	073 · API server, etcd, scheduler, controllers y kubelet →

Evaluación — 072 Proyecto: stack OCI endurecido y observable

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-contenedores 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.