Multi-Cloud Engineering Program

Parte 00 — Fundamentos de computación, redes y Linux

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	00 de 23
Clases	001–012
Evaluaciones	12
Horas estimadas	52

Índice

Parte 00 — Fundamentos de computación, redes y Linux

Programa · Índice completo · Parte 01 →

Descargar: Esta parte en PDF · Manual integral

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

Construir el vocabulario y las destrezas técnicas que la nube da por supuestas.

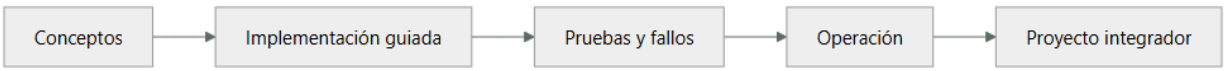
Resultados de la parte

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

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

Secuencia



Clases

ID	Clase	Laboratorio	Horas
001	Computación digital y modelo mental de la nube	foundation	4
002	Terminal, sistema de archivos, procesos y variables de entorno	shell	4
003	Git, GitHub y trabajo reproducible	git	4
004	Python, JSON y automatización mínima	automation	4
005	Redes por capas, TCP/IP, puertos y sockets	network	4
006	DNS, HTTP, HTTPS y TLS de extremo a extremo	network	4
007	Linux: usuarios, permisos, servicios y logs	linux	4
008	Virtualización, hipervisores e imágenes	virtualization	4
009	APIs REST, autenticación y contratos	api	4
010	Responsabilidad compartida y pensamiento de riesgo	security	4
011	Costo, energía, capacidad y medición básica	finops	4
012	Proyecto: servicio local reproducible 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

- How Linux Works — Brian Ward.
- Computer Networking: A Top-Down Approach — Kurose y Ross.
- Pro Git — Chacon y Straub.

001 — Computación digital y modelo mental de la nube

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 4 Laboratorio: foundation · Estado: EXECUTABLECORE

Propósito

Construir el modelo mental que sostiene todo el programa: qué ejecuta realmente una máquina, por qué la latencia y no la velocidad del procesador domina el diseño de un sistema distribuido, y qué cambia exactamente cuando ese cómputo se alquila en lugar de comprarse. Sin este modelo, «la nube» se aprende como un catálogo de nombres comerciales que caduca cada dieciocho meses.

Resultados de aprendizaje

Al finalizar podrás:

1. Situar una operación en la jerarquía de memoria y estimar su coste en órdenes de magnitud, de 1 ns a 150 ms.
2. Aplicar las cinco características esenciales de NIST SP 800-145 para decidir si un servicio es realmente cloud o solo hosting renombrado.
3. Distinguir IaaS, PaaS y SaaS como fronteras de responsabilidad operativa, no como niveles de comodidad.
4. Calcular un presupuesto de latencia extremo a extremo y detectar qué componente lo consume.
5. Explicar por qué la elasticidad cambia la unidad económica de la infraestructura: de activo amortizado a gasto por consumo.

Conceptos centrales

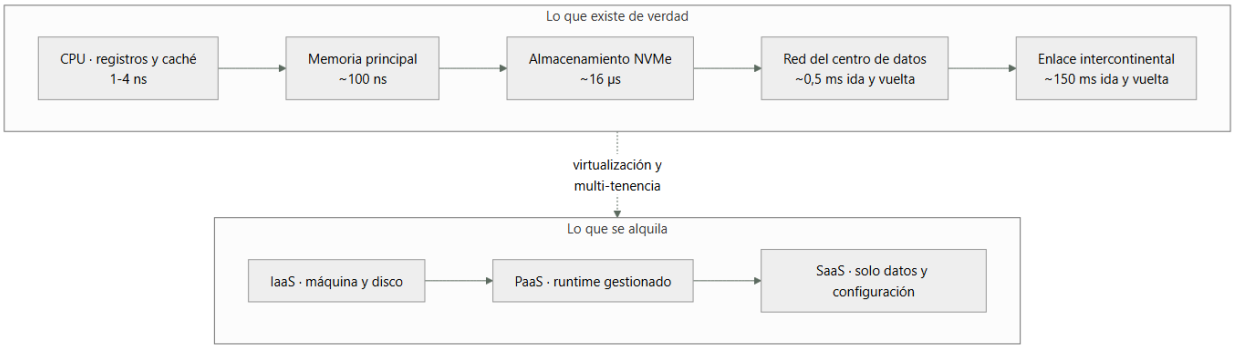
Concepto	Comprensión verificable
latencia	Tiempo entre emitir una petición y recibir la primera respuesta. Está acotada por la velocidad de la luz en fibra (200.000 km/s), así que ninguna optimización de software reduce el mínimo físico entre dos ciudades.
ancho de banda	Volumen transferido por unidad de tiempo. Se compra; la latencia no. Añadir ancho de banda no acelera una petición pequeña, solo permite más peticiones simultáneas.
elasticidad	Capacidad de aprovisionar y liberar recursos en minutos siguiendo la demanda. Es lo que convierte capacidad ociosa en coste evitado; sin ella, la nube es un centro de datos con factura mensual.
multi-tenencia	Varios clientes compartiendo el mismo hardware con aislamiento lógico. Explica a la vez el precio (se amortiza entre muchos) y la clase de riesgo (vecino ruidoso, escapes del hipervisor).
modelo de servicio	Dónde se traza la línea entre lo que administra el proveedor y lo que sigue siendo tuyo. IaaS te deja el sistema operativo; PaaS, el runtime; SaaS, solo los datos y la configuración.

Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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 jerarquía de memoria manda sobre el resto del diseño

Un procesador moderno ejecuta del orden de 10■ instrucciones por segundo y por núcleo, pero pasa la mayor parte del tiempo esperando datos. Los órdenes de magnitud que hay que tener memorizados, popularizados por Jeff Dean en Latency Numbers Every Programmer Should Know:

```
referencia a caché L1                    ~1 ns
referencia a caché L2                    ~4 ns
referencia a memoria principal         ~100 ns   (100x L1)
lectura aleatoria en SSD NVMe           ~16 µs   (160x RAM)
ida y vuelta dentro del centro de datos ~0,5 ms  (31x SSD)
búsqueda en disco mecánico               ~2 ms
ida y vuelta California - Países Bajos  ~150 ms  (300x centro de datos)
```

Entre el primer y el último renglón hay un factor de 150 millones. Esto tiene una consecuencia de diseño que se repetirá en las 288 clases: mover el cómputo hacia los datos casi siempre gana a mover los datos hacia el cómputo. Cuando en la parte 16 se estudien CDN y edge, y en la parte 12 la colocación de réplicas, la razón será siempre esta tabla.

2. La latencia tiene un suelo físico y el ancho de banda no

La luz viaja a 299.792 km/s en el vacío, pero en fibra óptica el índice de refracción la frena a unos 200.000 km/s. Madrid–Nueva York son 5.760 km en línea recta:

```
mínimo teórico ida     = 5.760 km / 200.000 km/s = 28,8 ms
mínimo teórico ida+vuelta                        = 57,6 ms
medido en la práctica                          ~ 85-100 ms
```

La diferencia entre 57,6 y 90 ms son enrutamiento no geodésico, colas en routers y conmutación. Puedes optimizar esa parte; los 57,6 ms no. Ninguna inversión hace que una petición Madrid–Nueva York baje de 58 ms. Por eso la respuesta a un problema de latencia rara vez es un servidor más rápido: es una réplica más cerca, una caché, o menos idas y vueltas.

El ancho de banda se comporta al revés: es una mercancía que se compra. Duplicar el caudal no acorta una petición de 2 KB, pero permite atender el doble de ellas.

3. Qué define a la nube según NIST, y qué no

La definición operativa la fija el NIST SP 800-145 (Mell y Grance, 2011) con cinco características esenciales. Un servicio que no cumple las cinco es hosting, por mucho que se anuncie como cloud:

Característica	Prueba concreta
Autoservicio bajo demanda	¿Puedes aprovisionar sin abrir un ticket ni hablar con nadie?
Acceso amplio por red	¿Se consume por protocolos estándar desde cualquier cliente?
Agrupación de recursos	¿El proveedor reasigna capacidad entre clientes de forma opaca?
Elasticidad rápida	¿Escala y libera en minutos, no en semanas?
Servicio medido	¿Se factura por consumo observable y auditable?

La quinta es la que más cuesta interiorizar y la que más aparece en la factura: si el recurso se mide, cada decisión de arquitectura es también una decisión de coste. Un bucle mal escrito en un centro de datos propio es capacidad desperdiciada; el mismo bucle en cloud es una línea en la factura del mes siguiente.

4. Los modelos de servicio son fronteras de responsabilidad

IaaS, PaaS y SaaS no son «niveles de facilidad». Son dónde se corta la responsabilidad operativa, y esa línea determina a quién despiertan de madrugada:

Capa	On-premise	IaaS	PaaS	SaaS
Datos y acceso	tú	tú	tú	tú
Aplicación	tú	tú	tú	proveedor
Runtime y middleware	tú	tú	proveedor	proveedor
Sistema operativo	tú	tú	proveedor	proveedor
Virtualización, servidores, red física	tú	proveedor	proveedor	proveedor

Observa la primera fila: los datos y el control de acceso nunca cambian de dueño. Ese es el núcleo del modelo de responsabilidad compartida que se estudia en la clase 010, y el origen de la mayoría de incidentes públicos atribuidos a «fallos del proveedor» que en realidad fueron configuraciones del cliente.

5. Elasticidad: el cambio económico, no el técnico

La virtualización existe desde los años sesenta (CP-40 de IBM, 1967). Lo que la nube añadió no fue la técnica sino el modelo económico: pasar de CAPEX amortizado a OPEX medido.

Supón un servicio con pico de 100 servidores durante 3 horas al día y 20 el resto:

```
Compra:  100 servidores × 24 h × 30 días = 72.000 servidor-hora contratadas
Uso real: (100 × 3 + 20 × 21) × 30       = 21.600 servidor-hora útiles
utilización = 21.600 / 72.000 = 30 %
```

En compra pagas el 100 % y usas el 30 %. Con elasticidad pagas cerca del 30 %, a un precio unitario mayor. El punto de equilibrio depende de esa relación, no de la moda: si tu carga es plana y predecible, comprar suele salir más barato. Multi-cloud y elasticidad se justifican por requisitos medibles, nunca por defecto.

Ejemplo trabajado

CloudShop necesita responder una página de producto en menos de 300 ms al percentil 95. El equipo desglosa el presupuesto de latencia para un usuario en Santiago de Chile y un despliegue en la región us-east-1 (Virginia), a unos 7.400 km:

```
resolución DNS (con caché fría)              40 ms
handshake TLS 1.3 (1 RTT sobre ~120 ms)     120 ms
petición HTTP ida y vuelta                  120 ms
consulta a base de datos en la misma región   8 ms
renderizado en servidor                      25 ms
-------------------------------------------------
total                                       313 ms   ✗ incumple el SLO
```

El componente dominante no es la aplicación —33 ms entre consulta y render— sino los 240 ms de red. Optimizar el código no salva el SLO: aunque el render bajara a 0 ms, quedarían 288 ms.

Dos intervenciones sobre la red:

```
CDN en Santiago para el HTML (RTT 10 ms)
  DNS cacheado                    2 ms
  TLS al borde (1 RTT sobre 10)  10 ms
  HTTP al borde                  10 ms
  origen solo para datos (1 RTT)120 ms
  consulta + render               33 ms
  ---------------------------------
  total                         175 ms   ✓ cumple
```

La decisión no fue «usar un CDN» sino reconocer que 240 de los 313 ms eran físicos y solo se atacan acercando el punto de terminación. El coste añadido del borde se justifica contra los 138 ms recuperados; si el SLO hubiera sido 500

ms, el gasto no tendría defensa.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/001-computacion-digital-y-modelo-mental-de-la-nube/lab.py
```

El laboratorio selecciona el motor de práctica foundation 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-de-componentes 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 mapa con fronteras, entradas, salidas y supuestos. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye mapa-de-componentes 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 sistema es lento y se responde comprando instancias más grandes	Se confundió latencia con capacidad de cómputo	Desglosa el presupuesto de latencia por componente antes de escalar; si domina la red, más CPU no cambia nada.
«Migramos a la nube» pero la factura supera al centro de datos anterior	Se replicó una carga plana sin usar elasticidad	Calcula la utilización real; si es estable y alta, la nube solo compensa por otros motivos (alcance, servicios gestionados, continuidad).
Se asume que el proveedor protege los datos	Se leyó el modelo de servicio como nivel de comodidad y no como frontera de responsabilidad	Sitúa cada capa en la tabla de responsabilidad; datos y acceso siguen siendo tuyos en IaaS, PaaS y SaaS.
Una prueba local va rápida y en producción no	En local todo cabe en RAM y la red es de 0 ms	Mide con las latencias reales entre zonas y regiones; la jerarquía de memoria del entorno de pruebas no es la de producción.
Se llama cloud a un servidor alquilado por meses	No se aplicaron las cinco características de NIST	Comprueba autoservicio, elasticidad y medición; si faltan, las decisiones de coste y escala no aplicarán.

■ 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. ¿Cuántos órdenes de magnitud separan una referencia a caché L1 de una ida y vuelta intercontinental, y qué decisión de arquitectura se deriva de esa diferencia?
2. Un servicio tarda 400 ms y el equipo propone duplicar la CPU. ¿Qué medición pedirías antes de aprobar el gasto?
3. ¿Cuál de las cinco características de NIST es la que convierte cada decisión técnica en una decisión de coste, y por qué?
4. En un despliegue PaaS, ¿qué sigue siendo responsabilidad tuya y qué pasa a ser del proveedor?
5. Una carga usa el 85 % de su capacidad las 24 horas. ¿Qué argumento económico queda a favor de la nube, si es que queda alguno?

Referencias

- Mell, P. y Grance, T. (2011). The NIST Definition of Cloud Computing, SP 800-145. National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.800-145>
- Dean, J. (2009). Latency Numbers Every Programmer Should Know — cifras de referencia de la jerarquía de memoria. <https://static.googleusercontent.com/media/research.google.com/en//people/jeff/stanford-295-talk.pdf>
- Kurose, J. y Ross, K. (2021). Computer Networking: A Top-Down Approach, 8.ª ed., cap. 1.4 — retardo, pérdida y throughput.
- Barroso, L., Hölzle, U. y Ranganathan, P. (2018). The Datacenter as a Computer, 3.ª ed. — economía y utilización del centro de datos. <https://doi.org/10.2200/S00874ED3V01Y201809CAC046>
- Gray, J. (2008). Distributed Computing Economics. ACM Queue 6(3) — por qué conviene mover el cómputo hacia los datos. <https://doi.org/10.1145/1394127.1394131>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 00 en PDF · Manual integral

Anterior	Índice	Siguiente
Inicio del programa	Parte 00 · Programa	002 · Terminal, sistema de archivos, procesos y variables de entorno →

Evaluación — 001 Computación digital y modelo mental de la nube

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-de-componentes 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.

002 — Terminal, sistema de archivos, procesos y variables de entorno

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 4 Laboratorio: shell · Estado: EXECUTABLECORE

Propósito

Dominar la terminal como interfaz primaria de operación cloud: qué es realmente un proceso, cómo el kernel le entrega su entorno, y por qué los descriptores de fichero y las variables de entorno explican la mitad de los incidentes de despliegue. Toda la automatización posterior —contenedores, CI/CD, runbooks— es este modelo repetido a escala.

Resultados de aprendizaje

Al finalizar podrás:

1. Describir el ciclo fork → exec → wait y qué hereda un proceso hijo de su padre.
2. Predecir qué variables de entorno ve un proceso según cómo se lanzó, y por qué un servicio systemd no ve tu .bashrc.
3. Redirigir stdout y stderr por separado y explicar por qué 2>&1 >f no equivale a >f 2>&1.
4. Interpretar un código de salida y una señal de terminación para diagnosticar sin adivinar.
5. Componer una tubería que resuelva un problema real de operación sin scripts intermedios.

Conceptos centrales

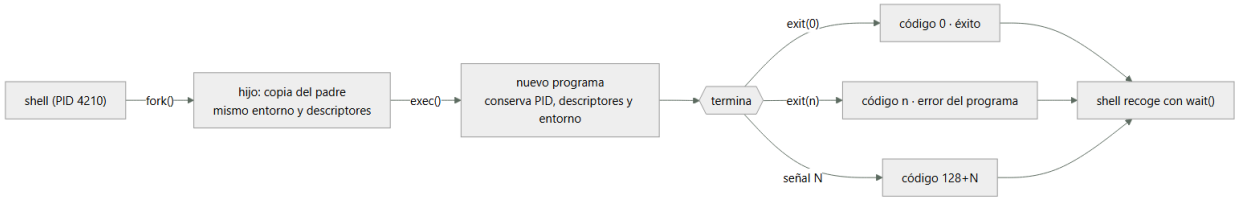
Concepto	Comprensión verificable
proceso	Instancia en ejecución de un programa, con su propio espacio de direcciones, tabla de descriptores y entorno. El kernel lo identifica por PID y lo relaciona con su padre por PPID.
descriptor de fichero	Entero que indexa la tabla de ficheros abiertos del proceso. Por convención 0 es stdin, 1 stdout y 2 stderr; el resto se asignan por orden. Redirigir es duplicar entradas de esa tabla, no mover datos.
variable de entorno	Par clave-valor copiado del padre al hijo en el momento del exec. La copia es unidireccional: un hijo no puede modificar el entorno del padre, lo que explica por qué cd debe ser un builtin del shell.
código de salida	Entero de 0 a 255 que devuelve un proceso al terminar. 0 significa éxito; 1-125 son errores del programa; 126 y 127 indican no ejecutable y no encontrado; 128+N significa terminado por la señal N.
señal	Notificación asíncrona del kernel a un proceso. SIGTERM (15) pide terminar y puede atraparse; SIGKILL (9) no es atrapable. Esa diferencia es la base del apagado ordenado en contenedores.

Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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

■ Flujo de razonamiento



Desarrollo

1. Un proceso nace por copia y se transforma por reemplazo

En UNIX no existe «lanzar un programa» como operación única. Son dos llamadas:

1. fork() duplica el proceso actual. El hijo recibe una copia del espacio de direcciones, la tabla de descriptores y el entorno. Devuelve 0 en el hijo y el PID del hijo en el padre; ese es el único modo que tiene cada uno de saber quién es.
2. exec() reemplaza la imagen del proceso por otro programa conservando el PID, los descriptores abiertos y el entorno.

```
$ echo $$          # PID del shell actual
4210
$ bash -c 'echo $$; echo $PPID'
4877               # el hijo tiene PID nuevo
4210               # y recuerda a su padre
```

La consecuencia práctica aparece en cuanto se automatiza: un proceso solo puede heredar hacia abajo. Por eso cd no puede ser un binario —cambiaría el directorio de un hijo que muere inmediatamente— y por eso exportar una variable en un script no la deja disponible en la terminal que lo invocó.

2. El entorno se copia en el exec, no se consulta en vivo

El entorno es un vector de cadenas CLAVE=valor que se pasa al exec. Se copia una vez, en ese instante. Un proceso que lleva tres días corriendo tiene el entorno del momento en que arrancó, aunque el fichero de configuración haya cambiado.

```
$ VAR=solo_este_comando env | grep VAR    # prefijo: solo para esa invocación
VAR=solo_este_comando
$ VAR=local; env | grep VAR               # sin export: no llega al hijo
$ export VAR=heredable; env | grep VAR    # con export: sí llega
VAR=heredable
```

Esto explica el incidente más repetido en despliegues: un servicio gestionado por systemd no lee /.bashrc ni /.profile. Esos ficheros los interpreta el shell al iniciar sesión de forma interactiva, y systemd no arranca un shell de login. El comando funciona a mano y falla como servicio, con la misma imagen y el mismo binario. La variable hay que declararla en la unidad (Environment= o EnvironmentFile=), y en contenedores en el ENV del Dockerfile o el manifiesto.

3. Redirección: se duplican descriptores, y el orden importa

> no «envía» la salida a ningún sitio: hace que el descriptor 1 apunte al mismo fichero abierto que otro. Como es una asignación secuencial, el orden cambia el resultado:

```
# stderr se copia al destino ACTUAL de stdout (la terminal) y luego stdout va al fichero
$ comando 2>&1 > salida.log      # errores a la terminal, salida al fichero

# stdout va al fichero y DESPUÉS stderr se copia a ese mismo destino
$ comando > salida.log 2>&1      # ambos al fichero
```

Es la causa clásica de un log que «pierde» los errores. En Bash moderno, &> fichero hace lo correcto sin ambigüedad.

La separación stdout/stderr no es cosmética: es un contrato. stdout lleva el resultado destinado a la siguiente etapa de la tubería; stderr lleva diagnóstico destinado al operador. Un programa que escribe avisos en stdout rompe cualquier tubería que lo consuma.

4. Códigos de salida y señales: el lenguaje del diagnóstico

El código de salida es el único canal estructurado que un proceso tiene para decir qué pasó. Sus rangos están convenidos:

Código	Significado	Aparece cuando
0	Éxito	Todo bien
1-125	Error del programa	Fallo de lógica o validación
126	Encontrado pero no ejecutable	Falta el bit de ejecución
127	Orden no encontrada	PATH incorrecto — típico en contenedores
128+N	Terminado por la señal N	137 = 128+9 (SIGKILL), 143 = 128+15 (SIGTERM)

El 137 merece memorizarse: en Kubernetes casi siempre significa que el contenedor excedió su límite de memoria y el OOM killer lo mató. En la parte 06 aparecerá con ese nombre; aquí ya sabes de dónde sale el número.

La diferencia entre SIGTERM y SIGKILL define el apagado ordenado: el orquestador envía SIGTERM, espera un plazo de gracia (30 s por defecto en Kubernetes) y solo entonces envía SIGKILL. Un proceso que ignora SIGTERM pierde las conexiones en vuelo.

5. Tuberías: procesos concurrentes, no secuenciales

a | b no ejecuta a y después b. Arranca ambos a la vez y conecta el descriptor 1 de a con el 0 de b mediante un búfer del kernel de 64 KB. Cuando el búfer se llena, el escritor se bloquea; cuando se vacía, el lector se bloquea. Es control de flujo gratuito, y permite procesar ficheros mayores que la memoria disponible.

```
# Los cinco procesos que más memoria residente consumen
$ ps -eo pid,rss,comm --sort=-rss | head -6

# Las 10 IP con más peticiones en un log de acceso
$ awk '{print $1}' access.log | sort | uniq -c | sort -rn | head -10
```

El código de salida de una tubería es el del último comando, lo que oculta fallos intermedios. En scripts de operación esto es una trampa seria:

```
$ false | true; echo $?
0                             # el fallo desaparece
$ set -o pipefail
$ false | true; echo $?
1                             # ahora se propaga
```

set -euo pipefail al principio de cada script de despliegue evita que un fallo silencioso se declare éxito.

Ejemplo trabajado

Un despliegue de CloudShop funciona a mano y falla como servicio. El binario es el mismo y la imagen también. El operador ejecuta el diagnóstico en orden:

```
$ ./cloudshop-api
escuchando en :8080                    # a mano funciona

$ sudo systemctl start cloudshop
$ systemctl show cloudshop -p ExecMainStatus
ExecMainStatus=127
```

127 = orden no encontrada. No es un fallo de la aplicación: es que algo del PATH no está. Se compara el entorno de ambos contextos:

```
$ echo $PATH
/home/ops/.local/bin:/usr/local/bin:/usr/bin:/bin

$ sudo systemctl show cloudshop -p Environment
Environment=
$ sudo systemd-run --quiet --pipe /usr/bin/env | grep ^PATH
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

La diferencia es /home/ops/.local/bin, que el shell interactivo añade desde /.profile y systemd no. El binario auxiliar que invoca el servicio vive ahí.

Se corrige declarando la dependencia en la unidad, no exportando en un perfil:

```
[Service]
Environment="PATH=/usr/local/bin:/usr/bin:/bin"
ExecStart=/opt/cloudshop/bin/cloudshop-api
```

```
$ sudo systemctl restart cloudshop; systemctl show cloudshop -p ExecMainStatus
ExecMainStatus=0
```

La lección no es el arreglo sino el método: el código 127 acotó el problema a resolución de ruta antes de leer una sola línea de la aplicación. Un diagnóstico sin ese dato habría empezado por los logs de la aplicación, donde no había nada que ver.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/002-terminal-sistema-de-archivos-procesos-y-variables-de-entorno/lab.py
```

El laboratorio selecciona el motor de práctica shell 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 bitacora-de-comandos 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 bitácora reproducible de comandos y resultados. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye bitacora-de-comandos 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 comando funciona en la terminal y falla como servicio o en el contenedor	El entorno interactivo carga perfiles que systemd y Docker no leen	Declara las variables en la unidad (Environment=) o en el ENV de la imagen; nunca dependas de .bashrc.
El fichero de log no contiene los errores	Se escribió 2>&1 > f, que copia stderr al destino anterior de stdout	Usa > f 2>&1 o &> f; el orden de la redirección es una asignación secuencial.
Un script de despliegue termina en éxito pese a que un paso falló	El código de salida de una tubería es solo el del último comando	Empieza los scripts con set -euo pipefail.
El contenedor muere con código 137 y no hay error en la aplicación	128+9: fue SIGKILL, casi siempre el OOM killer por exceder el límite de memoria	Revisa el límite de memoria y el consumo real; el problema es de recursos, no de código.
El servicio pierde peticiones en vuelo en cada despliegue	El proceso no atrapa SIGTERM y se le aplica SIGKILL al agotar el plazo de gracia	Atrapa SIGTERM, deja de aceptar conexiones nuevas y drena las abiertas antes de salir.

■ 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 cd tiene que ser un builtin del shell y no puede ser un binario en /usr/bin?
2. Un proceso lleva tres días ejecutándose y alguien cambia una variable en /etc/environment. ¿La ve el proceso? ¿Por qué?
3. ¿Qué diferencia práctica hay entre comando 2>&1 > f y comando > f 2>&1?
4. Un contenedor termina con código 143 y otro con 137. ¿Cuál se apagó ordenadamente y cuál no?
5. ¿Qué añade set -o pipefail que set -e por sí solo no cubre?

Referencias

- The Open Group (2018). POSIX.1-2017, System Interfaces: fork, execve, wait. <https://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html>
- Kerrisk, M. signal(7) — Linux manual pages: catálogo de señales y semántica de SIGTERM y SIGKILL. <https://man7.org/linux/man-pages/man7/signal.7.html>
- Kerrisk, M. (2010). The Linux Programming Interface, caps. 24-27 — creación de procesos y ejecución de programas.
- systemd (2024). systemd.exec(5) — cómo se construye el entorno de un servicio. <https://www.freedesktop.org/software/systemd/man/systemd.exec.html>
- Ward, B. (2021). How Linux Works, 3.ª ed., caps. 1-2 — procesos, dispositivos y arranque.
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 00 en PDF · Manual integral

Anterior	Índice	Siguiente
← 001 · Computación digital y modelo mental de la nube	Parte 00 · Programa	003 · Git, GitHub y trabajo reproducible →

Evaluación — 002 Terminal, sistema de archivos, procesos y variables de entorno

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 bitacora-de-comandos 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.

003 — Git, GitHub y trabajo reproducible

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 4 Laboratorio: git · Estado: EXECUTABLECORE

Propósito

Entender Git como un grafo dirigido acíclico de instantáneas inmutables direccionadas por contenido, no como una lista de parches. Esa diferencia explica por qué rebase, revert y reset hacen lo que hacen, y es la base de todo lo que viene después: GitOps, entrega continua, trazabilidad de auditoría y reproducibilidad de un despliegue.

Resultados de aprendizaje

Al finalizar podrás:

1. Explicar cómo el hash SHA-1/SHA-256 de un commit se deriva de árbol, padres, autor y mensaje, y qué implica para la integridad del historial.
2. Distinguir reset --soft, --mixed y --hard por lo que hacen sobre HEAD, índice y árbol de trabajo.
3. Elegir entre merge, rebase y revert según si la rama es compartida y si el historial debe ser auditable.
4. Recuperar trabajo aparentemente perdido usando reflog, entendiendo por qué existe la ventana de recuperación.
5. Diseñar un flujo de ramas trazable que permita responder «qué código exacto había en producción el martes a las 14:00».

Conceptos centrales

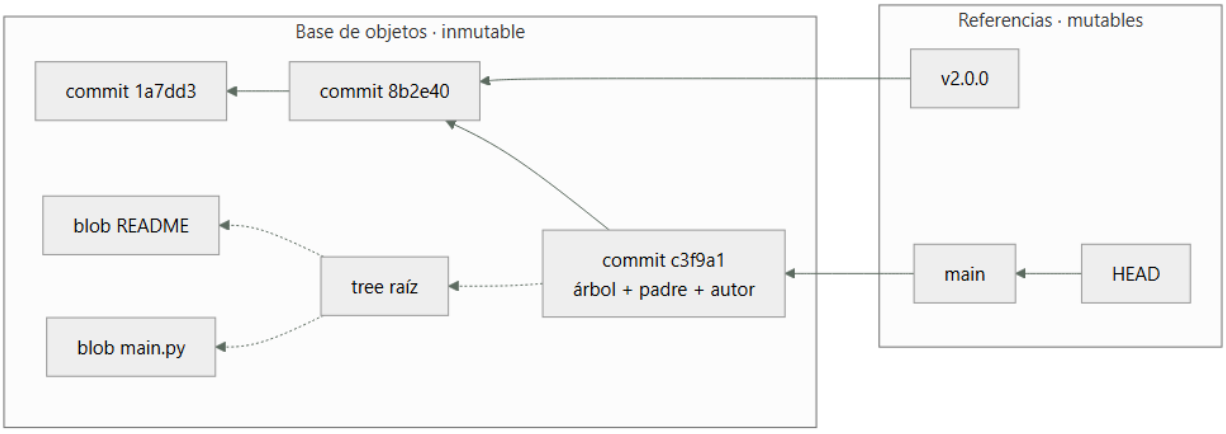
Concepto	Comprensión verificable
objeto	Unidad inmutable de almacenamiento en Git, nombrada por el hash de su contenido. Hay cuatro tipos: blob (contenido de fichero), tree (directorio), commit (instantánea con metadatos) y tag anotado.
commit	Objeto que apunta a un árbol completo, a cero o más padres, y lleva autor, fecha y mensaje. No guarda un diff: guarda el estado entero. Los diffs se calculan al vuelo comparando árboles.
índice	Área intermedia entre el árbol de trabajo y el repositorio, también llamada staging. Es un fichero binario que lista qué versión de cada ruta entrará en el próximo commit.
referencia	Puntero mutable a un commit. Las ramas y HEAD son referencias; por eso crear una rama cuesta 41 bytes y es una operación instantánea sea cual sea el tamaño del repositorio.
reflog	Registro local de todos los valores por los que ha pasado cada referencia. Permite recuperar commits que ya no alcanza ninguna rama, durante 90 días por defecto.

Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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. Git guarda instantáneas, no diferencias

La intuición equivocada más extendida es que un commit almacena «lo que cambió». Almacena el árbol completo del proyecto en ese instante. Los ficheros que no cambiaron no se duplican: el árbol nuevo reutiliza los mismos objetos blob, porque están nombrados por el hash de su contenido.

```
$ git cat-file -p HEAD
tree a4f2c81e9b3d5a7c2f8e1b6d4a9c3e7f2b8d5a1c
parent 8b2e40d7c1a5f9e3b6d2a8c4f7e1b9d3a5c8e2f4
author Vladimir Acuña <...> 1785546717 -0300
committer Vladimir Acuña <...> 1785546717 -0300

Optimiza el README principal
```

El hash del commit se calcula sobre exactamente ese texto. Cambiar una coma del mensaje, la fecha o el padre produce un hash distinto. De ahí se sigue la propiedad que sostiene toda auditoría: no se puede alterar historia pasada sin cambiar los identificadores de todo lo que vino después. Si alguien reescribe un commit de hace un mes, los 200 commits posteriores cambian de hash y cualquier clon lo detecta.

2. Tres zonas y el comando que las mueve

Todo el modelo operativo de Git cabe en tres zonas y una tabla:

```
árbol de trabajo  →  índice (staging)  →  repositorio (objetos)
        git add ■■■■■■■■          git commit ■■■■■■■■
```

git reset es un único comando que actúa sobre las tres según el modificador, y confundirlos es la causa habitual de trabajo perdido:

Comando	HEAD	Índice	Árbol de trabajo	Uso típico
reset --soft <c>	mueve	intacto	intacto	Rehacer el mensaje o agrupar commits
reset --mixed <c>	mueve	reinicia	intacto	Deshacer un add (por defecto)
reset --hard <c>	mueve	reinicia	descarta	Descartar todo; es el único destructivo

Solo --hard toca el árbol de trabajo. Los cambios no confirmados que destruye no están en ningún objeto, así que el reflog no los recupera: nunca llegaron a existir para Git.

3. Merge, rebase y revert: tres respuestas a preguntas distintas

No son alternativas de estilo. Responden a preguntas diferentes:

- merge crea un commit con dos padres. Preserva la historia tal como ocurrió y no cambia hashes existentes. Es la única opción segura sobre una rama que otros ya tienen.
- rebase reescribe: toma tus commits y los vuelve a aplicar sobre otra base, creando commits nuevos con hashes nuevos. Produce historia lineal y legible, pero si la rama es compartida, todo el que la tenga verá divergencia.

- revert crea un commit nuevo que deshace los efectos de otro. No borra nada. Es lo único aceptable para deshacer algo que ya está en producción o en una rama protegida.

La regla operativa, conocida como golden rule of rebasing: no reescribas historia que otros puedan tener. En la práctica: rebase libremente en tu rama local antes del primer push; después, merge o revert.

Para un repositorio con requisitos de auditoría —los de la parte 11— revert es obligatorio: deja constancia de que hubo un error y de cuándo se corrigió. Un reset --hard seguido de push --force borra esa evidencia.

4. El reflog: por qué casi nada se pierde de verdad

Cuando una rama deja de apuntar a un commit, ese commit queda huérfano: sigue en la base de objetos pero no lo alcanza ninguna referencia. El reflog registra cada movimiento de cada referencia, así que aún se puede llegar a él.

```
$ git reset --hard HEAD~3        # "perdí" tres commits
$ git reflog
c3f9a1 HEAD@{0}: reset: moving to HEAD~3
7d2b8e HEAD@{1}: commit: añade validación de contratos
$ git reset --hard HEAD@{1}      # recuperados
```

Los objetos huérfanos sobreviven hasta que git gc los recoge: 90 días para los alcanzables desde el reflog y 30 para el resto, según gc.reflogExpire. Dos matices que importan en operación:

1. El reflog es estrictamente local. No se clona ni se envía. Un push --force que destruye commits en el servidor no deja reflog para quien clone después.
2. Solo registra lo que llegó a ser un commit. Cambios en el árbol de trabajo que nunca se confirmaron no aparecen.

5. Trazabilidad: del commit al artefacto desplegado

La pregunta que toda auditoría acaba haciendo es: ¿qué código exacto estaba corriendo cuando ocurrió el incidente? Responderla exige una cadena sin huecos:

```
commit (sha) → build (id) → artefacto (digest) → despliegue (revisión) → instante
```

Cada eslabón debe ser inmutable y verificable. Por eso en las partes 05 y 08 se insistirá en referenciar imágenes por digest (sha256:...) y no por etiqueta: una etiqueta como :latest o :v2.0 es una referencia mutable —igual que una rama de Git— y puede apuntar a contenido distinto mañana.

El equivalente en Git es el tag anotado, que sí es un objeto con su propio hash, autor y mensaje, frente al tag ligero que es solo un puntero:

```
$ git tag -a v2.0.0 -m "Release 2.0.0"    # objeto, firmable y con metadatos
$ git tag v2.0.0                            # solo una referencia más
```

Para releases, el tag anotado y firmado (-s) es el que sostiene la afirmación «esto es lo que publicamos».

Ejemplo trabajado

Un despliegue del martes rompió el checkout de CloudShop y hay que responder tres preguntas: qué se desplegó, quién lo aprobó y cómo se revierte sin perder la evidencia.

Paso 1 — identificar el commit exacto que estaba en producción:

```
$ git log --format='%h %ad %an %s' --date=iso -3 v2.4.1
7d2b8e 2026-07-28 14:02:11 -0300  Ana Ruiz   Cambia el cálculo de impuestos
1a7dd3 2026-07-28 11:40:02 -0300  Luis Vega  Añade índice a pedidos
8b2e40 2026-07-27 17:15:44 -0300  Ana Ruiz   Actualiza dependencias
```

Paso 2 — confirmar que el artefacto desplegado corresponde a ese commit. La etiqueta no basta; se compara el digest:

```
$ git rev-parse v2.4.1
7d2b8e91c4a2f7d5b3e8a1c6f9d2b4e7a5c8f1d3
$ kubectl get deploy cloudshop -o jsonpath='{...image}'
registry/cloudshop@sha256:4f2a9c...
$ crane config registry/cloudshop@sha256:4f2a9c... | jq -r '.config.Labels."org.opencontainers.image.
revision"'
7d2b8e91c4a2f7d5b3e8a1c6f9d2b4e7a5c8f1d3     # coincide
```

Paso 3 — aislar el cambio culpable con búsqueda binaria sobre el historial:

```
$ git bisect start v2.4.1 v2.4.0
$ git bisect run ./tests/checkout_smoke.sh
7d2b8e91 is the first bad commit
```

Con 3 commits bastan 2 pruebas; con 1.000 bastarían 10, porque bisect es logarítmico: ■log■(1000)■ = 10.

Paso 4 — revertir sin borrar:

```
$ git revert 7d2b8e -m "Revierte el cálculo de impuestos: rompe el checkout"
[main 9e4c2a] Revert "Cambia el cálculo de impuestos"
```

El historial conserva ahora el error y su corrección. Un reset --hard 1a7dd3 && push --force habría dejado producción igual de sana y la auditoría sin nada que leer: no habría constancia de que el fallo existió, ni de cuánto tardó en detectarse.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/003-git-github-y-trabajo-reproducible/
lab.py
```

El laboratorio selecciona el motor de práctica git 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 repositorio-reproducible 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 historial pequeño, legible y verificable. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

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

■ Criterio de aceptación

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

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
Tras un rebase sobre una rama compartida, el equipo ve commits duplicados y conflictos repetidos	Se reescribieron hashes que otros ya tenían	Rebase solo antes del primer push; sobre ramas compartidas usa merge.
Se perdió trabajo con reset --hard y el reflog no lo recupera	Los cambios nunca se confirmaron, así que no existe objeto que recuperar	Confirma o usa git stash antes de cualquier operación destructiva.
Nadie puede decir qué código había en producción durante el incidente	El artefacto se referenció por etiqueta mutable en vez de por digest	Referencia imágenes por sha256: y graba el commit en una etiqueta OCI del artefacto.
El historial de la rama principal no permite auditar una corrección urgente	Se usó reset --hard y push --force en lugar de revert	Deshaz siempre con revert en ramas protegidas; deja el error visible junto a su corrección.
Un git tag no lleva autor ni fecha y no se puede firmar	Se creó un tag ligero, que es solo un puntero	Usa git tag -a (o -s para firmarlo) en cualquier release.

■ 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. Si un commit guarda el árbol completo, ¿por qué un repositorio con 10.000 commits no ocupa 10.000 veces el tamaño del proyecto?
2. ¿Cuál de los tres modos de reset puede destruir trabajo que el reflog no recuperará, y por qué precisamente ese?
3. Un compañero ya hizo pull de tu rama. ¿Qué te impide hacer rebase sobre ella y qué usarías en su lugar?
4. ¿Por qué desplegar imagen:v2.0 no permite responder qué código corría el martes, y qué referencia lo permitiría?
5. Con 1.024 commits entre una versión buena y una mala, ¿cuántas pruebas necesita git bisect en el peor caso?

Referencias

- Chacon, S. y Straub, B. (2014). Pro Git, 2.ª ed., cap. 10 «Git Internals» — objetos, referencias y modelo de almacenamiento. <https://git-scm.com/book/en/v2/Git-Internals-Git-Objects>
- Git (2024). git-reset(1) — tabla oficial del efecto de cada modo sobre HEAD, índice y árbol de trabajo. <https://git-scm.com/docs/git-reset#resetpathspec>
- Git (2024). git-bisect(1) — búsqueda binaria automatizada sobre el historial. <https://git-scm.com/docs/git-bisect>
- Open Container Initiative (2024). Image Specification — anotaciones estándar, incluida org.opencontainers.image.revision. <https://github.com/opencontainers/image-spec/blob/main/annotations.md>
- Git (2024). gitrevisions(7) — sintaxis de HEAD@{n}, y ^ para navegar el grafo. <https://git-scm.com/docs/gitrevisions>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 00 en PDF · Manual integral

Anterior	Índice	Siguiente
← 002 · Terminal, sistema de archivos, procesos y variables de entorno	Parte 00 · Programa	004 · Python, JSON y automatización mínima →

Evaluación — 003 Git, GitHub y trabajo reproducible

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 repositorio-reproducible 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.

004 — Python, JSON y automatización mínima

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 4 Laboratorio: automation · Estado: EXECUTABLECORE

Propósito

Usar Python y JSON como el pegamento mínimo de la operación cloud: leer una respuesta de API, validar su forma antes de confiar en ella y producir salida estructurada que otro programa pueda consumir. Casi todos los lab.py del programa emiten JSON, y casi toda automatización real consiste en encadenar contratos de este tipo.

Resultados de aprendizaje

Al finalizar podrás:

1. Distinguir los tipos de JSON de los de Python y anticipar las conversiones que rompen datos (enteros grandes, None, claves no textuales).
2. Explicar por qué los números en coma flotante de JSON no representan dinero y qué usar en su lugar.
3. Validar una estructura recibida antes de usarla, en vez de confiar en que la API cumple su documentación.
4. Producir salida determinista: mismas entradas y misma semilla producen bytes idénticos.
5. Diferenciar un fallo esperado, que se maneja, de uno inesperado, que debe propagarse con contexto.

Conceptos centrales

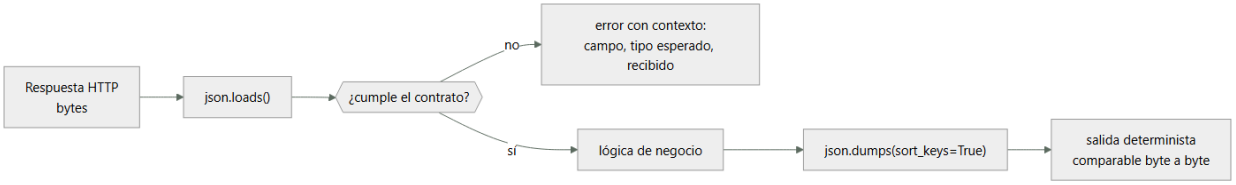
Concepto	Comprensión verificable
serialización	Convertir estructuras en memoria a una secuencia de bytes transportable. JSON solo admite seis tipos: objeto, array, cadena, número, booleano y null; todo lo demás debe traducirse explícitamente.
contrato de datos	Acuerdo explícito sobre qué campos existen, de qué tipo son y cuáles son obligatorios. Sin él, un cambio en el productor rompe al consumidor en producción y no en pruebas.
determinismo	Propiedad por la que las mismas entradas producen exactamente la misma salida. Exige orden estable de claves, semilla fija en cualquier aleatoriedad y ausencia de marcas de tiempo en la salida comparada.
idempotencia	Propiedad por la que repetir una operación no cambia el resultado respecto de ejecutarla una vez. Es lo que hace seguro reintentar tras un error de red.
excepción	Mecanismo para señalar que una operación no puede completarse. Capturar Exception de forma genérica convierte un fallo diagnosticable en uno silencioso.

Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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. JSON y Python no comparten sistema de tipos

El mapeo parece directo hasta que deja de serlo:

JSON	Python al leer	Trampa
object	dict	Las claves siempre acaban siendo str
array	list	Las tuplas se serializan como array y vuelven como lista
number	int o float	No hay distinción en JSON: la decide el punto decimal
true/false	bool	—
null	None	—

La conversión no es reversible:

```
>>> import json
>>> json.loads(json.dumps({1: "a", (2, 3): "b"}))
{'1': 'a', '2, 3': 'b'}          # las claves se volvieron cadenas
>>> json.loads(json.dumps((1, 2)))
[1, 2]                            # la tupla ya no es tupla
```

En una automatización que lee un estado, lo modifica y lo vuelve a escribir, esto significa que el ciclo no es una identidad. Si el consumidor compara claves numéricas, el segundo ciclo falla y el primero no.

2. Los números de JSON no sirven para dinero

JSON no define precisión: la mayoría de implementaciones usan IEEE 754 de doble precisión, que es binario. Los decimales que no son suma de potencias de dos no tienen representación exacta:

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
```

Aplicado a una factura cloud con 43.200 líneas de consumo horario, el error se acumula. La corrección es no usar coma flotante para valores exactos:

```
>>> from decimal import Decimal
>>> Decimal("0.1") + Decimal("0.2") == Decimal("0.3")
True
```

Hay un segundo límite, y es de interoperabilidad: JavaScript representa todos los números como double, así que solo garantiza enteros exactos hasta 2■³–1 = 9.007.199.254.740.991. Un identificador de 64 bits enviado como número JSON pierde precisión al pasar por cualquier consumidor JavaScript. Por eso las APIs serias envían los identificadores grandes como cadena; en la parte 09 se verá la misma decisión en los sistemas de mensajería.

3. Validar antes de confiar

El error de automatización más caro es asumir que la respuesta tiene la forma documentada. Una API puede devolver 200 con un cuerpo distinto del esperado, y el fallo aparece tres capas más abajo, sin contexto:

```
# frágil: revienta en otro sitio, con un KeyError sin pistas
costo = respuesta["data"]["billing"]["total"]

# explícito: falla donde está el problema y dice cuál es
def leer_costo(respuesta: dict) -> Decimal:
    for clave in ("data", "billing"):
        if clave not in respuesta:
            raise ValueError(f"falta '{clave}' en la respuesta; claves: {sorted(respuesta)}")
        respuesta = respuesta[clave]
    bruto = respuesta.get("total")
    if not isinstance(bruto, str):
        raise TypeError(f"'total' debía ser cadena decimal, llegó {type(bruto).__name__}")
    return Decimal(bruto)
```

La diferencia práctica: el primero produce KeyError: 'billing' a las 3 de la madrugada; el segundo dice qué campo falta y qué llegó en su lugar. En un runbook —parte 21— esa distinción es la diferencia entre cinco minutos y dos horas.

4. Determinismo: la propiedad que hace verificable un laboratorio

Los 288 laboratorios de este programa emiten un contrato JSON que debe ser byte a byte idéntico con la misma semilla. Tres condiciones lo garantizan:

```
import json, random

random.seed(42)                       # 1. aleatoriedad reproducible
resultado = {"decision": "...", "evidencia": [...]}
print(json.dumps(resultado,
                 sort_keys=True,      # 2. orden estable de claves
                 ensure_ascii=False)) # 3. codificación estable
```

Sin sortkeys, Python conserva el orden de inserción y dos ejecuciones que construyen el diccionario por caminos distintos producen bytes distintos con el mismo contenido. Sin semilla, no hay repetición posible.

Lo que nunca debe entrar en una salida comparada es la hora actual: convierte cualquier comparación en un falso negativo. Si hace falta registrar el instante, va en un fichero aparte o en un campo excluido de la comparación.

5. Fallos esperados frente a fallos inesperados

Un error de red al llamar a una API es esperado: ocurre, y el código debe reintentar. Un TypeError porque una función recibió una lista donde esperaba un diccionario es inesperado: es un defecto, y ocultarlo lo hace indetectable.

```
# destruye el diagnóstico: cualquier fallo se vuelve None
try:
    datos = pedir(url)
except Exception:
    datos = None

# maneja lo esperado y deja propagar lo demás
for intento in range(5):
    try:
        datos = pedir(url)
        break
    except (TimeoutError, ConnectionError) as e:
        espera = 2 ** intento + random.uniform(0, 1)   # 1, 2, 4, 8, 16 s + jitter
        log.warning("intento %d falló (%s); reintento en %.1f s", intento + 1, e, espera)
        time.sleep(espera)
else:
    raise RuntimeError(f"{url} no respondió tras 5 intentos")
```

El retroceso exponencial con jitter no es un detalle: sin el término aleatorio, mil clientes que fallan a la vez reintentan a la vez y mantienen caído el servicio que intentan usar. Es el thundering herd, y reaparece en las partes 10 y 21.

Ejemplo trabajado

Un script de FinOps suma el consumo horario de CloudShop y su total no cuadra con la factura del proveedor. La diferencia es de 0,37 USD sobre 8.412,55 — pequeña, pero impide cerrar el mes.

El script original:

```
total = 0.0
for linea in json.loads(respuesta)["lineItems"]:
    total += linea["cost"]          # cost llega como número JSON
print(f"{total:.2f}")
```

Reproducción del error con las 720 líneas de un mes:

```
>>> valores = [0.0117] * 720           # coste horario de una instancia pequeña
>>> sum(valores)
8.423999999999999
>>> float(Decimal("0.0117") * 720)
8.424
```

Sobre una línea el error es de 10■[1]■; sobre 43.200 líneas de consumo mensual —60 recursos × 720 horas— el error acumulado alcanza el orden de los céntimos. No es un fallo del proveedor: es que la suma se hizo en binario.

Corrección, sumando en decimal y validando el tipo de entrada:

```
from decimal import Decimal, ROUND_HALF_UP

total = Decimal("0")
```

```
for n, linea in enumerate(json.loads(respuesta)["lineItems"]):
    bruto = linea["cost"]
    if isinstance(bruto, float):
        raise TypeError(f"línea {n}: 'cost' llegó como float; exige cadena decimal")
    total += Decimal(str(bruto))
print(total.quantize(Decimal("0.01"), rounding=ROUND_HALF_UP))
```

```
8412.55        # cuadra con la factura
```

La comprobación de tipo es deliberada: convierte un error silencioso de céntimos en un fallo ruidoso en la primera línea. Si el proveedor cambia el formato y empieza a enviar cost como número, el script se para en vez de producir un total plausible pero incorrecto.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/004-python-json-y-automatizacion-minima/
lab.py
```

El laboratorio selecciona el motor de práctica automation 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 script-de-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 un script idempotente con salida estructurada. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye script-de-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
Un total monetario difiere en céntimos de la factura oficial	Se sumó en coma flotante IEEE 754, que no representa decimales exactos	Usa Decimal construido desde cadena y redondea solo al final.
Un identificador de 64 bits llega alterado al frontend	JavaScript solo garantiza enteros exactos hasta 2■³–1	Transporta los identificadores grandes como cadena JSON.
Dos ejecuciones con la misma semilla producen ficheros distintos	El orden de claves depende del orden de inserción, o hay una marca de tiempo en la salida	Serializa con sortkeys=True y saca la hora de la salida comparada.
El script falla con KeyError en un punto que no tiene relación con la causa	Se accedió a campos anidados sin validar la forma recibida	Valida presencia y tipo en el borde, y falla con un mensaje que diga qué faltaba.
Tras un incidente, mil clientes reintentan a la vez y el servicio no levanta	Retroceso sin jitter: todos reintentan sincronizados	Añade un término aleatorio al retroceso exponencial.

■ 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 json.loads(json.dumps(d)) puede no devolver d? Da dos casos concretos.
2. Un endpoint devuelve "id": 9007199254740993. ¿Qué le ocurre a ese valor en un consumidor JavaScript y cómo se evita?
3. ¿Qué tres condiciones debe cumplir un laboratorio para que su salida sea comparable byte a byte?
4. ¿Cuándo es correcto capturar una excepción y cuándo hacerlo oculta un defecto?
5. Sin jitter, ¿qué le ocurre a un servicio que se recupera si mil clientes usan el mismo retroceso exponencial?

Referencias

- Bray, T., ed. (2017). RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format — sección 6 sobre los límites de precisión numérica. <https://www.rfc-editor.org/rfc/rfc8259#section-6>
- IEEE (2019). Standard for Floating-Point Arithmetic (IEEE 754-2019). <https://doi.org/10.1109/IEEESTD.2019.8766229>
- Goldberg, D. (1991). What Every Computer Scientist Should Know About Floating-Point Arithmetic. ACM Computing Surveys 23(1). <https://doi.org/10.1145/103162.103163>
- Python Software Foundation (2024). decimal — Decimal fixed-point and floating-point arithmetic. <https://docs.python.org/3/library/decimal.html>
- Brooker, M. (2015). Exponential Backoff and Jitter. AWS Architecture Blog — datos comparados de las variantes de jitter. <https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 00 en PDF · Manual integral

Anterior	Índice	Siguiente
← 003 · Git, GitHub y trabajo reproducible	Parte 00 · Programa	005 · Redes por capas, TCP/IP, puertos y sockets →

Evaluación — 004 Python, JSON y automatización mínima

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 script-de-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.

005 — Redes por capas, TCP/IP, puertos y sockets

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 4 Laboratorio: network · Estado: EXECUTABLECORE

Propósito

Entender por qué una red se organiza en capas y qué garantiza cada una, para poder diagnosticar sin adivinar. Cuando en la parte 16 aparezcan VPC, tablas de rutas, NAT y balanceadores, todo será este modelo aplicado; y cuando un servicio «no responda», sabrás en qué capa mirar antes de tocar nada.

Resultados de aprendizaje

Al finalizar podrás:

1. Situar un síntoma en la capa correcta y elegir la herramienta de diagnóstico que corresponde a esa capa.
2. Explicar qué garantiza TCP que UDP no, y qué cuesta esa garantía en tiempo de establecimiento.
3. Identificar una conexión por su cuádrupla y deducir cuántas conexiones simultáneas admite un cliente contra un mismo destino.
4. Interpretar los estados de un socket —LISTEN, SYNSENT, TIMEWAIT— para distinguir un puerto cerrado de un filtrado.
5. Calcular el throughput máximo de una conexión TCP a partir de su ventana y su latencia.

Conceptos centrales

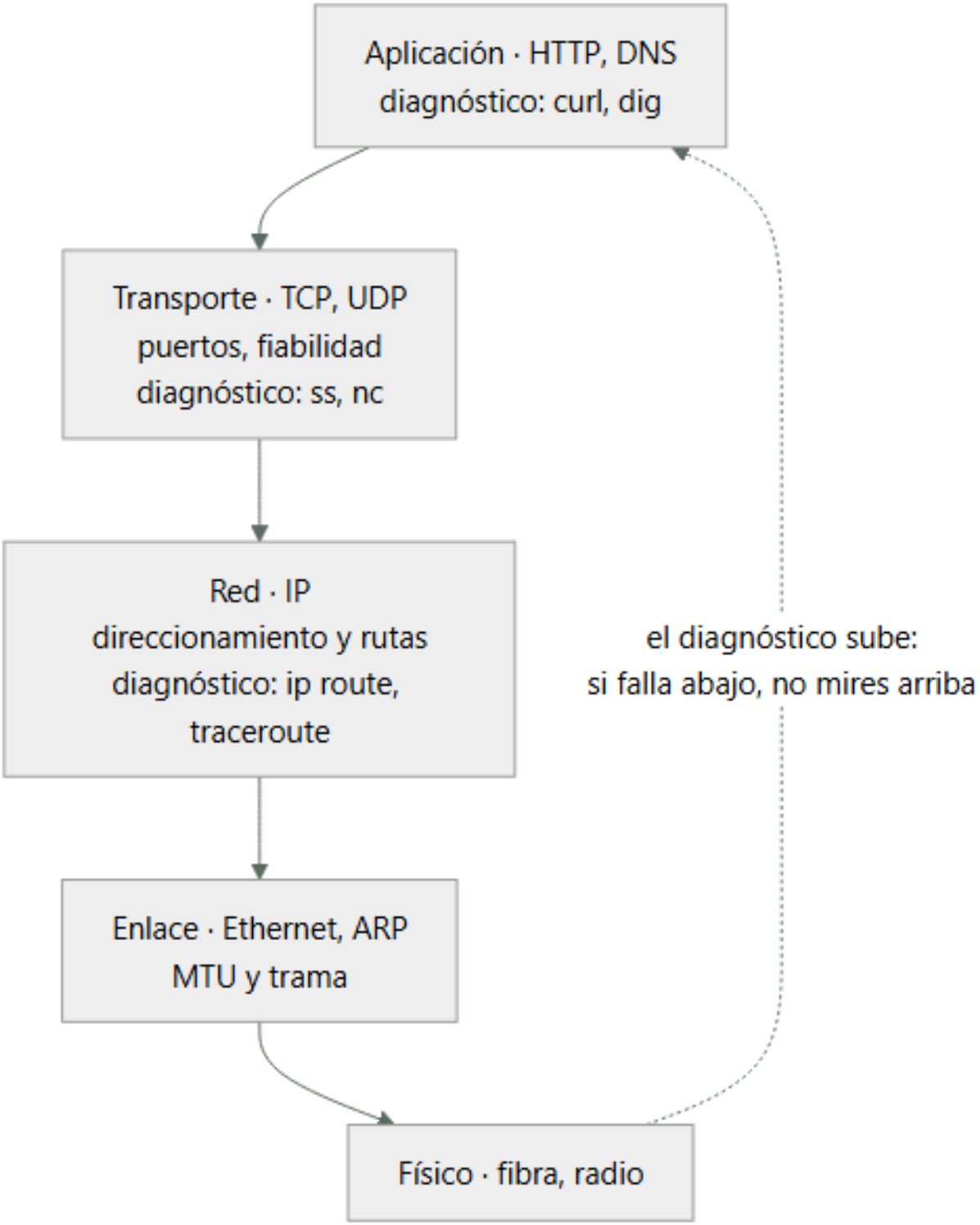
Concepto	Comprensión verificable
encapsulación	Cada capa envuelve los datos de la superior con su propia cabecera y no interpreta el contenido. Es lo que permite cambiar de medio físico sin tocar la aplicación.
socket	Extremo de una comunicación, identificado por protocolo, IP y puerto. El sistema operativo lo expone como descriptor de fichero, de ahí que read y write funcionen sobre él igual que sobre un archivo.
cuádrupla	IP origen, puerto origen, IP destino y puerto destino. Identifica unívocamente una conexión TCP; dos conexiones pueden compartir tres de los cuatro valores pero no los cuatro.
MTU	Tamaño máximo de trama que un enlace transporta sin fragmentar. 1500 bytes en Ethernet estándar; los túneles restan espacio para su propia cabecera y son la causa habitual de conexiones que se establecen pero se cuelgan al transferir.
ventana de recepción	Bytes que el receptor declara poder aceptar sin confirmar. Junto con la latencia determina el techo de velocidad de una conexión TCP, independientemente del ancho de banda contratado.

Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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. Cada capa garantiza una cosa y solo una

El valor del modelo por capas no es taxonómico: es que acota dónde buscar. Cada capa promete algo concreto y no se hace cargo de lo demás:

Capa	Garantiza	No garantiza	Herramienta
Aplicación	Semántica del mensaje	Que llegue	curl -v, dig
Transporte (TCP)	Orden, integridad, entrega	Tiempo de llegada	ss -tnp, nc -vz

Capa	Garantiza	No garantiza	Herramienta
Transporte (UDP)	Nada más que multiplexar por puerto	Orden ni entrega	nc -u
Red (IP)	Mejor esfuerzo hacia un destino	Orden, entrega ni ruta estable	ip route, traceroute
Enlace	Entrega dentro de un mismo segmento	Nada fuera de él	ip neigh, ping con MTU

La regla de diagnóstico es de abajo hacia arriba: si la capa 3 no alcanza el destino, leer logs de la aplicación es tiempo perdido. En la parte 21, los runbooks de triaje seguirán exactamente este orden.

2. TCP paga con tiempo lo que ofrece en garantías

TCP establece la conexión antes de enviar un solo byte útil, con el saludo de tres vías:

```
cliente ■■SYN■■■■■■■■■■■■ servidor
cliente ■■SYN+ACK■■■■■■■■ servidor
cliente ■■ACK■■■■■■■■■■■■ servidor      ya se pueden enviar datos
```

Eso es un RTT completo antes del primer byte. Con Madrid–Virginia a 90 ms de ida y vuelta:

```
TCP                          90 ms
TLS 1.2 (2 RTT)             180 ms
--------------------------------
total antes del primer byte 270 ms

TCP                          90 ms
TLS 1.3 (1 RTT)              90 ms
--------------------------------
total                       180 ms
```

Por eso TLS 1.3 no es una mejora cosmética: elimina un RTT completo de cada conexión nueva. Y por eso la reutilización de conexiones (keep-alive, pooling) importa tanto: amortiza ese coste entre muchas peticiones. Un cliente que abre y cierra conexión por petición paga 180 ms de peaje cada vez.

UDP no establece nada: el primer datagrama ya lleva datos. A cambio, la aplicación se hace cargo del orden, la pérdida y la congestión. Es la elección correcta para DNS, telemetría y voz, donde llegar tarde es peor que no llegar.

3. Puertos, cuádruplas y el límite real de conexiones

El puerto no identifica una conexión: identifica un extremo. La conexión es la cuádrupla completa, y por eso mil clientes pueden hablar con el puerto 443 del mismo servidor sin ambigüedad: difieren en IP y puerto de origen.

El límite aparece al revés, cuando un cliente abre muchas conexiones al mismo destino. Solo puede variar su puerto de origen, y el rango efímero de Linux es:

```
$ cat /proc/sys/net/ipv4/ip_local_port_range
32768■60999                  # 28.232 puertos disponibles
```

Un proxy o un pod que llama a un mismo backend tiene un techo de 28.232 conexiones simultáneas, y en la práctica bastante menos porque cada cierre deja el puerto en TIMEWAIT durante 2×MSL (60 s en Linux):

```
conexiones/s sostenibles ≈ 28.232 / 60 s ≈ 470 por segundo
```

Superado ese ritmo aparece EADDRNOTAVAIL y el síntoma es «la aplicación falla bajo carga sin error de la aplicación». La solución no es más CPU: es reutilizar conexiones. Este cálculo reaparecerá en las partes 06 y 16 al dimensionar NAT gateways, que tienen exactamente el mismo límite por IP.

4. El estado del socket dice qué falla

ss muestra en qué punto del protocolo se atascó la conexión, y cada estado apunta a una causa distinta:

```
$ ss -tnp state all '( dport = :443 )'
State       Recv-Q Send-Q  Local Address:Port   Peer Address:Port
ESTAB       0      0       10.0.1.5:51234       93.184.216.34:443
SYN-SENT    0      1       10.0.1.5:51240       10.0.9.9:443
TIME-WAIT   0      0       10.0.1.5:51201       93.184.216.34:443
```

Síntoma	Estado observado	Causa probable
Rechazo inmediato	Nada, ECONNREFUSED	Puerto cerrado: nadie escucha, pero la IP responde

Síntoma	Estado observado	Causa probable
Cuelgue hasta timeout	SYN-SENT persistente	Filtrado: un firewall descarta en silencio
Conecta y se cuelga al transferir	ESTAB con Send-Q creciendo	MTU: el saludo cabe, los datos no
Falla solo con carga	Muchos TIME-WAIT	Agotamiento de puertos efímeros

La distinción entre rechazado y filtrado es la más útil de todas: rechazado significa que llegaste al host y el servicio no está; filtrado significa que no llegaste. Son dos equipos distintos los que arreglan cada cosa.

5. El producto ancho de banda × retardo pone el techo

Una conexión TCP no puede ir más rápido que lo que permite su ventana dividida por el tiempo de ida y vuelta. El emisor manda una ventana y espera confirmación antes de continuar:

```
throughput_máximo = ventana / RTT
```

Con la ventana clásica de 64 KB y un enlace transatlántico de 90 ms:

```
65.536 bytes / 0,090 s = 728.178 B/s ≈ 5,8 Mbit/s
```

Da igual que el enlace sea de 10 Gbit/s: una sola conexión no pasará de 5,8 Mbit/s. Para llenar 1 Gbit/s con ese RTT hace falta:

```
ventana = 1.000.000.000 bit/s × 0,090 s / 8 = 11,25 MB
```

De ahí el window scaling de RFC 7323, que permite ventanas de hasta 1 GB, y de ahí que las transferencias intercontinentales usen varias conexiones en paralelo. Cuando alguien diga «contratamos más ancho de banda y no mejoró», la respuesta suele estar en esta fórmula.

Ejemplo trabajado

El servicio de pagos de CloudShop falla de forma intermitente contra el proveedor externo. Los logs de la aplicación solo dicen «timeout». El operador diagnostica por capas, de abajo hacia arriba.

Capa 3 — ¿se alcanza el destino?

```
$ ip route get 203.0.113.40
203.0.113.40 via 10.0.0.1 dev eth0 src 10.0.1.5
$ traceroute -n 203.0.113.40 | tail -2
 7  198.51.100.9  12.4 ms
 8  203.0.113.40  14.1 ms          # alcanzable
```

Capa 4 — ¿se establece la conexión?

```
$ nc -vz 203.0.113.40 443
Connection to 203.0.113.40 443 port [tcp/https] succeeded!
```

Conecta. Así que no es ruta ni firewall. El fallo es intermitente y bajo carga, así que se mira el estado de los sockets durante el pico:

```
$ ss -tan state time-wait | wc -l
27891
$ ss -tan state all '( dport = :443 )' | grep -c ESTAB
412
$ dmesg | tail -1
TCP: request_sock_TCP: Possible SYN flooding...
$ cat /proc/sys/net/ipv4/ip_local_port_range
32768■60999
```

27.891 puertos en TIMEWAIT sobre 28.232 disponibles. El servicio agota el rango efímero:

```
disponibles          = 60.999 − 32.768 = 28.232
en TIME_WAIT         = 27.891  (98,8 %)
libres               =    341
ritmo sostenible     = 28.232 / 60 s ≈ 470 conexiones/s
ritmo observado en pico             ≈ 610 conexiones/s
```

El servicio abre una conexión TCP nueva por cada pago y la cierra. A 610 pagos por segundo pide 140 conexiones/s más de las que el sistema puede reciclar, y los connect() empiezan a devolver EADDRNOTAVAIL, que el cliente HTTP reporta como «timeout».

La corrección es reutilizar conexiones, no tocar el sistema operativo:

```
sesion = requests.Session()          # pool persistente con keep-alive
adapter = HTTPAdapter(pool_maxsize=64)
sesion.mount("https://", adapter)
```

```
$ ss -tan state time-wait | wc -l
743                                  # de 27.891 a 743
```

64 conexiones reutilizadas sustituyen a 610 por segundo. Subir net.ipv4.tcptwreuse habría escondido el síntoma un tiempo; el problema era de diseño del cliente, y estaba a dos capas de donde los logs decían.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/005-redes-por-capas-tcp-ip-puertos-y-sockets/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 captura-y-mapa-de-red 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 captura-y-mapa-de-red 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 falla solo bajo carga y sin error propio de la aplicación	Agotamiento de puertos efímeros por abrir una conexión nueva por petición	Usa un pool con keep-alive; el techo es 28.000 puertos y 60 s de TIMEWAIT por cierre.
La conexión se establece pero se cuelga al transferir datos	MTU insuficiente en un túnel: el saludo cabe en paquetes pequeños y los datos no	Prueba con ping -M do -s para hallar la MTU real y ajusta MSS clamping.
Se contrató más ancho de banda y la transferencia no mejoró	El techo lo pone ventana/RTT, no el caudal del enlace	Habilita window scaling o paraleliza conexiones; calcula el producto ancho de banda × retardo.
Se depuran logs de la aplicación durante una hora y el problema era de red	Se diagnosticó de arriba hacia abajo	Verifica ruta y establecimiento de conexión antes de leer la aplicación.
No se distingue si el puerto está cerrado o filtrado	Se interpretó cualquier fallo de conexión como lo mismo	Rechazo inmediato es puerto cerrado; cuelgue en SYN-SENT es filtrado. Son equipos distintos.

■ 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. ¿Cuántos RTT median entre el primer paquete y el primer byte útil con TCP + TLS 1.3, y cuántos con TLS 1.2?
2. Un pod abre conexiones a un solo backend. ¿Cuál es su techo teórico de conexiones simultáneas y de dónde sale ese número?
3. Con ventana de 64 KB y RTT de 200 ms, ¿cuál es el throughput máximo de una conexión, y cambia si contratas 10 Gbit/s?
4. ¿Qué diferencia operativa hay entre un ECONNREFUSED inmediato y un cuelgue en SYN-SENT?
5. Una conexión se establece y se cuelga al enviar un cuerpo grande. ¿Qué capa sospechas y con qué comando lo confirmas?

Referencias

- Kurose, J. y Ross, K. (2021). Computer Networking: A Top-Down Approach, 8.ª ed., caps. 3-4 — transporte y capa de red.
- Postel, J., ed. (1981). RFC 793: Transmission Control Protocol — saludo de tres vías y máquina de estados. <https://www.rfc-editor.org/rfc/rfc793>
- Borman, D. et al. (2014). RFC 7323: TCP Extensions for High Performance — window scaling y el producto ancho de banda × retardo. <https://www.rfc-editor.org/rfc/rfc7323>
- Rescorla, E. (2018). RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3 — saludo de 1-RTT. <https://www.rfc-editor.org/rfc/rfc8446>
- Kerrisk, M. ss(8) y tcp(7) — estados de socket y parámetros del núcleo de Linux. <https://man7.org/linux/man-pages/man7/tcp.7.html>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 00 en PDF · Manual integral

Evaluación — 005 Redes por capas, TCP/IP, puertos y sockets

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 captura-y-mapa-de-red 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.

006 — DNS, HTTP, HTTPS y TLS de extremo a extremo

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 4 Laboratorio: network · Estado: EXECUTABLECORE

Propósito

Seguir una petición HTTPS desde el nombre hasta el byte: quién resuelve el nombre y con qué caché, qué negocia TLS y qué demuestra un certificado, y qué significan realmente los códigos y cabeceras de HTTP. Es el camino que recorren todas las peticiones del programa, y donde se originan la mayoría de incidentes de disponibilidad percibida.

Resultados de aprendizaje

Al finalizar podrás:

1. Trazar una resolución DNS completa e identificar en qué caché se queda un cambio que «no se propaga».
2. Explicar qué demuestra un certificado TLS y qué no, y por qué el cifrado no implica confianza en el otro extremo.
3. Elegir un TTL de DNS en función de la ventana de conmutación que necesita tu plan de continuidad.
4. Distinguir un 502 de un 503 y de un 504 para saber a qué componente apunta cada uno.
5. Diseñar una política de caché HTTP que permita invalidar sin esperar a que expire.

Conceptos centrales

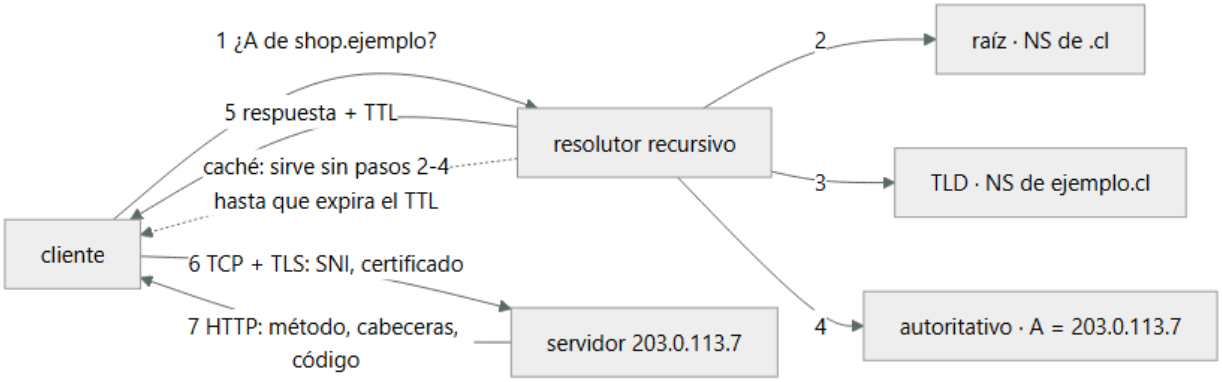
Concepto	Comprensión verificable
resolutor recursivo	Servidor que hace el trabajo de consultar la cadena de autoridades en nombre del cliente y guarda el resultado en caché. Es quien realmente decide durante cuánto tiempo verás una respuesta antigua.
TTL	Segundos que un registro DNS puede permanecer en caché. Fija el suelo del tiempo de conmutación: con TTL de 3600, un cambio de IP tarda hasta una hora en verse, sin importar lo rápido que lo publiques.
SNI	Extensión de TLS que envía el nombre del host solicitado en claro dentro del saludo, para que un servidor con muchos certificados sepa cuál presentar. Al ir en claro, revela qué sitio visitas aunque el contenido vaya cifrado.
cadena de confianza	Secuencia de firmas desde el certificado del servidor hasta una raíz que el cliente ya considera fiable. Un certificado válido demuestra control del nombre, no honestidad del operador.
caché condicional	Mecanismo por el que el cliente pregunta «¿ha cambiado desde esta versión?» con If-None-Match. Si no cambió, el servidor responde 304 sin cuerpo, ahorrando la transferencia pero no el viaje.

Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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. DNS: la caché que decide cuánto dura tu error

Una resolución completa recorre la jerarquía, pero casi ninguna lo hace: el resolutor responde desde caché. Esa caché es la que gobierna la realidad operativa.

```
$ dig +trace shop.ejemplo.cl A | tail -4
ejemplo.cl.     172800  IN  NS  ns1.ejemplo.cl.
shop.ejemplo.cl. 300    IN  A   203.0.113.7      # TTL de 300 s

$ dig shop.ejemplo.cl A +noall +answer
shop.ejemplo.cl. 187    IN  A   203.0.113.7       # quedan 187 s en caché
```

El TTL es una decisión de arquitectura de continuidad, no un parámetro técnico:

TTL	Coste en consultas	Ventana de conmutación	Cuándo
60 s	Alto	≤ 1 min	Antes de una migración planificada
300 s	Medio	≤ 5 min	Servicios con failover DNS
3600 s	Bajo	≤ 1 h	Registros estables
86400 s	Mínimo	≤ 24 h	NS, MX, registros de verificación

La regla operativa: baja el TTL antes de necesitarlo. Bajarlo a 60 s en mitad de un incidente no sirve, porque los resolutores siguen sirviendo la respuesta anterior con su TTL original. Hay que bajarlo con al menos un TTL antiguo de antelación.

Y hay un límite que no controlas: muchos resolutores públicos y sistemas operativos imponen mínimos propios e ignoran TTL muy bajos. DNS es control de tráfico con granularidad de minutos, nunca de segundos; por eso en la parte 13 el failover regional no se apoya solo en DNS.

2. Qué demuestra un certificado y qué no

TLS resuelve tres problemas distintos y conviene no confundirlos:

1. Confidencialidad: nadie en el camino lee el contenido.
2. Integridad: nadie lo modifica sin que se detecte.
3. Autenticación del servidor: hablas con quien dice el nombre.

El certificado solo aporta el tercero, y de forma acotada: prueba que alguien demostró control sobre ese nombre de dominio ante una autoridad en la que tu cliente confía. No dice nada sobre la honestidad del operador ni la seguridad de su código. Un sitio de phishing con certificado válido es exactamente eso: cifrado, íntegro y fraudulento.

```
$ openssl s_client -connect shop.ejemplo.cl:443 -servername shop.ejemplo.cl </dev/null 2>/dev/null \
  | openssl x509 -noout -subject -issuer -dates
subject=CN=shop.ejemplo.cl
issuer=C=US, O=Let's Encrypt, CN=R11
notBefore=Jul  2 00:00:00 2026 GMT
notAfter=Sep 30 23:59:59 2026 GMT
```

El -servername es SNI y es obligatorio: sin él, un servidor con varios certificados presenta el que tenga por defecto y el diagnóstico induce a error. Como SNI viaja en claro, un observador de red sabe qué sitio visitas aunque no vea el contenido; ECH (Encrypted Client Hello) es la respuesta en curso a esa fuga.

3. Los códigos HTTP dicen quién falló

Los códigos 5xx no son intercambiables: cada uno apunta a un componente distinto de la cadena, y confundirlos alarga los incidentes.

Código	Significado	Quién falló
500	Error interno	La aplicación de origen: excepción no controlada
502	Puerta de enlace incorrecta	El proxy habló con el origen y recibió basura o nada
503	Servicio no disponible	El origen se declara saturado o en mantenimiento
504	Tiempo agotado en la puerta	El proxy esperó y el origen no respondió a tiempo

La distinción práctica entre 502 y 504: en el 502 el origen respondió algo inválido o cerró la conexión, así que hay que mirar sus logs; en el 504 no respondió nada dentro del plazo, así que hay que mirar su latencia y saturación. Buscar excepciones en el origen ante un 504 no lleva a ningún sitio, porque probablemente la petición sigue procesándose.

El 503 tiene un matiz que casi nadie usa: admite la cabecera Retry-After, que le dice al cliente cuándo volver. Sin ella, los clientes reintentan de inmediato y agravan la saturación que causó el 503 — el mismo thundering herd de la clase 004.

4. Caché HTTP: el viaje que no se hace

La petición más rápida es la que no ocurre. HTTP ofrece dos niveles, y elegir mal el primero condena a soportar contenido obsoleto:

```
Cache-Control: public, max-age=31536000, immutable
```

Para recursos con huella en el nombre (app.4f2a9c.js): un año, sin revalidar. Como el nombre cambia con el contenido, invalidar es publicar un nombre nuevo; nunca hay que purgar nada.

```
Cache-Control: no-cache
ETag: "7d2b8e91"
```

Para documentos que cambian (index.html): no-cache no significa «no almacenar», significa «almacena pero revalida antes de usar». El cliente pregunta con If-None-Match: "7d2b8e91" y recibe 304 Not Modified sin cuerpo si no cambió: se ahorra la transferencia, no el viaje.

Para prohibir el almacenamiento de verdad hace falta no-store, y solo tiene sentido en respuestas con datos personales o tokens.

El error clásico es servir index.html con max-age largo: entonces una corrección urgente no llega hasta que expire, y no hay forma de forzarlo desde el servidor. Por eso el patrón que se repetirá en las partes 16 y 17 es siempre el mismo: HTML corto y revalidable, activos largos e inmutables con huella en el nombre.

5. El coste completo de una petición en frío

Sumando lo anterior, la primera petición a un origen lejano paga varios peajes secuenciales:

```
resolución DNS (caché fría)        1 RTT al resolutor + cadena
establecimiento TCP                1 RTT
saludo TLS 1.3                     1 RTT
petición y respuesta HTTP          1 RTT
-----------------------------------------------
mínimo                             ≈ 4 RTT
```

Con un RTT de 90 ms son 360 ms antes de mostrar nada, y ninguna optimización del servidor los reduce. Las palancas son todas de red:

- Reutilizar la conexión elimina TCP y TLS de las peticiones siguientes.
- Acercar la terminación (CDN, edge) reduce el RTT que multiplica a todo lo demás.

- 0-RTT de TLS 1.3 permite enviar datos en el primer paquete al reconectar, con la contrapartida de que esos datos son vulnerables a repetición: solo vale para peticiones idempotentes.

Es el mismo cálculo de la clase 001, ahora con los protocolos concretos que lo producen.

Ejemplo trabajado

CloudShop migra su frontend a una IP nueva. El equipo publica el cambio a las 10:00 y a las 11:30 sigue habiendo usuarios llegando al servidor viejo. Se acusa al proveedor de DNS.

Primero se comprueba qué publica la autoridad, sin pasar por caché:

```
$ dig @ns1.ejemplo.cl shop.ejemplo.cl A +noall +answer
shop.ejemplo.cl. 3600 IN A 203.0.113.44        # correcto, ya es la IP nueva
```

La autoridad está bien. Ahora qué ven los resolutores públicos:

```
$ dig @8.8.8.8 shop.ejemplo.cl A +noall +answer
shop.ejemplo.cl. 1187 IN A 203.0.113.7         # IP VIEJA, 1187 s restantes
$ dig @1.1.1.1 shop.ejemplo.cl A +noall +answer
shop.ejemplo.cl.  942 IN A 203.0.113.7
```

El TTL era 3600 s. Cuando se publicó el cambio, los resolutores ya tenían la respuesta anterior cacheada, cada uno con su reloj propio:

```
publicación del cambio          10:00
peor caso: un resolutor cacheó a 09:59  →  expira a 10:59
ventana teórica de convergencia         →  hasta 11:00
observado a 11:30                       →  hay resolutores con TTL de 3600 s
                                           que refrescaron a 10:30
```

La aritmética explica el residual: cada resolutor que refresca justo antes del cambio arrastra una hora más. El tiempo de convergencia no es el TTL: es hasta dos veces el TTL si el cambio coincide con el peor momento de refresco.

Lo que debió hacerse, con un día de antelación:

```
D-1 09:00  bajar TTL de 3600 a 60      (empieza a propagarse)
D-1 10:00  todos los resolutores tienen ya TTL de 60
D   10:00  cambiar la IP                (converge en ≤ 2 min)
D   12:00  restaurar TTL a 3600
```

En el incidente real, la mitigación no fue DNS sino red: se mantuvo el servidor viejo respondiendo con un 308 Permanent Redirect hacia el nuevo hasta que la caché expiró.

La lección: DNS no es un conmutador. Es una caché distribuida cuyo tiempo de reacción se decide con un TTL, y ese TTL hay que bajarlo antes de necesitarlo. Por eso los planes de continuidad de la parte 13 no dependen de DNS para conmutaciones de segundos.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/006-dns-http-https-y-tls-de-extremo-a-extremo/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 traza-de-solicitud 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 traza-de-solicitud para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

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

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
Se cambia una IP y horas después sigue llegando tráfico al servidor antiguo	El TTL alto estaba cacheado en los resolutores antes del cambio	Baja el TTL con al menos un TTL de antelación; cuenta hasta dos veces el TTL como ventana de convergencia.
openssl sclient muestra un certificado que no corresponde al sitio	Se omitió -servername, así que el servidor presentó su certificado por defecto	Incluye siempre SNI con -servername al diagnosticar hosts virtuales.
Se buscan excepciones en el origen ante un 504 y no aparece ninguna	El 504 significa que el origen no respondió a tiempo, no que fallara	Ante 504 mira latencia y saturación del origen; ante 502, sus logs de error.
Una corrección urgente del HTML no llega a los usuarios	Se sirvió el documento con max-age largo y no hay forma de invalidar desde el servidor	HTML con no-cache y ETag; activos inmutables con huella en el nombre.
Un 503 provoca una avalancha de reintentos que empeora la saturación	La respuesta no incluyó Retry-After y los clientes reintentaron de inmediato	Devuelve Retry-After y aplica retroceso con jitter en el cliente.

■ Seguridad, ética y costo

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

■ Preguntas de comprobación

1. Con un TTL de 3600 s, ¿cuál es el peor caso de convergencia tras cambiar una IP, y por qué no es una hora?
2. ¿Qué demuestra exactamente un certificado TLS válido, y qué afirmación sobre el sitio no respalda?
3. Un proxy devuelve 502 y otro 504. ¿En cuál mirarías los logs del origen y en cuál sus métricas de latencia?
4. ¿Por qué no-cache no impide almacenar, y qué cabecera sí lo impide?
5. ¿Cuántos RTT tiene una petición HTTPS en frío y cuáles se eliminan reutilizando la conexión?

Referencias

- Mockapetris, P. (1987). RFC 1035: Domain Names — Implementation and Specification. <https://www.rfc-editor.org/rfc/rfc1035>

- Rescorla, E. (2018). RFC 8446: TLS 1.3 — saludo de 1-RTT, 0-RTT y sus riesgos de repetición. <https://www.rfc-editor.org/rfc/rfc8446>
- Fielding, R. y Reschke, J., eds. (2022). RFC 9110: HTTP Semantics — semántica de códigos de estado. <https://www.rfc-editor.org/rfc/rfc9110>
- Fielding, R. et al., eds. (2022). RFC 9111: HTTP Caching — Cache-Control, validadores y respuestas 304. <https://www.rfc-editor.org/rfc/rfc9111>
- Grigorik, I. (2013). High Performance Browser Networking, caps. 1-4 — coste en RTT de DNS, TCP y TLS. <https://hpbn.co/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 00 en PDF · Manual integral

Anterior	Índice	Siguiente
← 005 · Redes por capas, TCP/IP, puertos y sockets	Parte 00 · Programa	007 · Linux: usuarios, permisos, servicios y logs →

Evaluación — 006 DNS, HTTP, HTTPS y TLS de extremo a extremo

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 traza-de-solicitud 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.

007 — Linux: usuarios, permisos, servicios y logs

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 4 Laboratorio: linux · Estado: EXECUTABLECORE

Propósito

Entender el modelo de identidad y privilegio de Linux, y el ciclo de vida de un servicio bajo systemd, porque son la base literal de todo lo que viene: los contenedores de la parte 05 son procesos Linux con espacios de nombres, y el mínimo privilegio de IAM en la parte 11 es esta misma idea trasladada a la nube.

Resultados de aprendizaje

Al finalizar podrás:

1. Descomponer un modo de permisos octal y explicar por qué el bit de ejecución significa cosas distintas en fichero y en directorio.
2. Sustituir el uso de root por capacidades concretas, justificando cuál se necesita y por qué.
3. Definir una unidad de systemd con reinicio, límites y usuario propio, y explicar cada directiva.
4. Consultar logs con journalctl filtrando por unidad, prioridad y ventana temporal.
5. Diagnosticar un servicio que no arranca distinguiendo permisos, dependencias y configuración.

Conceptos centrales

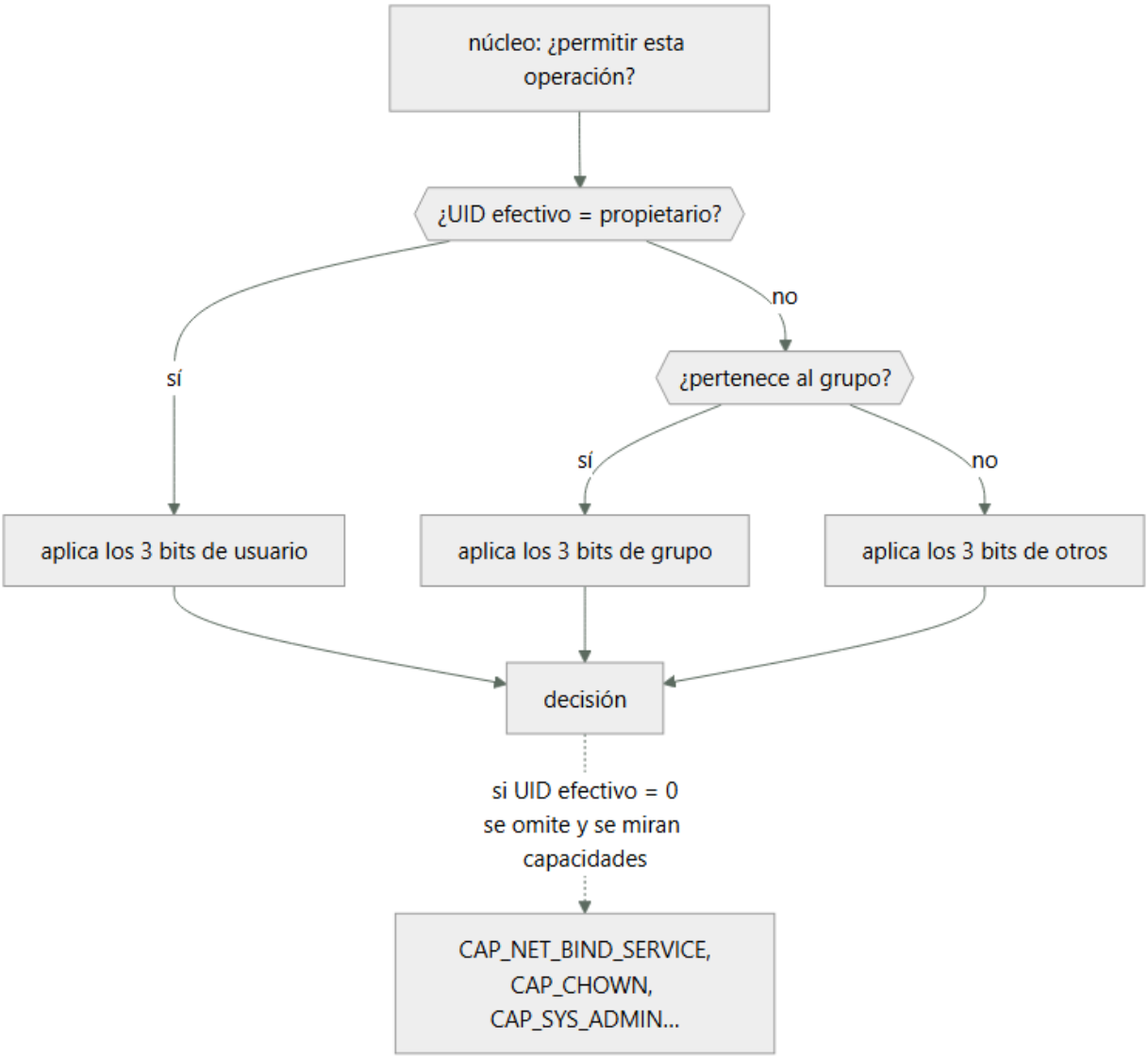
Concepto	Comprensión verificable
UID efectivo	Identificador con el que el núcleo evalúa cada comprobación de permiso. Puede diferir del UID real tras un setuid, y es el que importa para decidir si una operación se permite.
capacidad	Fragmento del poder histórico de root, otorgable por separado. CAPNETBINDSERVICE permite abrir puertos bajo 1024 sin conceder las otras 40 capacidades que root arrastra.
unidad	Objeto que systemd gestiona: servicio, socket, temporizador, punto de montaje o destino. Se declara de forma descriptiva y systemd deduce el orden de arranque a partir de las dependencias.
journal	Registro binario, indexado y estructurado de systemd. Cada entrada lleva metadatos —unidad, PID, UID, prioridad— consultables como campos, no como texto.
umask	Máscara que se resta de los permisos por defecto al crear ficheros. Con umask 022, un fichero nace 644 y un directorio 755; explica por qué lo que creas no es de escritura para el grupo.

Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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. Permisos: tres tríos y una regla que sorprende

Cada objeto del sistema de ficheros tiene propietario, grupo y tres tríos de bits. El núcleo evalúa el primer trío que aplica y se detiene ahí:

```
$ ls -l /opt/cloudshop/config.yaml
-rw-r----- 1 cloudshop cloudshop 412 Aug  1 09:12 config.yaml
#  ↑↑↑↑↑↑↑↑↑
#  rw-  usuario cloudshop: leer y escribir
#     r--  grupo cloudshop: solo leer
#        ---  otros: nada
```

En octal, 640. La consecuencia contraintuitiva: si eres el propietario y el trío de usuario deniega, no te salva pertenecer al grupo. Un fichero 0640 propiedad de ana y grupo ana es ilegible para ana si el primer trío fuera ---, aunque su grupo pueda leerlo.

El bit de ejecución significa cosas distintas según el objeto:

Bit	En fichero	En directorio
r	Leer el contenido	Listar los nombres
w	Modificar el contenido	Crear y borrar entradas
x	Ejecutarlo	Atravesarlo para llegar a su interior

De ahí dos efectos que confunden a diario:

- Un directorio r-- deja ver los nombres pero no consultar los metadatos de lo que contiene: ls funciona y ls -l da «permiso denegado».
- Para borrar un fichero no hace falta permiso sobre el fichero, sino w sobre el directorio que lo contiene. Por eso /tmp lleva el sticky bit (1777): permite a todos crear, pero solo al propietario borrar lo suyo.

2. root no es un usuario: es la ausencia de comprobaciones

Cuando el UID efectivo es 0, el núcleo omite la comprobación de permisos. Eso no es un privilegio más: es no tener ninguno que aplicar. Linux fragmentó ese poder en unas 40 capacidades otorgables por separado:

Capacidad	Permite	Uso legítimo
CAPNETBINDSERVICE	Abrir puertos < 1024	Un servidor web en el 80
CAPCHOWN	Cambiar el propietario	Gestores de paquetes
CAPSYSADMIN	Montar, espacios de nombres…	Casi equivale a root

El caso que se repite: un servicio quiere escuchar en el 443 y alguien lo ejecuta como root «porque si no, no puede». La alternativa correcta cede solo lo necesario:

```
[Service]
User=cloudshop
AmbientCapabilities=CAP_NET_BIND_SERVICE
CapabilityBoundingSet=CAP_NET_BIND_SERVICE
```

Si ese proceso resulta vulnerable, el atacante obtiene la capacidad de abrir puertos bajos y nada más: no puede leer /etc/shadow, ni cargar módulos, ni cambiar propietarios. CAPSYSADMIN es la excepción: concederla equivale prácticamente a conceder root, y verla en un manifiesto debe activar la misma alarma que privileged: true en la parte 06.

3. systemd declara el estado deseado, no los pasos

Una unidad no describe cómo arrancar: describe qué debe cumplirse. systemd deduce el orden a partir de las dependencias y paraleliza el resto.

```
[Unit]
Description=API de CloudShop
After=network-online.target          # orden: después de que haya red
Wants=network-online.target          # dependencia débil: si falla, seguimos

[Service]
Type=notify                          # el proceso avisa cuando está listo
User=cloudshop
ExecStart=/opt/cloudshop/bin/api
Restart=on-failure                   # no reiniciar si salió con 0
RestartSec=5
StartLimitBurst=5                    # 5 intentos…
StartLimitIntervalSec=300            # …en 5 minutos, o se rinde

NoNewPrivileges=true                 # ningún hijo escala privilegios
ProtectSystem=strict                 # todo el sistema en solo lectura
ProtectHome=true
PrivateTmp=true                      # /tmp propio, aislado
ReadWritePaths=/var/lib/cloudshop    # única excepción de escritura
MemoryMax=512M

[Install]
WantedBy=multi-user.target
```

Dos distinciones que evitan incidentes:

- Requires frente a Wants: Requires arrastra la caída del dependiente si la dependencia falla; Wants no. Usar Requires a la ligera propaga fallos en cascada.

- After no implica dependencia: solo fija orden. Un servicio con After=X pero sin Wants=X arranca aunque X no exista.

StartLimitBurst merece atención: sin él, un servicio que falla al arrancar entra en bucle de reinicio consumiendo CPU. Con él, systemd se rinde y deja el fallo visible en vez de esconderlo tras reintentos infinitos.

4. El journal es estructurado: consúltalo como tal

journalctl no busca texto: filtra campos indexados. Tratarlo con grep desperdicia su ventaja y produce falsos negativos.

```
# Errores y peores de una unidad en la última hora
$ journalctl -u cloudshop -p err --since "1 hour ago"

# Solo el arranque actual, siguiendo en vivo
$ journalctl -u cloudshop -b -f

# Salida estructurada para automatizar
$ journalctl -u cloudshop -o json --since today | jq -r 'select(.PRIORITY<="3") | .MESSAGE'

# Qué escribió un PID concreto, aunque ya no exista
$ journalctl _PID=4877
```

Las prioridades siguen la escala syslog, de 0 (emergencia) a 7 (depuración); -p err incluye 0 a 3. Un servicio que escribe todo como info hace inútil este filtro: la prioridad es parte del contrato del log, igual que stdout y stderr lo eran en la clase 002.

Dos límites operativos que importan: el journal es local y rotatorio, con tamaño acotado por SystemMaxUse. Si el disco se llena, se descartan las entradas más antiguas. Por eso en la parte 10 los logs se envían fuera del host: un log que solo existe en la máquina que falló no sirve para el postmortem.

5. Diagnóstico de un servicio que no arranca

El orden ahorra horas, porque cada paso descarta una familia de causas:

```
$ systemctl status cloudshop          # 1. ¿qué dice systemd?
$ journalctl -u cloudshop -n 50 -p warning   # 2. ¿qué dijo el proceso?
$ systemd-analyze verify cloudshop.service   # 3. ¿la unidad es válida?
$ sudo -u cloudshop /opt/cloudshop/bin/api   # 4. ¿arranca a mano con ese usuario?
$ systemctl show cloudshop -p ExecMainStatus -p ExecMainCode
```

El paso 4 es el que más separa: si a mano funciona con el mismo usuario, el problema está en el entorno de la unidad —PATH, variables, directorio de trabajo, endurecimiento— y no en la aplicación. Es exactamente el fallo de la clase 002.

Los códigos más frecuentes y su lectura inmediata:

ExecMainStatus	Causa habitual
203	ExecStart no existe o no es ejecutable
200	El directorio de trabajo no existe
208	User= no existe
226	El endurecimiento bloqueó una ruta necesaria
1-125	Fallo propio de la aplicación: ahora sí, mira sus logs

Los códigos 200-241 son de systemd, no de tu programa: significan que el proceso nunca llegó a ejecutarse.

Ejemplo trabajado

La API de CloudShop arranca en desarrollo y falla en el servidor nuevo. El log de la aplicación está vacío.

```
$ systemctl status cloudshop
● cloudshop.service - API de CloudShop
     Active: failed (Result: exit-code)
    Process: 8812 ExecStart=/opt/cloudshop/bin/api (code=exited, status=226/NAMESPACE)
```

226/NAMESPACE: no es la aplicación. systemd impidió el arranque por el endurecimiento antes de ejecutar una sola línea. El log vacío ahora se explica: el proceso nunca corrió.

```
$ journalctl -u cloudshop -n 5 -p err
cloudshop.service: Failed to set up mount namespacing:
  /var/log/cloudshop: Read-only file system
```

La unidad declara ProtectSystem=strict, que monta todo el sistema en solo lectura, y la aplicación escribe en /var/log/cloudshop, que no está en las excepciones:

```
$ systemctl show cloudshop -p ProtectSystem -p ReadWritePaths
ProtectSystem=strict
ReadWritePaths=/var/lib/cloudshop
```

Hay dos correcciones posibles y no son equivalentes:

```
# A) Ampliar la excepción: mantiene el endurecimiento
ReadWritePaths=/var/lib/cloudshop /var/log/cloudshop

# B) Escribir al journal por stdout: elimina la necesidad
# (se quita la ruta de log de la aplicación; systemd captura stdout)
```

Se elige B. Razón operativa, no estética: un fichero de log local se pierde cuando la instancia se recicla, no rota solo, y llena el disco. Escribir a stdout deja que systemd lo estructure y que el recolector de la parte 10 lo envíe fuera del host. Es además lo que exigirá el contenedor de la parte 05, donde escribir logs a fichero es directamente un antipatrón.

Comprobación final del privilegio real que conserva el servicio:

```
$ systemctl restart cloudshop && systemctl show cloudshop -p ExecMainStatus
ExecMainStatus=0
$ grep CapEff /proc/$(systemctl show -p MainPID --value cloudshop)/status
CapEff:■0000000000000400          # solo CAP_NET_BIND_SERVICE
```

0x400 es el bit 10, exactamente CAPNETBINDSERVICE. Si esa API resultara vulnerable, el atacante hereda el poder de abrir puertos bajos y nada más. Ese es el mismo razonamiento que en la parte 11 se aplicará a un rol de IAM: no «qué necesita para funcionar», sino qué obtiene quien lo comprometa.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/007-linux-usuarios-permisos-servicios-y-logs/lab.py
```

El laboratorio selecciona el motor de práctica linux 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 servicio-auditado 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 servicio con identidad, permisos y logs inspeccionados. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye servicio-auditado 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 se ejecuta como root solo para escuchar en el puerto 443	Se concedió todo el poder para obtener una capacidad concreta	Usa User= propio más AmbientCapabilities=CAPNETBINDSERVICE.
El servicio falla con 226/NAMESPACE y el log de la aplicación está vacío	El endurecimiento de systemd bloqueó una ruta antes de ejecutar el proceso	Los códigos 200-241 son de systemd: revisa ReadWritePaths y ProtectSystem, no el código.
Un servicio que falla al arrancar consume CPU en bucle de reinicio	Restart=always sin límite de intentos	Usa Restart=on-failure con StartLimitBurst y StartLimitIntervalSec.
Tras un incidente no hay logs de la máquina afectada	El journal es local y rotatorio; se perdió con la instancia	Envía los logs fuera del host; el journal sirve para diagnóstico en vivo, no para postmortem.
Un usuario no puede borrar un fichero del que es propietario	Borrar exige w sobre el directorio contenedor, no sobre el fichero	Revisa los permisos del directorio; el sticky bit en /tmp explota justamente esta regla.

■ 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. Un fichero es 0640, propiedad de ana, grupo ana. ¿Puede ana escribirlo? ¿Y si el trío de usuario fuera --- y el de grupo rw-?
2. ¿Por qué ls funciona y ls -l falla en un directorio con permisos r--?
3. ¿Qué obtiene un atacante que compromete un proceso con CAPNETBINDSERVICE frente a uno que corre como root?
4. ¿Qué diferencia práctica hay entre Requires= y Wants=, y cuál propaga fallos en cascada?
5. Un servicio devuelve ExecMainStatus=226. ¿Tiene sentido revisar el código de la aplicación? ¿Por qué?

Referencias

- Kerrisk, M. capabilities(7) — catálogo completo de capacidades de Linux y sus implicaciones. <https://man7.org/linux/man-pages/man7/capabilities.7.html>
- systemd (2024). systemd.exec(5) — directivas de endurecimiento: ProtectSystem, ReadWritePaths, NoNewPrivileges. <https://www.freedesktop.org/software/systemd/man/systemd.exec.html>
- systemd (2024). systemd.service(5) — códigos de salida 200-241 y política de reinicio. <https://www.freedesktop.org/software/systemd/man/systemd.service.html>
- systemd (2024). journalctl(1) — filtrado por campos, prioridades y arranque. <https://www.freedesktop.org/software/systemd/man/journalctl.html>
- Ward, B. (2021). How Linux Works, 3.ª ed., caps. 6-7 — arranque, systemd y configuración del sistema.
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 00 en PDF · Manual integral

Evaluación — 007 Linux: usuarios, permisos, servicios y logs

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 servicio-auditado 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.

008 — Virtualización, hipervisores e imágenes

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 4 Laboratorio: virtualization · Estado: EXECUTABLECORE

Propósito

Distinguir los tres mecanismos que la industria llama «virtualización» —máquinas virtuales, contenedores y microVM— por lo que realmente aíslan y lo que realmente cuestan. Sin esa distinción, la elección entre EC2, Fargate y Lambda en la parte 17 se hace por moda; con ella, se hace por requisito de aislamiento y perfil de arranque.

Resultados de aprendizaje

Al finalizar podrás:

1. Explicar qué aísla un hipervisor que un espacio de nombres no, y qué superficie de ataque queda en cada caso.
2. Comparar tiempo de arranque, sobrecarga y densidad de VM, contenedor y microVM con órdenes de magnitud.
3. Justificar el uso de microVM cuando se ejecuta código de terceros no confiable.
4. Describir cómo el almacenamiento por capas hace que 50 contenedores de la misma imagen ocupen poco más que uno.
5. Reconocer que una imagen es una lista ordenada de capas más un manifiesto, y por qué el digest es su única referencia inmutable.

Conceptos centrales

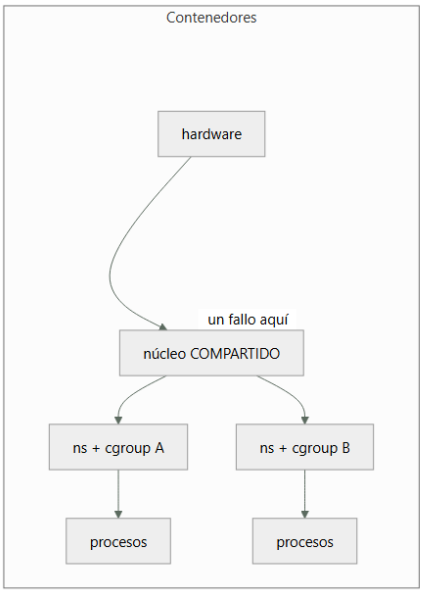
Concepto	Comprensión verificable
hipervisor	Capa que presenta hardware virtual a varios sistemas operativos huésped. De tipo 1 corre sobre el hardware desnudo; de tipo 2, sobre un sistema anfitrión. El aislamiento lo impone la CPU, no el software.
espacio de nombres	Mecanismo del núcleo Linux que da a un grupo de procesos su propia vista de un recurso: PID, red, montajes, usuarios. Aísla lo que se ve, no lo que se ejecuta: el núcleo sigue siendo compartido.
cgroup	Subsistema que limita y contabiliza CPU, memoria y E/S de un grupo de procesos. Complementa a los espacios de nombres: estos aíslan la vista y aquel reparte el recurso.
microVM	Máquina virtual reducida al mínimo dispositivo emulado para arrancar en decenas de milisegundos. Combina el aislamiento de hardware del hipervisor con un perfil de arranque cercano al del contenedor.
capa	Conjunto de diferencias del sistema de ficheros, inmutable y direccionado por su digest. Las imágenes que comparten una capa base la almacenan y descargan una sola vez.

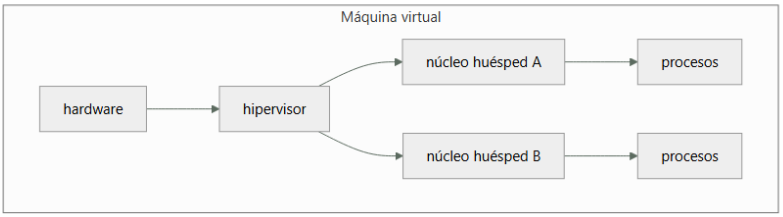
Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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

■ Flujo de razonamiento





Desarrollo

1. Dos aislamientos que no son comparables

La diferencia decisiva cabe en una frase: una máquina virtual tiene su propio núcleo; un contenedor comparte el del anfitrión.

En una VM, el huésped ejecuta su núcleo y el hipervisor le presenta hardware virtual. El aislamiento lo hace cumplir la propia CPU mediante extensiones de virtualización (Intel VT-x, AMD-V): para escapar hay que romper el hipervisor, que expone una interfaz reducida.

En un contenedor no hay huésped. Son procesos normales del anfitrión con una vista restringida por espacios de nombres y un reparto impuesto por cgroups. La superficie de ataque es toda la interfaz de llamadas al sistema del núcleo Linux: más de 300 llamadas, algunas con historial de vulnerabilidades de escalada.

```
$ docker run --rm alpine uname -r
6.8.0-45-generic          # el núcleo del anfitrión, no de la imagen
$ uname -r
6.8.0-45-generic          # idéntico
```

La imagen aporta bibliotecas y binarios; el núcleo siempre es el del anfitrión. De ahí tres consecuencias que reaparecerán en la parte 05: no se pueden ejecutar contenedores Windows sobre un núcleo Linux; un contenedor no puede cargar módulos; y una vulnerabilidad del núcleo afecta simultáneamente a todos los contenedores de esa máquina.

2. El coste del aislamiento, en números

Los órdenes de magnitud, no las cifras exactas, son lo que hay que retener:

	Máquina virtual	Contenedor	microVM (Firecracker)
Arranque	30-60 s	50-200 ms	125 ms
Sobrecarga de memoria	512 MB-2 GB	1-10 MB	< 5 MB
Densidad por host	decenas	miles	miles
Aislamiento	hardware	núcleo compartido	hardware
Núcleo	propio	del anfitrión	propio, reducido

Una VM arranca lento porque hace lo mismo que una máquina física: firmware, cargador, núcleo, init, servicios. Un contenedor solo ejecuta un proceso más con una vista distinta.

Firecracker rompe la disyuntiva: elimina casi todos los dispositivos emulados —sin BIOS, sin USB, sin VGA— y deja solo lo mínimo. Arranca en unos 125 ms con menos de 5 MB de sobrecarga, y mantiene el aislamiento por hardware. Por eso AWS lo usa bajo Lambda y Fargate: ejecutan código de clientes distintos en el mismo host y el aislamiento por núcleo compartido no sería aceptable.

La regla de decisión: si ejecutas tu código, el contenedor basta. Si ejecutas código de terceros que no controlas, necesitas frontera de hardware.

3. Espacios de nombres y cgroups: qué aísla cada cosa

Un contenedor no es una primitiva del núcleo. Es la combinación de varios mecanismos independientes que se pueden usar por separado:

Espacio de nombres	Aísla
pid	La tabla de procesos: dentro, tu proceso es el PID 1
net	Interfaces, rutas, reglas de firewall y puertos
mnt	El árbol de montajes visible
uts	Nombre de host y de dominio
ipc	Colas de mensajes y memoria compartida
user	El mapeo de UID: root dentro puede ser un UID sin privilegios fuera
cgroup	La vista de la propia jerarquía de cgroups

Se ven directamente:

```
$ docker run --rm -d --name demo alpine sleep 300
$ pid=$(docker inspect -f '{{.State.Pid}}' demo)
$ sudo ls -l /proc/$pid/ns/
lrwxrwxrwx net -> 'net:[4026532501]'      # espacio propio
lrwxrwxrwx pid -> 'pid:[4026532503]'
lrwxrwxrwx user -> 'user:[4026531837]'    # el MISMO del anfitrión
```

Ese último renglón es el hallazgo importante: por defecto el espacio de usuarios no está aislado, así que el UID 0 dentro del contenedor es el UID 0 del anfitrión. Si el proceso escapa, escapa como root. Activar user remapea root a un UID sin privilegios y es la mitigación más rentable frente a una fuga.

Los cgroups son la otra mitad: sin un límite de memoria, un contenedor consume la del anfitrión y el OOM killer mata al que más memoria use, que puede ser otro contenedor inocente. Ese es el origen del código 137 de la clase 002.

4. Capas: por qué 50 contenedores caben en el disco de uno

Una imagen es una lista ordenada de capas inmutables, cada una nombrada por el digest de su contenido, más un manifiesto que las enumera. Al arrancar, el runtime apila esas capas en solo lectura y añade encima una capa de escritura propia del contenedor, mediante copy-on-write con OverlayFS.

```
$ docker image inspect cloudshop:2.4 -f '{{range .RootFS.Layers}}{{println .}}{{end}}'
sha256:8e012198eba1...      # base: alpine 3.20
sha256:4f2a9c73b8d5...      # dependencias
sha256:7d2b8e91c4a2...      # aplicación
```

Si 50 contenedores usan esta imagen, las tres capas base se almacenan una vez. Cada contenedor solo añade lo que escribe:

```
imagen                      180 MB   (compartida por los 50)
capa de escritura por contenedor ~2 MB
-------------------------------------------
50 contenedores              180 + 50×2 = 280 MB
50 VM con el mismo software  50 × 180  = 9.000 MB
```

De aquí sale una regla de construcción que dominará la parte 05: ordena las capas de menos a más volátil. Si copias el código antes de instalar dependencias, cualquier cambio de una línea invalida la capa de dependencias y obliga a reconstruirla y redescargarla entera.

Y de aquí sale también por qué el digest importa: una etiqueta como :2.4 es mutable —puede reapuntarse mañana— mientras que sha256:... identifica exactamente ese contenido. Es la misma distinción entre rama y commit de la clase 003.

5. Qué elegir, y con qué criterio

La decisión se toma con dos preguntas, en este orden:

1. ¿De quién es el código que se ejecuta?
   - Propio o auditado → contenedor.
   - De terceros, sin auditar, o multi-cliente → frontera de hardware: VM o microVM.
2. ¿Qué perfil de arranque exige la carga?
   - Larga vida, estado en disco, núcleo específico → VM.
   - Efímera, escalado por ráfagas, sin estado → contenedor o microVM.

Situación	Elección	Por qué
API propia con escalado horizontal	Contenedor	Arranque rápido, densidad alta, código confiable
Ejecutar plugins de clientes	microVM	Frontera de hardware entre inquilinos
Base de datos con disco y ajuste de núcleo	VM	Control del núcleo y del almacenamiento
Trabajo por lotes de minutos	Contenedor	Sobrecarga de arranque despreciable
Función invocada miles de veces por segundo	microVM	Aislamiento sin pagar 30 s de arranque

Lo que no es criterio válido: «los contenedores son más modernos». Un contenedor y una VM resuelven problemas de aislamiento distintos, y el coste de equivocarse no aparece en las pruebas: aparece cuando alguien escapa del núcleo compartido.

Ejemplo trabajado

CloudShop quiere ofrecer reglas de descuento programables por sus comercios: cada uno sube un fragmento de código que se ejecuta en cada compra. El equipo propone contenedores «porque ya usamos Kubernetes». Se evalúa con las dos preguntas.

Pregunta 1 — ¿de quién es el código? De terceros y sin auditar. El aislamiento por núcleo compartido pone en juego toda la interfaz de llamadas al sistema:

```
$ docker run --rm alpine sh -c 'grep -c . /proc/kallsyms; ls /proc/sys/kernel | wc -l'
#  la superficie visible es la del núcleo del anfitrión
```

Se cuantifica el riesgo con datos, no con opinión. Escapes históricos por CVE de contenedores conocidos: CVE-2019-5736 (reescritura de runc desde el contenedor), CVE-2022-0492 (escalada por cgroups v1), CVE-2024-21626 (fuga de descriptor en runc). Todos permitieron ejecución en el anfitrión desde dentro.

Pregunta 2 — ¿qué perfil de arranque? Una regla se ejecuta por compra: hasta 600 invocaciones por segundo en pico, de unos 20 ms cada una. Una VM clásica queda descartada por aritmética:

```
arranque de VM      ≈ 40 s
trabajo útil        ≈ 0,02 s
sobrecarga          = 40 / 0,02 = 2.000x    inviable
```

Y el modelo de preaprovisionamiento tampoco cierra:

```
600 inv/s × 0,02 s = 12 ejecuciones concurrentes en media
VM preaprovisionadas para pico (×3)        = 36 VM permanentes
memoria: 36 × 1 GB de sobrecarga           = 36 GB solo de hipervisor
```

Con microVM:

```
arranque de microVM ≈ 0,125 s
sobrecarga          = 0,125 / 0,02 = 6,25x   asumible con reutilización
memoria: 36 × 5 MB                           = 180 MB
```

180 MB frente a 36 GB, con el mismo aislamiento de hardware. Ese factor de 200 es lo que hace viable el producto.

Decisión registrada: microVM por comercio, reutilizada entre invocaciones del mismo comercio y destruida al cambiar de comercio. El contenedor se descarta no por rendimiento —sería mejor— sino porque el requisito es ejecutar código no confiable de inquilinos distintos, y ahí el núcleo compartido no es una frontera aceptable.

Límite explícito de la decisión: si en el futuro el código dejara de ser de terceros —por ejemplo, si se pasara a un lenguaje de reglas restringido que no permite ejecución arbitraria— la pregunta 1 cambiaría de respuesta y el contenedor volvería a ser correcto.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/008-virtualizacion-hipervisores-e-imagenes/lab.py
```

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

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa comparativa-de-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 comparación medida de aislamiento y consumo. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye comparativa-de-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
Se ejecuta código de terceros en contenedores compartiendo núcleo	Se confundió aislamiento de vista con aislamiento de seguridad	Para código no confiable usa frontera de hardware: microVM o VM dedicada.
Un contenedor comprometido da acceso root al anfitrión	El espacio de nombres de usuario no estaba activado, así que UID 0 dentro es UID 0 fuera	Activa el remapeo de usuarios y ejecuta el proceso con un UID sin privilegios.
Un contenedor consume toda la memoria y el núcleo mata a otro distinto	No había límite de cgroup: el OOM killer elige por consumo, no por culpa	Define límites de memoria en todos los contenedores, no solo en los sospechosos.
Cada cambio de una línea de código reconstruye y redescarga cientos de MB	El código se copió antes de instalar dependencias e invalida la capa de estas	Ordena las capas de menos a más volátil: dependencias primero, código al final.
Un despliegue reproducible deja de serlo sin que nadie cambie nada	Se referenció la imagen por etiqueta mutable en vez de por digest	Fija imagen@sha256:... en todo lo que deba ser reproducible.

■ 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 aísla un hipervisor que un espacio de nombres no, y qué superficie de ataque queda en cada caso?
2. ¿Por qué no se pueden ejecutar contenedores Windows sobre un núcleo Linux, si la imagen trae todas sus bibliotecas?
3. Una carga se invoca 600 veces por segundo y dura 20 ms. ¿Por qué una VM clásica es inviable y una microVM no?
4. Por defecto, ¿qué UID del anfitrión corresponde a root dentro de un contenedor, y qué mitigación lo cambia?
5. ¿Por qué 50 contenedores de la misma imagen no ocupan 50 veces su tamaño en disco?

Referencias

- Agache, A. et al. (2020). Firecracker: Lightweight Virtualization for Serverless Applications. USENIX NSDI — cifras de arranque y sobrecarga. <https://www.usenix.org/conference/nsdi20/presentation/agache>
- Kerrisk, M. namespaces(7) — los siete espacios de nombres de Linux y su semántica. <https://man7.org/linux/man-pages/man7/namespaces.7.html>
- Kerrisk, M. cgroups(7) — límites, contabilidad y jerarquía de control de recursos. <https://man7.org/linux/man-pages/man7/cgroups.7.html>
- Open Container Initiative (2024). Image Specification — manifiesto, capas y direccionamiento por digest. <https://github.com/opencontainers/image-spec/blob/main/spec.md>
- Rice, L. (2020). Container Security, caps. 4-8 — espacios de nombres, capacidades y vectores de escape.
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 00 en PDF · Manual integral

Anterior	Índice	Siguiente
← 007 · Linux: usuarios, permisos, servicios y logs	Parte 00 · Programa	009 · APIs REST, autenticación y contratos →

Evaluación — 008 Virtualización, hipervisores e 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 comparativa-de-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.

009 — APIs REST, autenticación y contratos

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 4 Laboratorio: api · Estado: EXECUTABLECORE

Propósito

Diseñar y consumir APIs HTTP tratando el contrato como la unidad que de verdad se despliega. Toda la nube se opera por API —la consola es solo un cliente más— y las decisiones de esta clase sobre idempotencia, paginación, autenticación y versionado son las que hacen que una automatización sobreviva a un reintento, a un fallo parcial y a un cambio del proveedor.

Resultados de aprendizaje

Al finalizar podrás:

1. Elegir el método HTTP correcto a partir de sus propiedades de seguridad e idempotencia, no de la costumbre.
2. Hacer idempotente una operación que por naturaleza no lo es, mediante clave de idempotencia.
3. Distinguir autenticación de autorización y explicar qué prueba un token y qué no.
4. Paginar un conjunto grande sin perder ni duplicar elementos cuando la colección cambia durante el recorrido.
5. Evolucionar un contrato sin romper a los consumidores existentes, sabiendo qué cambio es compatible y cuál no.

Conceptos centrales

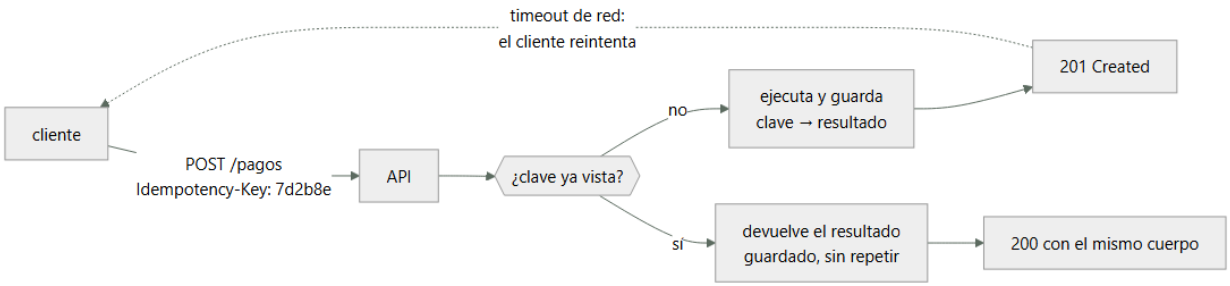
Concepto	Comprensión verificable
método seguro	El que no altera el estado del servidor: GET, HEAD, OPTIONS. Permite que proxies y navegadores lo repitan o lo precarguen sin consecuencias, lo que a su vez prohíbe usar GET para operaciones con efecto.
idempotencia	Propiedad por la que N ejecuciones idénticas dejan el mismo estado que una. PUT y DELETE lo son por definición; POST no, y por eso necesita una clave explícita para poder reintentarse.
clave de idempotencia	Identificador único que el cliente genera y envía para que el servidor reconozca un reintento y devuelva el resultado original en vez de repetir el efecto.
token portador	Credencial que autoriza a quien la presente, sin más prueba. Quien lo roba lo puede usar: de ahí que su vida deba ser corta y su transporte siempre cifrado.
cursor	Marca opaca de posición en una colección, estable frente a inserciones y borrados. Sustituye al desplazamiento numérico, que salta o repite elementos cuando la colección cambia entre páginas.

Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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. Los métodos tienen propiedades, no solo nombres

Elegir el método no es cuestión de estilo: sus propiedades determinan qué pueden hacer con la petición los proxies, los navegadores y las bibliotecas de reintento.

Método	Seguro	Idempotente	Consecuencia práctica
GET	Sí	Sí	Cacheable; un proxy puede repetirlo sin avisar
HEAD	Sí	Sí	Como GET sin cuerpo; útil para comprobar existencia
PUT	No	Sí	Reintentable sin riesgo: sustituye el recurso entero
DELETE	No	Sí	Reintentable: borrar dos veces deja el mismo estado
POST	No	No	No reintentable sin protección adicional
PATCH	No	No garantizada	Depende del formato del parche

La fila de GET explica un incidente clásico: exponer GET /pedidos/42/cancelar parece cómodo hasta que un precargador del navegador, un rastreador o un proxy lo visitan y cancelan pedidos que nadie pidió cancelar. Si tiene efecto, no puede ser GET.

Y la fila de POST explica el problema central de esta clase: es el único método común que no se puede reintentar con seguridad, y es justo el que se usa para crear pagos, pedidos y recursos.

2. Idempotencia: la propiedad que hace segura una red poco fiable

Un cliente envía POST /pagos, la red se corta antes de la respuesta y el cliente no sabe si el pago se ejecutó. Solo tiene dos opciones malas: reintentar y arriesgarse a cobrar dos veces, o no reintentar y arriesgarse a no cobrar.

La solución es que el cliente genere un identificador único por intención y lo repita en cada reintento:

```
POST /pagos HTTP/1.1
Idempotency-Key: <uuid v4 que genera el cliente>
Content-Type: application/json

{"pedido": "A-1042", "importe": "149.90", "moneda": "CLP"}
```

El servidor guarda la clave junto al resultado. Al recibirla de nuevo:

1. Si no la ha visto, ejecuta y persiste clave → resultado en la misma transacción que el efecto.
2. Si ya la vio y terminó, devuelve el resultado original sin repetir el cobro.
3. Si ya la vio y sigue en curso, responde 409 Conflict.

El punto 1 es donde suele fallar la implementación: si el registro de la clave y el cobro no son atómicos, existe una ventana en la que el cobro ocurrió y la clave no se guardó, y el reintento cobra otra vez. La clave debe persistirse en la misma transacción que el efecto que protege.

Es el mismo mecanismo que en la parte 09 aparecerá como deduplicación en sistemas de mensajería, y por la misma razón: la entrega «exactamente una vez» no existe a nivel de red; se construye con idempotencia en el receptor.

3. Autenticación no es autorización

Son dos preguntas distintas y se resuelven en momentos distintos:

- Autenticación: ¿quién eres? Se resuelve una vez y produce una credencial.
- Autorización: ¿puedes hacer esto sobre este recurso? Se resuelve en cada petición.

Saltarse la segunda produce la vulnerabilidad más común de las APIs, catalogada como BOLA (Broken Object Level Authorization, primer puesto del OWASP API Security Top 10):

```
# vulnerable: autentica pero no autoriza sobre el objeto
@app.get("/pedidos/{id}")
def ver(id: str, usuario = Depends(autenticar)):
    return db.pedidos.get(id)          # cualquiera autenticado ve CUALQUIER pedido

# correcto: la autorización es por objeto
@app.get("/pedidos/{id}")
def ver(id: str, usuario = Depends(autenticar)):
    pedido = db.pedidos.get(id)
    if pedido is None or pedido.cliente_id != usuario.cliente_id:
        raise HTTPException(404)        # 404, no 403: no revelar existencia
    return pedido
```

Devolver 404 en vez de 403 es deliberado: un 403 confirma que el recurso existe y permite enumerar identificadores ajenos. Es la misma lógica por la que un formulario de acceso no debe decir «esa contraseña es incorrecta» frente a «ese usuario no existe».

Los códigos correctos: 401 significa «no sé quién eres» y admite reintento con credenciales; 403 significa «sé quién eres y no puedes», y reintentar no ayuda.

4. Paginación: por qué el desplazamiento numérico pierde datos

GET /pedidos?offset=100&limit=50 parece razonable y falla en cuanto la colección cambia durante el recorrido. Con orden descendente por fecha:

```
t0  el cliente lee offset=0..49   (elementos 1-50)
t1  se insertan 3 pedidos nuevos al principio
t2  el cliente lee offset=50..99
    → los elementos 48, 49 y 50 se han desplazado y se REPITEN
    → si en vez de insertar se hubieran borrado 3, se PERDERÍAN
```

Y hay un segundo problema, de coste: OFFSET 100000 obliga a la base de datos a leer y descartar cien mil filas antes de devolver la página. El coste crece linealmente con la profundidad.

La paginación por cursor usa una marca estable —la clave del último elemento— en vez de una posición:

```
GET /pedidos?limit=50
{"datos": [...], "siguiente": "eyJ0IjoiMjAyNi0wOC0wMVQwOToxMiJ9"}

GET /pedidos?limit=50&cursor=eyJ0IjoiMjAyNi0wOC0wMVQwOToxMiJ9
```

La consulta pasa a ser WHERE (creado, id) < (:t, :id) ORDER BY creado DESC, id DESC LIMIT 50, que usa índice y cuesta lo mismo en la página 1 que en la 10.000. El cursor debe ser opaco para el cliente: si se documenta su estructura, se convierte en parte del contrato y ya no se puede cambiar.

El orden debe incluir un desempate único (id): sin él, dos filas con la misma marca de tiempo pueden repetirse o perderse en el corte de página.

5. Evolucionar sin romper: qué cambio es compatible

Un contrato desplegado tiene consumidores que no controlas. La distinción práctica:

Cambio	¿Compatible?	Por qué
Añadir un campo opcional a la respuesta	Sí	Los clientes ignoran lo que no conocen
Añadir un parámetro opcional	Sí	Su ausencia mantiene el comportamiento
Añadir un valor nuevo a un enumerado	No	Los clientes con switch exhaustivo fallan
Renombrar o eliminar un campo	No	Rotura directa

Cambio	¿Compatible?	Por qué
Cambiar un tipo (número → cadena)	No	Rotura de deserialización
Hacer obligatorio un campo opcional	No	Rotura para quien no lo enviaba
Endurecer una validación	No	Peticiones antes válidas empiezan a fallar

La fila del enumerado es la que más sorprende: parece aditiva, pero un consumidor que hace exhaustivo sobre los valores conocidos falla al recibir uno nuevo. Por eso los contratos maduros documentan desde el principio que los enumerados pueden crecer y exigen un caso por defecto.

Cuando el cambio es incompatible, versionar. Y la regla que importa no es dónde poner la versión —ruta, cabecera o tipo de medio— sino cuánto tiempo mantienes la anterior viva y cómo avisas: una versión sin fecha de retirada publicada no es una versión, es deuda.

El campo Deprecation y Sunset (RFC 8594) permiten anunciarlo en la propia respuesta, para que el cliente se entere antes de romperse.

Ejemplo trabajado

Durante una promoción, 1 de cada 400 clientes de CloudShop aparece cobrado dos veces. El importe duplicado siempre coincide, y los dos cargos distan menos de 30 segundos.

La hipótesis es reintento sobre POST no idempotente. Se confirma en los logs:

```
$ jq -r 'select(.ruta=="/pagos") | [.trace, .pedido, .estado, .ms] | @tsv' pagos.jsonl | sort | head -
4
4f2a9c  A-1042  timeout  30021
7d2b8e  A-1042  201      412
```

Dos trazas distintas para el mismo pedido: la primera agotó el plazo del cliente a los 30 s, pero el servidor la completó —el cargo existe— y el cliente reintentó con una petición nueva.

La aritmética del incidente:

```
peticiones de pago en la promoción      48.000
timeout del cliente                       30 s
p99,8 de latencia del proveedor           31 s
peticiones que superan el timeout    ≈ 0,25 %  → 120
de ellas, completadas en el servidor ≈ 100 %  → 120 cobros dobles
observado                                       118
```

Cuadra: el problema no es la tasa de error del proveedor, es que el timeout del cliente es menor que la cola de latencia del servidor. Subir el timeout reduce la frecuencia pero no elimina la clase de fallo: siempre habrá una petición que se corte después de tener efecto.

Corrección con clave de idempotencia, generada por el cliente una vez por intención y reutilizada en los reintentos:

```
clave = str(uuid.uuid4())            # fuera del bucle: la MISMA en cada reintento
for intento in range(3):
    try:
        r = sesion.post("/pagos", json=cuerpo,
                        headers={"Idempotency-Key": clave}, timeout=35)
        break
    except (TimeoutError, ConnectionError):
        time.sleep(2 ** intento + random.uniform(0, 1))
```

Y en el servidor, con la clave persistida en la misma transacción que el cargo:

```
BEGIN;
  INSERT INTO idempotencia (clave, estado) VALUES ($1, 'en_curso')
    ON CONFLICT (clave) DO NOTHING;
  -- si no insertó ninguna fila, es un reintento: devolver lo guardado
  INSERT INTO cargos (pedido, importe) VALUES ($2, $3);
  UPDATE idempotencia SET estado='hecho', respuesta=$4 WHERE clave=$1;
COMMIT;
```

Resultado tras el despliegue: 0 cobros dobles en 52.000 pagos de la promoción siguiente, con 96 reintentos que devolvieron el resultado original en vez de repetir el cargo.

El detalle que decide todo es dónde se genera la clave: fuera del bucle de reintento. Generarla dentro produce una clave distinta por intento y deja el sistema exactamente igual de roto, con la ilusión de estar protegido.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/009-apis-rest-autenticacion-y-contratos/lab.py
```

El laboratorio selecciona el motor de práctica api 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 cliente-api-verificado 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 contrato versionado con pruebas positivas y negativas. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

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

■ Criterio de aceptación

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

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
Se ejecutan acciones que nadie solicitó	Una operación con efecto se expuso como GET y la repitió un precargador o un proxy	GET debe ser seguro; usa POST, PUT o DELETE para cualquier cosa con efecto.
Un reintento tras timeout duplica un cobro	POST no es idempotente y no había clave de idempotencia	Genera la clave una vez por intención, fuera del bucle, y persístela en la misma transacción que el efecto.
Un usuario autenticado accede a datos de otro cambiando el identificador de la URL	Se autenticó pero no se autorizó por objeto (BOLA)	Comprueba la propiedad del recurso en cada petición y responde 404, no 403.
Un recorrido paginado repite o pierde elementos	Paginación por desplazamiento sobre una colección que cambia	Usa cursor opaco sobre una clave estable con desempate único.
Añadir un valor a un enumerado rompe consumidores en producción	Se asumió que todo cambio aditivo es compatible	Documenta desde el principio que los enumerados crecen y exige un caso por defecto en el cliente.

■ Seguridad, ética y costo

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

■ Preguntas de comprobación

1. ¿Por qué PUT y DELETE se pueden reintentar sin riesgo y POST no?
2. ¿En qué punto exacto debe persistirse la clave de idempotencia para que no quede una ventana de doble efecto?
3. Un usuario autenticado pide un recurso ajeno. ¿Por qué 404 es mejor respuesta que 403?
4. Explica con un ejemplo cómo la paginación por desplazamiento pierde un elemento cuando la colección cambia.
5. ¿Cuáles de estos cambios rompen a un consumidor: añadir un campo opcional, añadir un valor a un enumerado, hacer obligatorio un campo opcional?

Referencias

• Fielding, R. y Reschke, J., eds. (2022). RFC 9110: HTTP Semantics, secs. 9.2.1-9.2.2 — métodos seguros e idempotentes. <https://www.rfc-editor.org/rfc/rfc9110#section-9.2>

• Jena, J. et al. (2024). The Idempotency-Key HTTP Header Field, IETF draft — semántica y manejo de reintentos. <https://datatracker.ietf.org/doc/draft-ietf-httpapi-idempotency-key-header/>

• OWASP (2023). API Security Top 10, API1:2023 Broken Object Level Authorization. <https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/>

• Wilde, E. (2019). RFC 8594: The Sunset HTTP Header Field — anunciar la retirada de un contrato. <https://www.rfc-editor.org/rfc/rfc8594>

• Nottingham, M. (2022). RFC 9205: Building Protocols with HTTP — buenas prácticas de diseño sobre HTTP. <https://www.rfc-editor.org/rfc/rfc9205>

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

Descargar: Parte 00 en PDF · Manual integral

Anterior	Índice	Siguiente
← 008 · Virtualización, hipervisores e imágenes	Parte 00 · Programa	010 · Responsabilidad compartida y pensamiento de riesgo →

Evaluación — 009 APIs REST, autenticación y contratos

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 cliente-api-verificado 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.

010 — Responsabilidad compartida y pensamiento de riesgo

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 4 Laboratorio: security · Estado: EXECUTABLECORE

Propósito

Trazar con precisión dónde termina la responsabilidad del proveedor y empieza la tuya, y convertir esa línea en un método de análisis de riesgo aplicable antes de escribir infraestructura. Es la clase que explica por qué casi ningún incidente público «de la nube» fue del proveedor, y la que fija el criterio con el que se juzgarán todas las decisiones de las 278 clases restantes.

Resultados de aprendizaje

Al finalizar podrás:

1. Situar cualquier control concreto del lado correcto de la línea según el modelo de servicio contratado.
2. Distinguir disponibilidad del servicio, durabilidad del dato y confidencialidad, y qué garantiza el proveedor de cada una.
3. Leer un SLA identificando qué mide, qué excluye y cuál es la compensación real.
4. Aplicar un método de análisis de riesgo que produzca controles verificables en vez de una lista de miedos.
5. Calcular el riesgo residual de una decisión y declararlo explícitamente en vez de dejarlo implícito.

Conceptos centrales

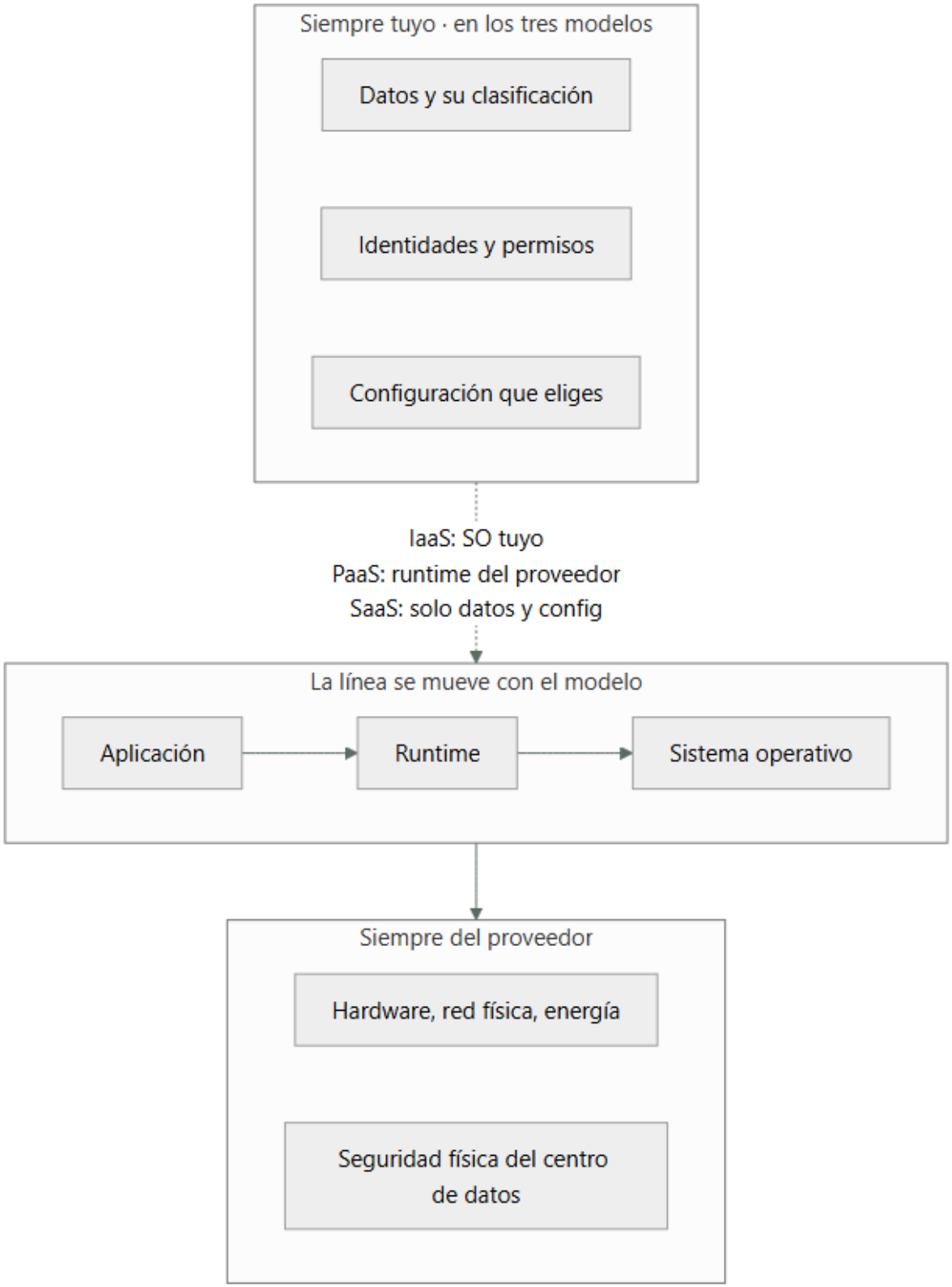
Concepto	Comprensión verificable
responsabilidad compartida	Reparto contractual de controles entre proveedor y cliente. El proveedor responde de la seguridad de la nube; el cliente, de la seguridad en la nube. La línea se mueve con el modelo de servicio, nunca desaparece.
SLA	Compromiso contractual con una métrica, un umbral y una compensación. Casi siempre mide disponibilidad del plano de servicio y excluye lo que más duele: tus datos, tu configuración y tu lucro cesante.
durabilidad	Probabilidad de que un objeto almacenado siga existiendo tras un periodo. Es independiente de la disponibilidad: un dato puede ser durable y estar inaccesible, o estar disponible y haber sido borrado por ti.
riesgo residual	Lo que queda después de aplicar los controles. No es un fallo del análisis: es el resultado esperado. Un diseño honesto lo nombra y dice quién lo acepta.
radio de impacto	Alcance de lo que se rompe cuando algo falla. Reducirlo —por celdas, cuentas o regiones— suele ser más barato y más eficaz que intentar evitar el fallo.

Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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

■ Flujo de razonamiento



Desarrollo

1. Lo que nunca cambia de lado

El modelo de servicio mueve la línea, pero tres cosas permanecen siempre del lado del cliente, en IaaS, en PaaS y en SaaS:

1. Los datos: qué guardas, cómo los clasificas y cuánto tiempo los conservas.

2. Las identidades y sus permisos: quién puede hacer qué.

3. La configuración que eliges: si un bucket es público, lo hiciste público tú.

Esto explica un hecho incómodo: la inmensa mayoría de las brechas atribuidas públicamente a «la nube» fueron configuraciones del cliente, no fallos del proveedor. Almacenamiento expuesto sin autenticación, claves de acceso subidas a un repositorio público, permisos comodín concedidos «temporalmente» hace dos años.

Gartner lleva años sosteniendo la misma proyección: hasta 2025, en torno al 99 % de los fallos de seguridad en la nube serán culpa del cliente. El dato importa menos que su consecuencia de diseño: el trabajo de seguridad no está en elegir proveedor, está en el lado que te toca.

El error simétrico también existe: asumir que nada hace el proveedor. El cifrado del disco físico, la destrucción segura de medios, el control de acceso al centro de datos y el parcheo del hipervisor no son tuyos y no deberías replicarlos.

2. Disponibilidad, durabilidad y confidencialidad son tres cosas distintas

Se confunden a diario y cada una tiene garantías, riesgos y controles diferentes:

Propiedad	Pregunta	Garantía típica	Qué la rompe
Disponibilidad	¿Puedo acceder ahora?	99,9 %-99,99 % en SLA	Caída de zona, saturación, error de red
Durabilidad	¿Seguirá existiendo?	99,999999999 % anual (11 nueves)	Un borrado tuyo, no un fallo de disco
Confidencialidad	¿Solo lo ve quien debe?	No hay SLA	Configuración, permisos, fuga de credenciales

Los once nueves de durabilidad de un almacenamiento de objetos significan que la probabilidad anual de perder un objeto dado por fallo de infraestructura es 10^{-11}. Con 10 millones de objetos:

```
pérdida esperada = 10.000.000 × 10^-11 = 0,0001 objetos al año
                 ≈ 1 objeto cada 10.000 años
```

Y sin embargo se pierden datos constantemente. La razón es que la durabilidad no protege del borrado autorizado: un DELETE con credenciales válidas, un ciclo de vida mal configurado o un ransomware con tus permisos destruyen el objeto sin que la infraestructura falle en absoluto.

De ahí que la copia de seguridad siga siendo tuya, con dos propiedades que la durabilidad no da: versionado —para volver atrás— e inmutabilidad —para que ni tus propias credenciales puedan borrarla—. Se retoma en la parte 21.

3. Leer un SLA por lo que excluye

Un SLA tiene tres partes y la tercera casi nunca es la que se recuerda:

1. Métrica y umbral: por ejemplo, 99,99 % mensual de disponibilidad del servicio.

2. Compensación: un porcentaje de crédito sobre lo facturado de ese servicio.

3. Exclusiones: lo que no cuenta como indisponibilidad.

La aritmética del umbral, en un mes de 30 días:

```
99,9 %   → 43,2 min de caída permitida al mes
99,95 %  → 21,6 min
99,99 %  →  4,3 min
99,999 % →   26 s
```

El salto de 99,9 a 99,99 no es «un poco mejor»: exige reducir la ventana de fallo diez veces, lo que en la práctica obliga a redundancia entre zonas y a automatizar la conmutación, porque ninguna intervención humana cabe en 4,3 minutos.

Las exclusiones habituales, que conviene buscar antes de firmar:

- Indisponibilidad por tu configuración o tu código.
- Ventanas de mantenimiento anunciadas.
- Fuerza mayor y fallos de tu proveedor de red.
- Servicios en vista previa o beta.

Y la exclusión que más importa: la compensación es un crédito de servicio, no una indemnización. Si una caída de 4 horas te cuesta 80.000 USD en ventas, el SLA te devuelve un porcentaje de lo que pagaste por ese servicio ese mes —quizá 200 USD—. El SLA no transfiere tu riesgo de negocio: acota el del proveedor. Tu continuidad la pagas tú, con arquitectura.

4. Un método de riesgo que produce controles, no miedos

Una lista de amenazas sin estructura no sirve para decidir. Cinco preguntas, en orden, sí:

1. ¿Qué protegemos? El activo, con su clasificación. «La base de datos» no es un activo; «los datos personales de clientes, categoría alta» sí.
2. ¿De quién y de qué? Actor y capacidad: un atacante externo sin credenciales, un empleado con acceso legítimo, un error de operación, un fallo de infraestructura.
3. ¿Por dónde? El vector concreto: credencial filtrada, permiso excesivo, dependencia comprometida, error de configuración.
4. ¿Qué control lo corta? Preventivo, detectivo o correctivo, y verificable.
5. ¿Qué queda? El riesgo residual, nombrado y aceptado por alguien con autoridad para aceptarlo.

La columna que separa un análisis útil de un documento decorativo es la cuarta: cada control debe tener una prueba. «Cifrado en tránsito» no es un control; «TLS 1.2 mínimo, verificado por una prueba automática que falla el despliegue si se acepta TLS 1.0» sí lo es.

Y los tres tipos no son intercambiables. Un control preventivo que falle en silencio es peor que uno detectivo que avise: el primero da confianza falsa. Por eso todo control preventivo necesita una prueba negativa —comprobar que efectivamente deniega— que es exactamente lo que hacen los lab.py de este programa.

5. Radio de impacto: contener sale más barato que evitar

Los fallos ocurren. La pregunta de diseño no es cómo evitarlos todos, sino qué se rompe cuando ocurren.

```
Una cuenta, todo junto:        1 credencial comprometida → todo el negocio
Cuentas separadas por entorno: 1 credencial comprometida → un entorno
Cuentas por dominio + entorno: 1 credencial comprometida → un dominio de un entorno
```

La separación no impide la brecha; acota su alcance, y eso suele ser mucho más barato que intentar hacer la brecha imposible. Es el mismo razonamiento de la clase 007 sobre capacidades: no «qué necesita para funcionar», sino «qué obtiene quien lo comprometa».

Tres preguntas que hay que poder responder de cualquier diseño:

- ¿Qué se rompe si esto falla? Si la respuesta es «todo», el diseño tiene un punto único de fallo aunque el componente sea redundante.
- ¿Cuánto tarda en detectarse? Un fallo silencioso de días es peor que una caída ruidosa de minutos.
- ¿Se puede revertir? Un cambio irreversible exige un nivel de control distinto al de uno reversible.

Estas tres preguntas son el guion de los ADR de la parte 12 y de los game days de la parte 21.

Ejemplo trabajado

Un informe de exposición avisa a CloudShop de que un bucket con facturas de clientes es accesible sin autenticación. El equipo quiere responder «el proveedor garantiza cifrado y once nueves». Se aplica el método.

1. ¿Qué protegemos? 412.000 facturas en PDF con nombre, RUT, dirección y detalle de compra. Clasificación: datos personales, categoría alta. Retención legal: 6 años.

2. ¿De quién? Actor externo sin credenciales. No hace falta más: el objeto es legible por cualquiera con la URL.

3. ¿Por dónde? Se reconstruye el vector:

```
$ aws s3api get-bucket-policy --bucket cloudshop-facturas | jq -r '.Policy' | jq '.Statement[0]'
{"Effect": "Allow", "Principal": "*", "Action": "s3:GetObject", ...}
$ aws cloudtrail lookup-events --lookup-attributes \
    AttributeKey=EventName,AttributeValue=PutBucketPolicy --max-items 1 \
```

```
    | jq -r '.Events[0] | "\(.EventTime) \(.Username)"'
2024-11-14T16:22:08Z  ci-deploy-role
```

La política la puso un despliegue hace 21 meses. El cifrado en reposo estaba activo todo el tiempo y no sirvió de nada: cifra frente a quien acceda al disco físico, no frente a quien hace una petición autorizada por la política. La durabilidad de once nueves tampoco: garantiza que el objeto no se pierda, no que no se lea.

Cuantificación de la exposición:

```
objetos legibles                     412.000
ventana               21 meses ≈ 638 días
peticiones GET anónimas en los logs   1.847  (de 12 IP distintas)
```

4. ¿Qué controles, y cómo se verifican?

Tipo	Control	Prueba
Preventivo	Bloqueo de acceso público a nivel de cuenta	Prueba negativa: intentar publicar y comprobar que falla
Preventivo	Política de organización que prohíbe Principal: ""	Despliegue de prueba que debe ser rechazado
Detectivo	Alerta sobre cualquier PutBucketPolicy	Ejecutarla y verificar que notifica en < 5 min
Correctivo	Versionado + retención inmutable	Restaurar un objeto borrado en un simulacro

Cada fila tiene una prueba ejecutable. La segunda columna sin la tercera es una intención, no un control.

5. ¿Qué riesgo queda? Los datos estuvieron expuestos 638 días y eso no se puede revertir: hay que asumir que fueron copiados y actuar en consecuencia —notificación a los afectados y a la autoridad, según la normativa aplicable—. Además, un operador con permiso legítimo sigue pudiendo exportarlos; ese riesgo se acota con registro de acceso y revisión periódica, no se elimina.

La conclusión que importa: ni el cifrado ni la durabilidad protegían de esto, porque ninguno de los dos estaba del lado del problema. El control que faltaba era de configuración, y la configuración es siempre del cliente en los tres modelos de servicio.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/010-responsabilidad-compartida-y-pensamiento-de-riesgo/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 matriz-de-responsabilidad 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 matriz-de-responsabilidad para el caso CloudShop. Incluye una alternativa descartada, un supuesto que pueda falsarse, una prueba de fallo y una decisión de rollback.

■ Criterio de aceptación

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

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
Se asume que el cifrado del proveedor protege los datos de un acceso indebido	El cifrado en reposo protege frente al disco físico, no frente a una petición autorizada	Separa los controles por propiedad: confidencialidad se protege con permisos y política, no con cifrado en reposo.
Se confía en los once nueves de durabilidad como estrategia de respaldo	La durabilidad no protege del borrado autorizado, que es la causa real de pérdida	Añade versionado y retención inmutable; son tuyos, no del proveedor.
Tras una caída larga, el SLA compensa una fracción mínima de la pérdida	La compensación es un crédito sobre lo facturado de ese servicio, no una indemnización	Dimensiona la continuidad con arquitectura; el SLA acota el riesgo del proveedor, no el tuyo.
El análisis de riesgo es una lista de amenazas que nadie usa para decidir	No llegó a controles verificables ni nombró el riesgo residual	Exige a cada control una prueba ejecutable y un responsable que acepte lo que queda.
Una credencial comprometida da acceso a todo el negocio	No hay separación de cuentas ni entornos: el radio de impacto es total	Separa por entorno y dominio; contener el alcance es más barato que evitar la brecha.

■ 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 responsabilidades siguen siendo del cliente en IaaS, PaaS y SaaS por igual?
2. Un almacenamiento con once nueves de durabilidad pierde datos de un cliente. ¿Cómo es posible sin que falle la infraestructura?
3. ¿Cuántos minutos de caída mensual permite un SLA de 99,99 %, y por qué eso descarta la intervención humana?
4. Un control dice «cifrado en tránsito». ¿Qué le falta para ser verificable?
5. ¿Por qué un control preventivo que falla en silencio es peor que uno detectivo?

Referencias

- AWS (2024). Shared Responsibility Model — reparto de controles por modelo de servicio. <https://aws.amazon.com/compliance/shared-responsibility-model/>
- Microsoft (2024). Shared responsibility in the cloud — tabla comparada IaaS/PaaS/SaaS. <https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility>
- Cloud Security Alliance (2024). Top Threats to Cloud Computing — taxonomía de amenazas con incidentes documentados. <https://cloudsecurityalliance.org/research/topics/top-threats>
- Shostack, A. (2014). Threat Modeling: Designing for Security — las cuatro preguntas y el uso de STRIDE.

• NIST (2018). Framework for Improving Critical Infrastructure Cybersecurity v1.1 — funciones identificar, proteger, detectar, responder y recuperar. <https://doi.org/10.6028/NIST.CSWP.04162018>

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

Descargar: Parte 00 en PDF · Manual integral

Anterior	Índice	Siguiente
← 009 · APIs REST, autenticación y contratos	Parte 00 · Programa	011 · Costo, energía, capacidad y medición básica →

Evaluación — 010 Responsabilidad compartida y pensamiento de riesgo

Preguntas

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

Reto verificable

Entrega matriz-de-responsabilidad 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.

011 — Costo, energía, capacidad y medición básica

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 4 Laboratorio: finops · Estado: EXECUTABLECORE

Propósito

Convertir el coste en una magnitud de ingeniería con la que se decide, no en una factura que se revisa al final del mes. Aquí se establece la unidad económica que el programa usará en las 277 clases restantes: coste por transacción útil. Sin ella, «optimizar costes» es apagar cosas al azar y esperar que nada se rompa.

Resultados de aprendizaje

Al finalizar podrás:

1. Definir una unidad de coste ligada a algo que el negocio entiende, y calcularla a partir de la factura.
2. Aplicar la ley de Little para dimensionar capacidad a partir de tasa de llegada y tiempo de servicio.
3. Explicar por qué la latencia se dispara antes de llegar al 100 % de utilización, y a partir de qué punto.
4. Comparar modelos de precio —bajo demanda, compromiso, capacidad excedente— con un umbral de utilización calculado.
5. Estimar el coste energético de una carga y situarlo frente a su coste monetario.

Conceptos centrales

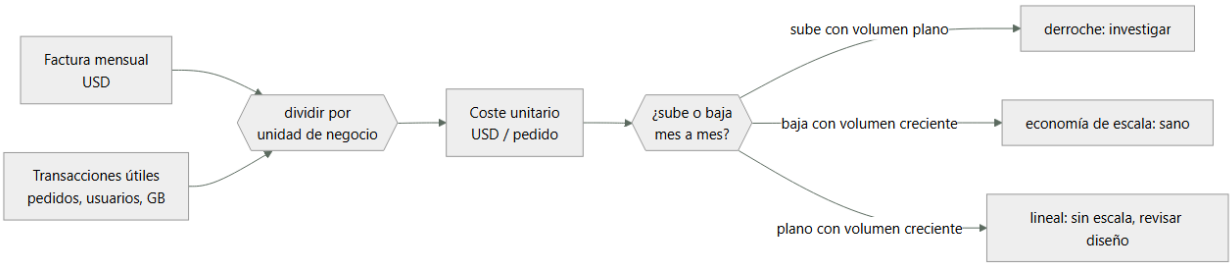
Concepto	Comprensión verificable
unidad de coste	Denominador que convierte la factura en una métrica decidible: coste por pedido, por usuario activo, por GB procesado. Sin denominador, un aumento de factura no distingue crecimiento de derroche.
utilización	Fracción del tiempo que un recurso está ocupado, entre 0 y 1. Por encima de 0,7 el tiempo de espera crece de forma no lineal, así que perseguir el 100 % es perseguir una cola infinita.
ley de Little	En un sistema estable, L = λ·W: el número medio de elementos en el sistema es la tasa de llegada por el tiempo medio de permanencia. No supone ninguna distribución, lo que la hace aplicable casi siempre.
coste amortizado	Coste efectivo de un compromiso repartido en su plazo. Permite comparar un descuento por reserva de 1 o 3 años con el precio bajo demanda sobre la misma base.
PUE	Power Usage Effectiveness: energía total del centro de datos dividida por la que llega al equipamiento informático. 1,0 sería perfecto; los hiperescalares operan cerca de 1,1 y una sala tradicional ronda 1,8.

Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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. Sin denominador, la factura no dice nada

«La factura subió un 40 %» es una observación inútil hasta que se divide por algo. Con denominador, la misma cifra puede significar tres cosas opuestas:

```
mes 1:  12.400 USD /  620.000 pedidos = 0,0200 USD/pedido
mes 2:  17.360 USD /  980.000 pedidos = 0,0177 USD/pedido   ← +40 % factura, −11 % unitario
mes 3:  17.360 USD /  610.000 pedidos = 0,0285 USD/pedido   ← +43 % unitario: hay derroche
```

El mes 2 es crecimiento sano: se factura más porque se vende más, y cada pedido cuesta menos. El mes 3 con la misma factura es una alarma: el volumen cayó y el coste no.

La unidad debe cumplir tres condiciones para servir:

1. La entiende el negocio: pedidos, usuarios activos, GB procesados. No «horas de instancia».
2. Se puede atribuir: existe un mecanismo de etiquetado que asigna cada recurso a un producto o equipo.
3. Se mide igual todos los meses: cambiar el denominador rompe la serie histórica.

La condición 2 es la que exige trabajo previo: sin etiquetado consistente, el 30-40 % de la factura acaba en un cubo de «sin asignar» que nadie reclama y nadie optimiza. Por eso el etiquetado es la primera tarea de FinOps, antes que cualquier optimización.

2. La ley de Little: dimensionar sin adivinar

Para cualquier sistema estable, con independencia de la distribución de llegadas o servicios:

```
L = λ · W

L = elementos en el sistema (concurrencia)
λ = tasa de llegada (peticiones por segundo)
W = tiempo medio en el sistema (segundos)
```

Sirve para dimensionar antes de desplegar. Con 800 peticiones por segundo y 250 ms de latencia media:

```
L = 800 × 0,250 = 200 peticiones concurrentes
```

Si cada trabajador atiende una petición a la vez, hacen falta 200 trabajadores solo para el estado estacionario. Con 40 trabajadores por instancia:

```
instancias = 200 / 40 = 5   (mínimo teórico, utilización 1,0)
```

Y ese «mínimo teórico» es exactamente el número que no hay que desplegar, por lo que viene a continuación.

La ley también funciona al revés, para detectar incoherencias: si mides 200 conexiones concurrentes y 800 peticiones por segundo, la latencia media tiene que ser 250 ms. Si tu panel dice 80 ms, alguna de las tres medidas está mal —normalmente porque se está midiendo la media de una distribución con cola larga—.

3. Por qué el 100 % de utilización es una trampa

En un sistema con llegadas aleatorias, el tiempo de espera no crece de forma lineal con la utilización. Para una cola M/M/1:

```
W = W_servicio / (1 − ρ)          ρ = utilización
```

Con un servicio de 100 ms:

ρ	Tiempo total	Multiplicador
0,50	200 ms	2×
0,70	333 ms	3,3×
0,80	500 ms	5×
0,90	1.000 ms	10×
0,95	2.000 ms	20×
0,99	10.000 ms	100×

Entre 0,70 y 0,90 la utilización sube 20 puntos y la latencia se triplica. Ese es el motivo de que el objetivo operativo habitual sea 0,60-0,70 y no 0,95: el 30 % de capacidad «ociosa» no es derroche, es lo que absorbe la varianza de las llegadas.

Aplicado al cálculo anterior:

```
mínimo teórico (ρ = 1,0)          5 instancias
objetivo ρ = 0,70                 5 / 0,70 = 7,1 → 8 instancias
tolerancia a la caída de una zona (n+1)      → 9 instancias
```

Las 4 instancias adicionales sobre el mínimo teórico no son exceso: son el precio de un percentil 95 estable y de sobrevivir a la pérdida de una zona. Optimizar costes recortando ahí es cambiar dinero por incidentes, y el intercambio hay que hacerlo explícito.

4. Modelos de precio: el umbral que decide

Los tres modelos comunes, con el mismo recurso como referencia:

Modelo	Precio relativo	Compromiso	Riesgo
Bajo demanda	1,00	Ninguno	Ninguno
Compromiso 1 año	0,60	Pagas uses o no	Sobredimensionar
Compromiso 3 años	0,40	Ídem, más largo	Cambio tecnológico
Capacidad excedente	0,10-0,30	Ninguno	Interrupción con preaviso corto

El umbral de utilización a partir del cual compensa comprometerse sale de igualar ambos costes:

```
coste bajo demanda = u × 730 h × P
coste comprometido = 730 h × 0,60 P

umbral: u × P = 0,60 P  →  u = 0,60
```

Con más del 60 % de uso sostenido, el compromiso a un año sale a cuenta. Por debajo, no. Ese único número evita la mayoría de las discusiones sobre reservas.

La capacidad excedente merece su propio criterio: cuesta entre un 70 % y un 90 % menos, pero puede retirarse con un preaviso de dos minutos. Es correcta para trabajos por lotes reanudables, colas asíncronas y entornos de pruebas. Nunca para un componente cuya caída rompa una petición de usuario, salvo que la arquitectura absorba la pérdida sin degradación visible.

5. La energía es el coste que no aparece en la factura

El coste monetario ya incorpora la energía, pero la huella no, y cada vez más organizaciones tienen que declararla.

```
energía_total = potencia_TI × horas × PUE
```

Para 9 instancias de 80 W durante un mes:

```
potencia TI     = 9 × 80 W = 720 W = 0,72 kW
energía TI      = 0,72 kW × 730 h = 525,6 kWh
con PUE 1,12    = 525,6 × 1,12 = 588,7 kWh/mes
```

Y su traducción a emisiones depende por completo de dónde se ejecuta:

```
región con 50 g CO■e/kWh:  588,7 × 0,050 =  29,4 kg CO■e/mes
región con 400 g CO■e/kWh: 588,7 × 0,400 = 235,5 kg CO■e/mes
```

Un factor de 8 por la misma carga, con la misma factura. La elección de región, que en la clase 001 se justificaba por latencia y en la 010 por soberanía del dato, tiene aquí un tercer eje.

Dos matices honestos: los factores de emisión varían por hora según el mix de generación, y el PUE no incluye el coste de fabricación del hardware. Cualquier cifra de este tipo es una estimación con orden de magnitud útil, no una medición.

Ejemplo trabajado

La factura de CloudShop pasa de 12.400 a 17.360 USD en un mes y dirección pide «bajar costes un 30 %». El equipo empieza calculando el denominador antes de tocar nada.

```
mes anterior  12.400 USD / 620.000 pedidos = 0,0200 USD/pedido
mes actual    17.360 USD / 980.000 pedidos = 0,0177 USD/pedido
```

El coste unitario bajó un 11 %. El aumento es crecimiento, no derroche, y ese dato cambia la conversación: recortar un 30 % lineal significaría recortar capacidad mientras el volumen sube un 58 %.

Se desglosa por unidad para encontrar dónde sí hay margen:

```
cómputo (9 instancias)        6.480     0,0066        37 %
base de datos                 4.900     0,0050        28 %
transferencia de salida       3.100     0,0032        18 %
almacenamiento                1.480     0,0015         9 %
observabilidad                1.400     0,0014         8 %
```

Se revisan las dos partidas mayores con los criterios de la clase.

Cómputo. Utilización medida sobre 30 días: media 0,58, percentil 95 diario 0,71. Por la ley de Little con λ=800/s y W=250 ms hacen falta 200 concurrentes; a ρ=0,70 son 8 instancias y con tolerancia n+1, nueve. Están correctamente dimensionadas: no hay recorte sin degradar. Pero la utilización sostenida del 58 % está cerca del umbral de compromiso:

```
umbral de compromiso a 1 año: u > 0,60
medido: 0,58 en media, 0,71 en p95
→ comprometer 6 de las 9 instancias (base estable), dejar 3 bajo demanda
ahorro: 6 × 720 USD × 0,40 = 1.728 USD/mes    (−27 % del cómputo)
```

Transferencia de salida. 3.100 USD equivalen a unos 34 TB. Se revisa la tasa de acierto del CDN:

```
aciertos de caché      61 %
salida desde origen    34 TB
```

Subir el acierto al 90 % ajustando Cache-Control en los activos con huella —lo de la clase 006— reduce la salida desde origen:

```
salida estimada  34 TB × (1 − 0,90)/(1 − 0,61) = 8,7 TB
ahorro                          ≈ 2.300 USD/mes    (−74 % de esa partida)
```

Resultado combinado:

```
ahorro total            1.728 + 2.300 = 4.028 USD/mes  (−23 %)
coste unitario nuevo    13.332 / 980.000 = 0,0136 USD/pedido   (−32 %)
```

No se llegó al 30 % de la factura, pero sí se superó el 30 % en coste unitario, que es la métrica que el negocio quería sin saber nombrarla. Y ningún ahorro salió de recortar capacidad: salieron de comprometer la base estable y de dejar de pagar transferencia evitable.

Riesgo residual declarado: el compromiso a un año sobre 6 instancias deja de ser rentable si el volumen cae más de un 40 %. Se acepta con revisión trimestral.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/011-costo-energia-capacidad-y-medicion-basica/lab.py
```

El laboratorio selecciona el motor de práctica finops 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 estimacion-de-costo 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 cálculo trazable con unidad, supuesto y sensibilidad. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

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

■ Criterio de aceptación

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

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
Se pide recortar un porcentaje de la factura sin más contexto	No hay unidad de coste, así que no se distingue crecimiento de derroche	Divide la factura por una unidad de negocio antes de decidir; el unitario puede bajar mientras la factura sube.
La latencia se dispara al subir la utilización del 70 % al 90 %	El tiempo de espera crece como 1/(1–ρ), no linealmente	Fija el objetivo de utilización en 0,60-0,70; la capacidad libre absorbe la varianza.
Un compromiso de 3 años se convierte en gasto muerto	Se comprometió capacidad con utilización por debajo del umbral	Compromete solo la base estable, con u > 0,60; deja el pico bajo demanda.
Una carga en capacidad excedente rompe peticiones de usuario al ser retirada	Se usó capacidad interrumpible para un componente del camino crítico	Resérvala para trabajos reanudables; el preaviso puede ser de dos minutos.
El 35 % de la factura está en «sin asignar» y nadie lo optimiza	Falta etiquetado consistente, así que no hay atribución posible	Impón etiquetas obligatorias en el aprovisionamiento antes de intentar cualquier optimización.

■ 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. La factura sube un 40 % y el coste unitario baja un 11 %. ¿Hay un problema de costes? Justifica.
2. Con 500 peticiones por segundo y 400 ms de latencia media, ¿cuántas peticiones concurrentes hay en el sistema?

3. ¿Por qué desplegar al 95 % de utilización sale más caro que al 70 %, aunque use menos instancias?
4. ¿A partir de qué utilización sostenida compensa un compromiso de un año al 60 % del precio, y de dónde sale ese umbral?
5. Dos regiones tienen el mismo precio. ¿Qué otro criterio, con qué orden de magnitud, puede decidir entre ellas?

Referencias

- Little, J. D. C. (1961). A Proof for the Queuing Formula: L = λW. Operations Research 9(3), 383-387. <https://doi.org/10.1287/opre.9.3.383>
- Storment, J. R. y Fuller, M. (2023). Cloud FinOps, 2.ª ed., caps. 6-9 — unidad de coste, etiquetado y modelos de compromiso.
- Gunther, N. (2007). Guerrilla Capacity Planning — leyes de escalabilidad y el efecto de la utilización sobre la latencia.
- Barroso, L., Hölzle, U. y Ranganathan, P. (2018). The Datacenter as a Computer, 3.ª ed., cap. 6 — PUE, eficiencia y coste total. <https://doi.org/10.2200/S00874ED3V01Y201809CAC046>
- FinOps Foundation (2024). FinOps Framework — capacidades de asignación, previsión y optimización de tasa. <https://www.finops.org/framework/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 00 en PDF · Manual integral

Anterior	Índice	Siguiente
← 010 · Responsabilidad compartida y pensamiento de riesgo	Parte 00 · Programa	012 · Proyecto: servicio local reproducible y observable →

Evaluación — 011 Costo, energía, capacidad y medición básica

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 estimacion-de-costo 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.

012 — Proyecto: servicio local reproducible y observable

Parte: 00 — Fundamentos de computación, redes y Linux Nivel: inicial · Horas estimadas: 8 Laboratorio: capstone · Estado: EXECUTABLECORE

Propósito

Integrar las once clases anteriores en un artefacto único: un servicio local que arranca con un comando, se observa, falla de forma controlada y se recupera. Es el punto de partida de CloudShop, el sistema que evolucionará durante las 276 clases restantes hasta operar en tres nubes. Lo que se construye aquí no es un ejercicio: es la línea base contra la que se medirá cada cambio posterior.

Resultados de aprendizaje

Al finalizar podrás:

1. Ensamblar un servicio con comprobaciones de vida y de disponibilidad distinguiendo correctamente sus semánticas.
2. Instrumentar las cuatro señales doradas y justificar por qué la media es insuficiente para la latencia.
3. Provocar un fallo de dependencia y demostrar que el servicio degrada en vez de caer.
4. Establecer una línea base de rendimiento reproducible que sirva de referencia para comparar cambios futuros.
5. Documentar el servicio de modo que otra persona lo levante, lo observe y lo recupere sin conocimiento tácito.

Conceptos centrales

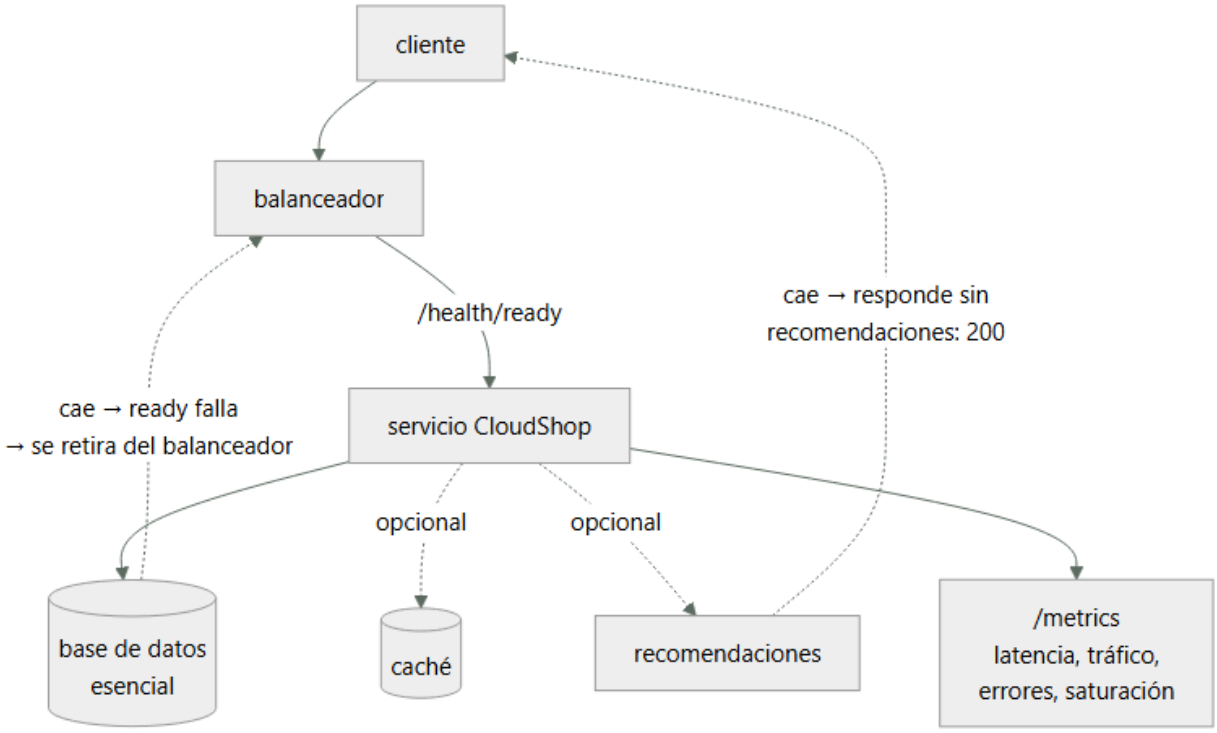
Concepto	Comprensión verificable
liveness	Comprobación de si el proceso debe reiniciarse. Debe fallar solo ante un estado irrecuperable: si comprueba dependencias externas, una caída de la base de datos reinicia todas las réplicas y convierte una degradación en una caída total.
readiness	Comprobación de si el proceso puede recibir tráfico ahora. Sí debe mirar dependencias: al fallar, el balanceador retira la instancia sin matarla, y vuelve a incluirla cuando se recupera.
señales doradas	Latencia, tráfico, errores y saturación. Cuatro métricas que, según el SRE Book, bastan para detectar la mayoría de los problemas de un servicio orientado a peticiones.
degradación elegante	Responder con funcionalidad reducida en vez de fallar por completo cuando una dependencia no esencial no está disponible.
línea base	Medición reproducible del comportamiento actual. Sin ella, cualquier afirmación posterior sobre mejora o regresión es una opinión.

Modelo mental

Antes de alquilar infraestructura remota, aprende a observar una computadora local como un conjunto de procesos, redes, archivos, identidades y recursos medibles.

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. Liveness y readiness responden a preguntas opuestas

Confundirlas es el error que convierte una degradación en una caída total, y ocurre porque ambas «comprueban si el servicio está bien».

	Liveness	Readiness
Pregunta	¿Hay que reiniciar?	¿Puede recibir tráfico?
Acción al fallar	Matar y reiniciar	Retirar del balanceador
¿Mira dependencias?	Nunca	Sí
Coste de un falso positivo	Reinicio innecesario	Capacidad reducida

```
@app.get("/health/live")
def live():
    return {"status": "ok"}          # solo: ¿responde el bucle de eventos?

@app.get("/health/ready")
def ready():
    fallos = []
    if not db.ping(timeout=1.0):
        fallos.append("db")           # esencial
    if fallos:
        raise HTTPException(503, {"no_listo": fallos})
    return {"status": "ok", "degradado": estado_opcionales()}
```

El escenario que justifica la regla: la base de datos se cae 40 segundos. Con liveness mirando la base de datos, las 9 réplicas fallan la comprobación a la vez y el orquestador las mata todas; cuando la base vuelve, no hay servicio porque todas están arrancando en frío. Con la separación correcta, las 9 salen del balanceador, siguen vivas, y vuelven a entrar en cuanto la dependencia responde.

Hay una tercera comprobación que conviene desde el principio: startup, para procesos con arranque lento. Sin ella hay que relajar el plazo de liveness para todo el ciclo de vida, lo que retrasa la detección de bloqueos reales.

2. Las cuatro señales doradas, y por qué la media miente

El SRE Book reduce la instrumentación mínima a cuatro señales:

Señal	Qué mide	Cómo se instrumenta
Latencia	Cuánto tarda, separando éxitos de errores	Histograma, no media
Tráfico	Demanda: peticiones/s	Contador
Errores	Tasa de fallo, explícito e implícito	Contador por código
Saturación	Cuán lleno está el recurso más restrictivo	Medidor

La separación de latencia entre éxitos y errores no es un detalle: los errores suelen ser rápidos, así que mezclarlos baja artificialmente la media justo cuando el servicio está peor.

Y la media es insuficiente por sí misma. Con 1.000 peticiones:

```
950 peticiones a  50 ms
 50 peticiones a 2.000 ms

media  = (950×50 + 50×2000)/1000 = 147,5 ms   ← parece aceptable
p95    = 50 ms
p99    = 2.000 ms                              ← 1 de cada 100 usuarios espera 2 s
```

La media de 147 ms no la experimenta nadie: unos ven 50 ms y otros 2 segundos. Por eso los SLO se fijan sobre percentiles, y por eso las métricas se exportan como histograma —que permite calcular percentiles agregando entre instancias— y no como media previamente calculada, que es matemáticamente imposible de agregar.

La saturación es la más olvidada y la más predictiva: es la única que avisa antes de que la latencia y los errores se degraden. Con lo visto en la clase 011, la saturación por encima de 0,7 anticipa el problema que las otras tres señales aún no muestran.

3. Degradar en vez de caer

No todas las dependencias son iguales, y tratarlas igual convierte cualquier fallo en una caída completa. Clasificarlas es una decisión de producto, no técnica:

Dependencia	Clase	Si no está
Base de datos de pedidos	Esencial	503: no se puede servir
Caché	Opcional	Responder más lento desde origen
Recomendaciones	Opcional	Responder sin esa sección
Telemetría	Opcional	Registrar localmente y continuar

```
async def pagina_producto(pid: str):
    producto = await db.producto(pid)              # esencial: si falla, propaga
    recomendados, degradado = [], []
    try:
        recomendados = await asyncio.wait_for(rec.para(pid), timeout=0.15)
    except (asyncio.TimeoutError, ConnectionError):
        degradado.append("recomendaciones")        # opcional: se anota y sigue
    return {"producto": producto, "recomendados": recomendados, "degradado": degradado}
```

Tres detalles que hacen que esto funcione de verdad:

1. El plazo es obligatorio. Sin timeout, una dependencia lenta bloquea al llamante y propaga la saturación hacia arriba. Una dependencia opcional sin plazo es una dependencia esencial disfrazada.
2. La degradación se declara en la respuesta. El campo degradado permite que el cliente y el panel sepan que el 200 no es completo.
3. El plazo debe ser menor que el del llamante. Si el usuario espera 300 ms, una dependencia opcional con plazo de 500 ms nunca llega a tiempo y solo añade latencia.

4. Una línea base que se pueda repetir

Sin medición previa, «esto mejoró el rendimiento» es una creencia. La línea base debe fijar carga, duración, entorno y percentiles, y guardarse junto al código:

```
$ hey -z 60s -c 50 -q 20 http://localhost:8080/api/productos/A-1042

Summary:
  Total:        60.0031 secs
  Requests/sec: 987.4210

Latency distribution:
  50% in 0.0384 secs
  95% in 0.0912 secs
  99% in 0.1847 secs

Status code distribution:
  [200] 59248 responses
```

Se registra como contrato comparable:

```
{"escenario": "productos-lectura", "concurrencia": 50, "duracion_s": 60,
 "rps": 987.4, "p50_ms": 38.4, "p95_ms": 91.2, "p99_ms": 184.7, "errores": 0}
```

Dos verificaciones con lo aprendido en la clase 011. Ley de Little:

```
L = λ · W = 987 × 0,0384 = 37,9 concurrentes
```

Coherente con los 50 solicitados: el sistema no está saturado, hay holgura. Si L hubiera dado 50 exactos, la concurrencia sería el cuello de botella y la medida estaría limitada por el generador de carga, no por el servicio.

Relación p99/p50 = 184,7/38,4 = 4,8×. Es la métrica que hay que vigilar entre versiones: si sube, la cola se está alargando aunque la media no se mueva.

5. Reproducible significa sin conocimiento tácito

El criterio de aceptación del proyecto no es que funcione en tu máquina: es que otra persona lo levante sin preguntarte nada. Eso exige que el repositorio contenga:

```
projects/cloudshop/
■■■ README.md          qué es, cómo se levanta, cómo se comprueba, cómo se para
■■■ Dockerfile         entorno de ejecución fijado
■■■ compose.yaml       el servicio y sus dependencias
■■■ app.py             el servicio
■■■ smoke.sh           comprobación de extremo a extremo con código de salida
■■■ baseline.json      la línea base medida, con su escenario
```

Y que el recorrido completo quepa en cuatro comandos con salida verificable:

```
$ docker compose up -d
$ ./smoke.sh                    # sale 0 si todo responde; distinto de 0 si no
$ curl -s localhost:8080/metrics | grep -c '^http_request_duration'
$ docker compose down -v
```

El smoke.sh con código de salida no es cosmético: es lo que permite que este mismo recorrido se ejecute en CI en la parte 08 sin cambiar nada. Un script que imprime «OK» pero siempre sale 0 no verifica nada.

Lo que no debe estar: credenciales reales, rutas absolutas de tu máquina, dependencias instaladas globalmente y no declaradas, y pasos que solo tú conoces. Cada uno de esos es conocimiento tácito, y el conocimiento tácito es lo que hace que un servicio sea irrecuperable cuando la persona que lo sabe está de vacaciones.

Ejemplo trabajado

Se ensambla CloudShop v0 y se comprueba con un fallo provocado que degrada en vez de caer.

Línea base con todas las dependencias sanas:

```
$ docker compose up -d && ./smoke.sh
ready: ok  ·  degradado: []
$ hey -z 30s -c 50 http://localhost:8080/api/productos/A-1042 | grep -E '95%|99%|Requests/sec'
  Requests/sec: 987.4210
  95% in 0.0912 secs
  99% in 0.1847 secs
```

Prueba negativa 1 — cae una dependencia opcional. Se detiene el servicio de recomendaciones:

```
$ docker compose stop recomendaciones
$ curl -s -o /dev/null -w '%{http_code}\n' localhost:8080/api/productos/A-1042
200
$ curl -s localhost:8080/api/productos/A-1042 | jq -c '.degradado'
["recomendaciones"]
$ curl -s localhost:8080/health/ready | jq -c '.'
{"status":"ok","degradado":["recomendaciones"]}
```

Sigue respondiendo 200 y declara la degradación. El impacto en latencia se mide, no se supone:

```
p95 con recomendaciones     91,2 ms
p95 sin recomendaciones     73,8 ms      ← más rápido: ya no espera esa llamada
```

Prueba negativa 2 — la dependencia opcional no cae, se vuelve lenta. Este es el caso que de verdad rompe sistemas, porque nada falla:

```
$ docker compose exec recomendaciones tc qdisc add dev eth0 root netem delay 800ms
$ hey -z 30s -c 50 http://localhost:8080/api/productos/A-1042 | grep '95%'
  95% in 0.0947 secs
```

El p95 apenas se mueve: 94,7 ms frente a 91,2. El plazo de 150 ms corta la llamada lenta y el resultado es equivalente a que estuviera caída. Sin ese plazo, el p95 habría subido a más de 800 ms y la saturación se habría propagado hacia arriba: cada petición mantendría un trabajador ocupado 800 ms, y por la ley de Little la concurrencia necesaria se multiplicaría por 8.

Prueba negativa 3 — cae la dependencia esencial.

```
$ docker compose stop db
$ curl -s -o /dev/null -w '%{http_code}\n' localhost:8080/health/ready
503
$ curl -s -o /dev/null -w '%{http_code}\n' localhost:8080/health/live
200                                  # sigue VIVO: no debe reiniciarse
$ docker compose start db && sleep 3
$ curl -s localhost:8080/health/ready | jq -r .status
ok                                   # vuelve solo, sin reiniciar
```

Ese 200 en liveness durante la caída es el resultado que valida el diseño: el orquestador retira la instancia del balanceador pero no la mata, así que cuando la base vuelve hay capacidad caliente inmediata. Si liveness hubiera consultado la base, las nueve réplicas habrían muerto a la vez y la recuperación habría añadido el arranque en frío al tiempo de indisponibilidad.

Evidencia registrada en evidence/: baseline.json, las tres pruebas negativas con su salida, y una limitación explícita — la línea base se midió en una sola máquina sin latencia de red entre componentes; los números no son extrapolables a un despliegue distribuido, solo comparables contra sí mismos.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-00-foundations-computing-networking-linux/012-proyecto-servicio-local-
reproducible-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 servicio-local 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 servicio-local 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
Cae la base de datos y el orquestador reinicia todas las réplicas a la vez	La comprobación de liveness consulta dependencias externas	Liveness solo mira el propio proceso; las dependencias van en readiness.
El panel muestra latencia media aceptable mientras los usuarios se quejan	La media oculta la cola y mezcla errores rápidos con éxitos lentos	Exporta histogramas, separa éxitos de errores y fija los SLO sobre p95 y p99.
Una dependencia opcional se vuelve lenta y satura todo el servicio	La llamada no tenía plazo, así que la dependencia opcional era esencial de hecho	Toda llamada opcional necesita timeout menor que el presupuesto del llamante.
Nadie puede afirmar si un cambio mejoró o empeoró el rendimiento	No existe una línea base reproducible con escenario y percentiles	Registra carga, duración, entorno y p50/p95/p99 en un fichero versionado.
Solo una persona sabe levantar el servicio	El recorrido depende de conocimiento tácito no escrito	Exige que otra persona lo levante siguiendo solo el README, y corrige lo que le falte.

■ 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 liveness no debe consultar la base de datos, y qué ocurre exactamente si lo hace durante una caída de 40 s?
2. Con 950 peticiones a 50 ms y 50 a 2.000 ms, ¿cuál es la media y cuál el p99? ¿Cuál describe mejor la experiencia?
3. Una dependencia opcional se vuelve lenta pero no falla. ¿Qué la convierte de hecho en esencial y cómo se evita?
4. ¿Por qué no se pueden agregar medias de latencia entre instancias, y qué formato de métrica sí lo permite?
5. Tu línea base da 987 rps y p50 de 38 ms con concurrencia 50. Por la ley de Little, ¿estaba saturado el sistema?

Referencias

- Beyer, B. et al., eds. (2016). Site Reliability Engineering, cap. 6 «Monitoring Distributed Systems» — las cuatro señales doradas. <https://sre.google/sre-book/monitoring-distributed-systems/>
- Kubernetes (2024). Configure Liveness, Readiness and Startup Probes — semántica y efectos de cada sonda. <https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>

• Nygard, M. (2018). Release It!, 2.ª ed., caps. 4-5 — patrones de estabilidad: timeouts, bulkheads y circuit breaker.
• Prometheus (2024). Histograms and summaries — por qué los histogramas son agregables y los cuantiles precalculados no. <https://prometheus.io/docs/practices/histograms/>
• Dean, J. y Barroso, L. (2013). The Tail at Scale. Communications of the ACM 56(2), 74-80. <https://doi.org/10.1145/2408776.2408794>
• Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 00 en PDF · Manual integral

Anterior	Índice	Siguiente
← 011 · Costo, energía, capacidad y medición básica	Parte 00 · Programa	013 · Definición NIST y características esenciales de cloud →

Evaluación — 012 Proyecto: servicio local reproducible 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 servicio-local 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.