Multi-Cloud Engineering Program

Parte 09 — Datos, mensajería, serverless e integración

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	09 de 23
Clases	109–120
Evaluaciones	12
Horas estimadas	52

Índice

Parte 09 — Datos, mensajería, serverless e integración

← Parte 08 · Índice completo · Parte 10 →

Descargar: Esta parte en PDF · Manual integral

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

Elegir servicios de datos e integración por sus garantías y patrones de acceso.

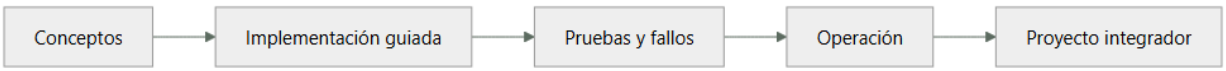
Resultados de la parte

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

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

Secuencia



Clases

ID	Clase	Laboratorio	Horas
109	Bases relacionales administradas y pooling	data	4
110	NoSQL: clave-valor, documento, columna y grafo	data	4
111	Caché, invalidación, TTL y consistencia	data	4
112	Object storage, data lake y formatos columnares	data	4
113	Colas, entrega, reintentos y dead-letter queues	messaging	4
114	Pub/sub, streams, particiones y orden	messaging	4
115	Arquitectura dirigida por eventos y contratos	architecture	4
116	Sagas, outbox, idempotencia y deduplicación	distributed	4
117	Serverless: límites, cold starts y concurrencia	serverless	4
118	API management, cuotas, versiones y monetización	api	4
119	Workflows y orquestación durable	orchestration	4
120	Proyecto: pipeline de pedidos orientado a eventos	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

- Designing Data-Intensive Applications — Martin Kleppmann.
- Enterprise Integration Patterns — Hohpe y Woolf.
- Building Event-Driven Microservices — Adam Bellemare.

109 — Bases relacionales administradas y pooling

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 4 Laboratorio: data · Estado: EXECUTABLECORE

Propósito

Abrir la parte del estado con el servicio de datos más común y peor entendido. Un motor relacional administrado quita el trabajo de instalar, parchear y copiar, y no quita ninguna de las decisiones que importan. La clase se centra en las tres que rompen sistemas en producción: el número de conexiones, que tiene un techo duro y se agota justo cuando el sistema escala; el retardo de las réplicas, que rompe la suposición de leer lo que acabas de escribir; y las decisiones de creación que no se pueden cambiar después.

Resultados de aprendizaje

Al finalizar podrás:

1. Separar lo que el servicio administrado resuelve de lo que sigue siendo tuyo.
2. Calcular cuántas conexiones abrirá tu sistema y compararlo con el techo.
3. Elegir el modo de agrupación de conexiones y saber qué rompe cada uno.
4. Diseñar la lectura en réplicas contando con el retardo.
5. Identificar las decisiones que solo se toman al crear la base de datos.

Conceptos centrales

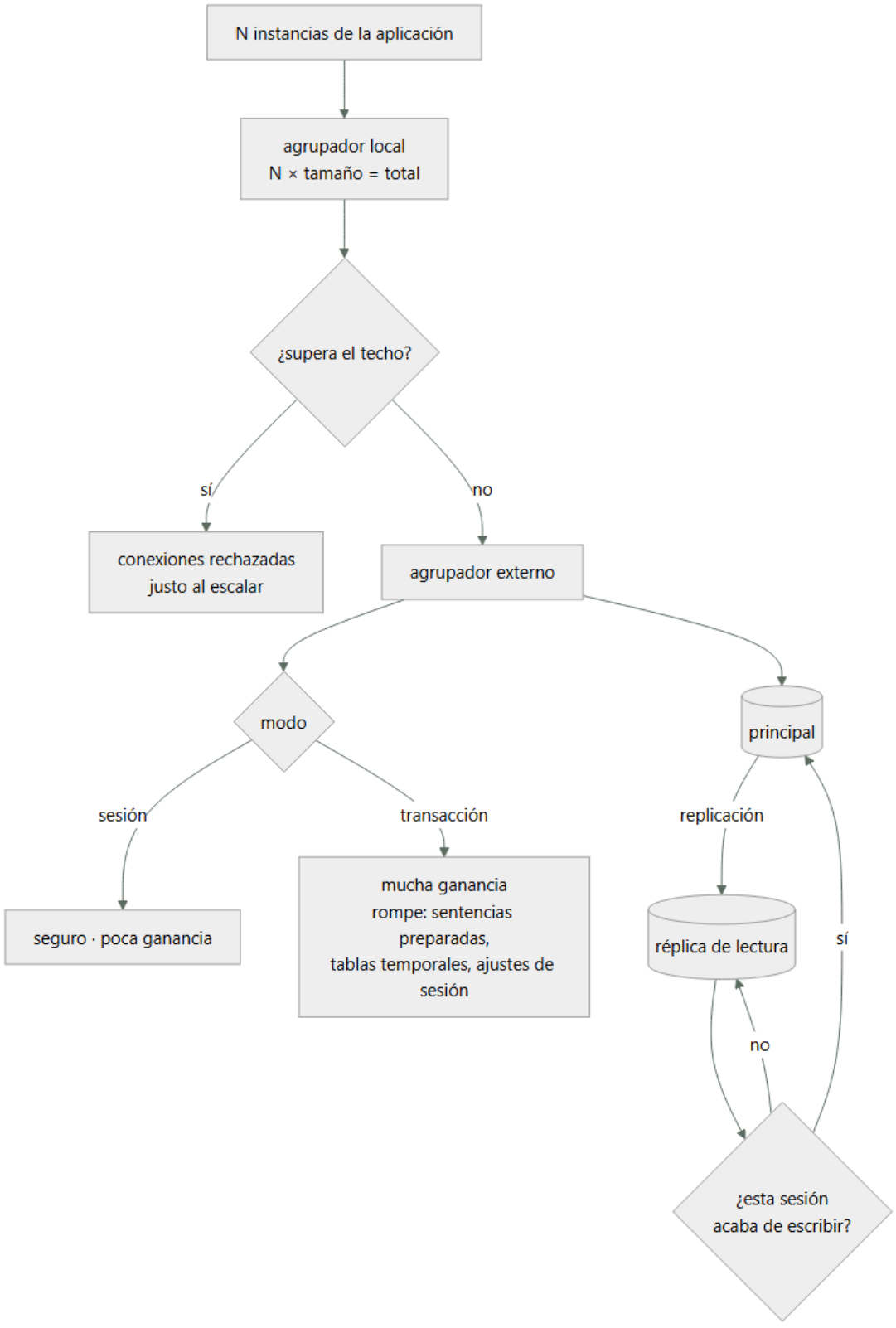
Concepto	Comprensión verificable
servicio administrado	El proveedor opera el motor: instalación, parches, copias, conmutación. El esquema, los índices, las consultas y las conexiones siguen siendo tuyos.
techo de conexiones	Número máximo de conexiones simultáneas. Es un límite duro que sale de la memoria del servidor, no un parámetro que se sube sin consecuencias.
agrupación de conexiones	Reutilizar un conjunto pequeño de conexiones para muchas peticiones. Sin ella, el número de conexiones crece con el número de instancias.
modo de transacción	El agrupador asigna la conexión solo mientras dura la transacción. Multiplica la capacidad y rompe todo lo que dependa de la sesión.
retardo de réplica	Tiempo que tarda un cambio del principal en verse en la réplica. Convierte «leer lo que acabo de escribir» en una suposición falsa.
conmutación	Promoción de una réplica a principal cuando el principal falla. No es instantánea y las transacciones en vuelo se pierden.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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

■ Flujo de razonamiento



Desarrollo

1. Qué resuelve el servicio administrado y qué no

Conviene trazar la línea antes de nada, porque casi todos los incidentes de bases de datos administradas caen del lado que no cubre el proveedor.

```
LO QUE RESUELVE
  instalación y parches del motor
```

```
  copias de seguridad automáticas y restauración a un instante
  réplica y conmutación
  cifrado en reposo y en tránsito
  métricas básicas y registros

LO QUE SIGUE SIENDO TUYO
  el esquema y los índices
  las consultas y sus planes
  el número de conexiones
  el retardo que tolera cada lectura
  las decisiones que se toman al crear
  saber si la copia restaura, que solo se sabe restaurando
```

La última línea repite lo que la clase 088 estableció y sigue siendo cierta aquí: una copia no probada no es una copia. El proveedor garantiza que existe, no que tu sistema vuelva a funcionar con ella.

Y tres parámetros que el servicio expone y que casi nunca se tocan, con lo que pasa si no se tocan:

```
límite de tiempo por sentencia
  sin él, una consulta mal escrita ocupa una conexión indefinidamente
  → y con suficientes, agota el techo

límite de tiempo de transacción inactiva
  sin él, una transacción abierta y olvidada bloquea filas
  y en algunos motores impide limpiar versiones antiguas

registro de consultas lentas
  sin él, el diagnóstico empieza cuando ya hay un incidente
```

El primero es el más rentable de los tres y el más olvidado. Un valor por defecto sensato:

```
ALTER ROLE app SET statement_timeout = '15s';
ALTER ROLE app SET idle_in_transaction_session_timeout = '30s';
```

Y una advertencia sobre el dimensionado que este programa ya vio en la clase 078 con otra forma: la memoria del servidor se reparte entre conexiones y caché. Subir el techo de conexiones no es gratis; se paga en memoria que deja de estar disponible para el caché de datos, y el efecto se nota en las consultas.

2. La cuenta de las conexiones

Este es el problema que aparece en cuanto el sistema escala, y tiene una aritmética que se puede hacer antes de que ocurra.

En los motores de proceso por conexión, cada conexión cuesta memoria fija aunque esté ociosa:

```
servidor de 8 GB
memoria por conexión, entre 5 y 10 MB
techo razonable                        ~200-400 conexiones
```

Y el número que tu sistema abre no depende de tu tráfico: depende de cuántas instancias tienes.

```
instancias de la aplicación                    12
tamaño del agrupador por instancia             10
                                              ■■■■■■■
conexiones abiertas                           120

y cuando el autoescalador sube a 40 instancias
                                              400   ← techo alcanzado
```

Y el momento en que ocurre es el peor posible: el autoescalador sube instancias porque hay carga, y al subirlas agota las conexiones. El síntoma es un fallo total, no una degradación.

Lo mismo con funciones sin servidor, y peor, porque la concurrencia no está acotada por instancias:

```
500 ejecuciones concurrentes × 1 conexión cada una = 500
```

Las tres respuestas, y no son alternativas sino capas:

```
1. AGRUPADOR EN LA APLICACIÓN
   imprescindible, y no resuelve el problema: lo multiplica por instancias
   regla práctica: tamaño pequeño, 5-10, no 50

2. AGRUPADOR EXTERNO
   un proceso intermedio con miles de conexiones de cliente
   y pocas decenas contra la base
```

```
   → es lo que rompe la proporcionalidad con el número de instancias

3. NO ABRIR UNA CONEXIÓN POR PETICIÓN
   agrupar en un servicio intermedio cuando el cliente es efímero
```

Y la cuenta que conviene hacer al dimensionar el agrupador externo, porque el error habitual es ponerlo enorme:

```
conexiones útiles ≈ núcleos del servidor × 2 a 4
→ en un servidor de 8 núcleos, entre 16 y 32
```

Más conexiones activas que eso no aumentan el trabajo hecho: aumentan la competencia por el mismo procesador y por los mismos bloqueos. Una base de datos con 300 conexiones activas hace menos trabajo que la misma con 30.

Y el indicador que avisa antes del fallo:

```
SELECT state, count(*) FROM pg_stat_activity GROUP BY state;
-- active | idle | idle in transaction  ← la tercera creciendo es la señal mala
```

3. Modo de transacción: lo que gana y lo que rompe

El agrupador externo tiene dos modos, y la diferencia entre ellos es la fuente de una clase entera de errores raros.

```
MODO SESIÓN
  la conexión se asigna al cliente hasta que se desconecta
  + no rompe nada
  − la ganancia es pequeña: sigue habiendo una conexión por cliente activo

MODO TRANSACCIÓN
  la conexión se asigna solo mientras dura la transacción
  + 1.000 clientes sobre 25 conexiones reales
  − rompe todo lo que viva en la sesión y no en la transacción
```

Y la lista de lo que rompe, que conviene tener delante porque los síntomas son desconcertantes:

```
sentencias preparadas          se preparan en una conexión y se ejecutan en otra
                               → «la sentencia preparada no existe»
tablas temporales              creadas en una conexión, invisibles en la siguiente
bloqueos de aviso              se toman en una sesión y se liberan en otra
ajustes de sesión              SET aplicado a una conexión que ya no es la tuya
secuencias con currval         valor de la sesión
escucha y notificación         requiere sesión persistente
transacciones que abarcan
varias peticiones              no existen en este modo
```

Y lo característico es que funciona en desarrollo y falla en producción, porque con poca concurrencia el agrupador suele devolver la misma conexión.

Las correcciones, en orden:

```
desactivar las sentencias preparadas en el controlador
  o usar un controlador que las emule en el cliente
sustituir tablas temporales por tablas normales con vida acotada
sustituir bloqueos de aviso por una tabla de bloqueos con expiración
aplicar los ajustes por sentencia, no por sesión
y para lo que necesite sesión —escucha y notificación—, una conexión aparte
  fuera del agrupador
```

La última línea es la solución práctica más frecuente: dos rutas de acceso, una por el agrupador en modo transacción para el tráfico normal y otra directa para lo poco que necesita sesión.

Y una nota sobre el agrupador como punto único de fallo: si todo el tráfico pasa por un proceso, ese proceso necesita el mismo cuidado que la base. Dos instancias detrás de una dirección estable, y una comprobación de salud que mire la base a través del agrupador, no el agrupador solo.

4. Réplicas, conmutación y lo que se decide al crear

Réplicas de lectura. Alivian al principal y traen un problema que la aplicación tiene que resolver: la réplica va por detrás.

```
retardo típico                milisegundos a segundos
retardo bajo carga            segundos a minutos
retardo durante una carga masiva o una migración   mucho peor
```

Y lo que rompe siempre es el mismo patrón:

```
el usuario guarda un pedido        → escribe en el principal
la pantalla siguiente lo lee       → lee de la réplica
```

```
                                                  → «tu pedido no existe»
```

La regla que lo resuelve sin pensar en cada caso:

```
si esta sesión ha escrito en los últimos N segundos → lee del principal
si no                                               → lee de la réplica
```

Y qué puede ir a réplica sin riesgo:

```
sí   informes, listados históricos, búsquedas, exportaciones
no   cualquier lectura que decida una escritura posterior
     comprobar existencia antes de insertar
     leer el saldo antes de descontarlo
```

La segunda lista no es cuestión de retardo: una lectura que decide una escritura necesita coherencia, y la réplica no la da.

Y vigilar el retardo es obligatorio, con una alerta que mira segundos, no bytes:

```
SELECT EXTRACT(EPOCH FROM (now() - pg_last_xact_replay_timestamp()));
```

Conmutación. No es instantánea, y la aplicación tiene que sobrevivir a ella:

```
detección del fallo             10-30 s
promoción de la réplica         10-60 s
propagación del punto de acceso  0-30 s   ← depende del mecanismo
                                ■■■■■■■■■■
total habitual                  30-120 s
```

Y lo que la aplicación debe tener escrito:

```
reconexión automática con reintento y espera creciente
las transacciones en vuelo SE PIERDEN: hay que poder repetirlas
y para poder repetirlas, la operación tiene que ser repetible sin
  efecto adicional  ← esto se ve en la clase 116
no cachear la dirección resuelta más allá de su vigencia
```

Y una prueba que hay que hacer antes de necesitarla: provocar una conmutación en preproducción y medir. Casi siempre aparece algo: un controlador que no reconecta, un caché de nombres demasiado largo, un reintento que duplica.

Lo que se decide al crear. Aquí empieza a comprobarse la predicción de la clase 108 sobre la ley 14:

```
versión mayor del motor        se cambia con migración, con parada o con réplica
conjunto de caracteres y
  ordenación                   cambiarlo obliga a recrear y recargar
cifrado en reposo              en varios proveedores, solo al crear
región y distribución en zonas se cambia recreando
tipo de almacenamiento         a veces sí, a veces recreando
nombre del punto de acceso     queda en cadenas de conexión por todas partes
```

Y una decisión de esquema con el mismo carácter: el tipo de la clave primaria. Cambiar un entero de 4 bytes por uno de 8 en una tabla de mil millones de filas es una migración de días, y el momento de decidirlo es antes de la primera fila.

Y la lista de comprobación de la clase:

- ■ está calculado cuántas conexiones abre el sistema en su máximo
- ■ el agrupador de la aplicación es pequeño y hay agrupador externo
- ■ el modo del agrupador está elegido y se sabe qué rompe
- ■ hay límite de tiempo por sentencia y por transacción inactiva
- ■ se vigila el número de conexiones en estado inactivo-en-transacción
- ■ las lecturas que deciden escrituras no van a réplica
- ■ hay regla de lectura tras escritura por sesión
- ■ se alerta por segundos de retardo de réplica
- ■ se ha provocado una conmutación y se ha medido
- ■ las decisiones de creación están revisadas antes de crear
- ■ la restauración de la copia se ha probado de verdad

Y el cierre que enlaza con la clase siguiente: todo esto asume que el modelo relacional es el adecuado. Cuándo no lo es, y qué se gana y qué se pierde al cambiarlo, es la materia de la clase 110.

Ejemplo trabajado

El servicio de pedidos de CloudShop pasó a un motor relacional administrado. En los primeros ocho meses tuvo cuatro incidentes, y los cuatro están en las cuatro secciones de esta clase. Se resuelven en orden.

Incidente 1: el sistema cae justo cuando llega tráfico.

```
11:40  campaña de marketing, tráfico ×3
11:41  el autoescalador sube de 12 a 34 instancias
11:41  errores de conexión: «demasiadas conexiones»
11:52  caída total, 11 minutos
```

La cuenta que nadie había hecho:

```
tamaño del agrupador por instancia                  25
instancias en reposo                                12   →  300 conexiones
techo configurado                                  340
instancias con el pico                              34   →  850 conexiones
```

Y el detalle que lo hace peor: la aplicación abría el agrupador entero al arrancar, así que una instancia nueva consumía 25 conexiones antes de servir una sola petición.

Las dos correcciones y su efecto:

```
tamaño por instancia                      25            6                6
conexiones con 34 instancias             850          204               24
conexiones activas contra la base        850          204               24
peticiones por segundo servidas        fallo        1.180            1.310
latencia p95                           fallo       310 ms           240 ms
```

La última columna es lo contraintuitivo: con 24 conexiones se sirve más y más rápido que con 204. El servidor tenía 8 núcleos; las otras 180 conexiones solo competían entre sí.

Incidente 2: errores que no se reproducen en preproducción.

Al activar el agrupador externo en modo transacción:

```
errores «la sentencia preparada s_3 no existe»    en producción, ~1 de cada 400
en preproducción                                  cero
```

La causa es la del apartado tercero: con poca concurrencia el agrupador devolvía casi siempre la misma conexión. El inventario de lo que había que cambiar:

```
sentencias preparadas del controlador       → desactivadas
tablas temporales en 2 informes             → tablas normales con vida acotada
bloqueos de aviso en el proceso nocturno    → tabla de bloqueos con expiración
escucha/notificación en 1 servicio          → conexión directa, fuera del agrupador
SET de zona horaria por sesión              → por sentencia
```

Y la prueba que faltaba, que se añadió a la canalización: generar concurrencia suficiente en preproducción para que el agrupador reparta conexiones distintas. Con 200 clientes concurrentes, el error aparecía en preproducción a los 40 segundos.

Incidente 3: «mi pedido no aparece».

```
quejas de clientes en 6 semanas                        31, sin relacionar
causa                          la pantalla de confirmación leía de la réplica
retardo medio de la réplica                            180 ms
retardo durante el proceso nocturno de informes        41 s
```

Y lo interesante: el retardo medio era irrelevante. El daño se concentraba en la ventana del proceso nocturno, cuando la réplica se quedaba a cuarenta segundos.

La corrección, con la regla del apartado cuarto:

```
lecturas en réplica                  todas las de lectura   solo informes
regla de lectura tras escritura      no había          5 s por sesión
lecturas que deciden escrituras      en réplica        en principal
alerta de retardo                    no había          > 10 s
quejas por pedido no visible         31 / 6 semanas    0
```

Y al revisar «lecturas que deciden escrituras» apareció algo peor que las quejas: la comprobación de existencia antes de insertar un pedido leía de la réplica, lo que permitía duplicados durante el retardo. Se encontraron 14 pedidos duplicados en el histórico.

Incidente 4: la conmutación.

La primera conmutación real ocurrió a los siete meses:

```
detección + promoción                        74 s
tiempo hasta que la aplicación se recuperó  6 min 20 s   ← el problema
```

La diferencia entre 74 segundos y 6 minutos:

```
caché de resolución de nombres del contenedor        300 s
el agrupador externo no reintentaba: se quedó fijo
el controlador no reconectaba tras el error inicial
transacciones en vuelo                                47, perdidas
de ellas, repetidas automáticamente sin duplicar       0
de ellas, que crearon un pedido duplicado al repetir   3
```

Las correcciones, y el ensayo que las validó:

```
caché de nombres                          300 s          30 s
reintento del agrupador                   no             sí
reconexión del controlador                no             sí, con espera creciente
operación repetible sin efecto adicional  no             clase 116

conmutación provocada en preproducción    nunca          trimestral
tiempo de recuperación medido           6 min 20 s      52 s
```

A los ocho meses.

```
conexiones contra la base en el pico            850            24
peticiones por segundo servidas                fallo         1.310
latencia p95                                     —           240 ms
límite de tiempo por sentencia                no había         15 s
conexiones inactivas-en-transacción          hasta 60          0-2
quejas por lectura tras escritura           31 / 6 sem          0
pedidos duplicados por lectura en réplica        14             0
recuperación tras conmutación               6 min 20 s        52 s
ensayo de conmutación                         nunca          trimestral
restauración de copia probada                 nunca          trimestral
```

La lección que esta clase abre para la parte 09: los cuatro incidentes tienen algo en común que no tenían los de las partes anteriores. Ninguno se arreglaba desplegando otra versión. El primero era una cuenta que nadie hizo, el segundo una propiedad del intermediario, el tercero una suposición sobre el tiempo y el cuarto un evento que no se había ensayado. La reversión, el canario y el entorno efímero —los mecanismos de la parte 08— no intervinieron en ninguno, que es justo lo que la hipótesis de la clase 108 predijo que iba a pasar.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/109-bases-relacionales-administradas-y-pooling/lab.py
```

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

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa modelo-relacional 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 modelo de datos ligado a patrones de acceso. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye modelo-relacional 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 falla por completo justo cuando sube el tráfico	El autoescalador añade instancias y cada una abre su agrupador; se agota el techo de conexiones	Agrupador pequeño en la aplicación más un agrupador externo, y calcula el máximo de conexiones antes de que ocurra.
Errores de sentencia preparada inexistente que no se reproducen fuera de producción	Modo de transacción: la sesión no persiste entre sentencias, y con poca concurrencia el agrupador devuelve la misma conexión	Desactiva las sentencias preparadas del controlador, sustituye lo que dependa de sesión y genera concurrencia real en preproducción.
Más conexiones y menos trabajo hecho	Las conexiones activas superan con mucho a los núcleos y compiten entre sí	Dimensiona las conexiones útiles en núcleos × 2 a 4 y deja que el agrupador encole el resto.
El usuario no ve lo que acaba de guardar	La lectura posterior va a una réplica con retardo	Regla de lectura tras escritura por sesión durante unos segundos, y ninguna lectura que decida una escritura en réplica.
La conmutación tarda un minuto y la aplicación tarda seis	Caché de nombres largo, agrupador sin reintento y controlador que no reconecta	Acorta el caché, activa reintentos con espera creciente y ensaya una conmutación cada trimestre.
Una consulta mal escrita degrada todo el sistema	No hay límite de tiempo por sentencia ni por transacción inactiva	Fija ambos límites en el rol de la aplicación y vigila las conexiones en estado inactivo-en-transacción.

■ 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 resuelve un servicio de base de datos administrado y qué sigue siendo responsabilidad tuya?
2. ¿Cómo se calcula el número de conexiones que abrirá el sistema y por qué falla al escalar?
3. ¿Qué gana el modo de transacción y qué cinco cosas rompe?
4. ¿Qué lecturas no deben ir nunca a una réplica y por qué?
5. ¿Qué decisiones de una base de datos solo se pueden tomar al crearla?

Referencias

- PostgreSQL (2025). Connections and resource consumption — coste por conexión y parámetros de límite de tiempo. <https://www.postgresql.org/docs/current/runtime-config-connection.html>
- PgBouncer (2025). Pooling modes — sesión, transacción y sentencia, y lo que cada modo rompe. <https://www.pgbouncer.org/features.html>
- AWS (2025). RDS Proxy: connection pooling and failover — agrupación gestionada y comportamiento durante la conmutación. <https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/rds-proxy.html>
- Azure (2025). Azure Database for PostgreSQL: read replicas and lag — retardo de replicación y cuándo no usar réplica. <https://learn.microsoft.com/azure/postgresql/flexible-server/concepts-read-replicas>
- Google Cloud (2025). Cloud SQL: high availability and failover — tiempos de conmutación y requisitos de la aplicación. <https://cloud.google.com/sql/docs/postgres/high-availability>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 09 en PDF · Manual integral

Anterior	Índice	Siguiente
← 108 · Proyecto: fábrica de software multi-cloud	Parte 09 · Programa	110 · NoSQL: clave-valor, documento, columna y grafo →

Evaluación — 109 Bases relacionales administradas y pooling

Preguntas

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

Reto verificable

Entrega modelo-relacional 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.

110 — NoSQL: clave-valor, documento, columna y grafo

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 4 Laboratorio: data · Estado: EXECUTABLECORE

Propósito

Decidir cuándo un almacén no relacional es la elección correcta, con un criterio que no es «escala» sino otro: cuánto sabes de antemano sobre cómo vas a consultar los datos. La clase recorre las cuatro familias por lo que resuelven en un acceso directo y por lo que no pueden hacer, y se detiene en la decisión que la clase 108 predijo que dominaría esta parte —la clave de partición, que se elige al crear, decide el reparto y no se cambia— y en su consecuencia más cara: una partición caliente.

Resultados de aprendizaje

Al finalizar podrás:

1. Decidir entre relacional y no relacional por lo que sabes de los accesos, no por el volumen.
2. Distinguir las cuatro familias por lo que cada una resuelve en un acceso directo.
3. Diseñar una clave de partición que reparta, y detectar una caliente.
4. Modelar por consulta, asumiendo el coste de las consultas que aún no existen.
5. Elegir el nivel de consistencia de cada lectura y saber qué se paga por él.

Conceptos centrales

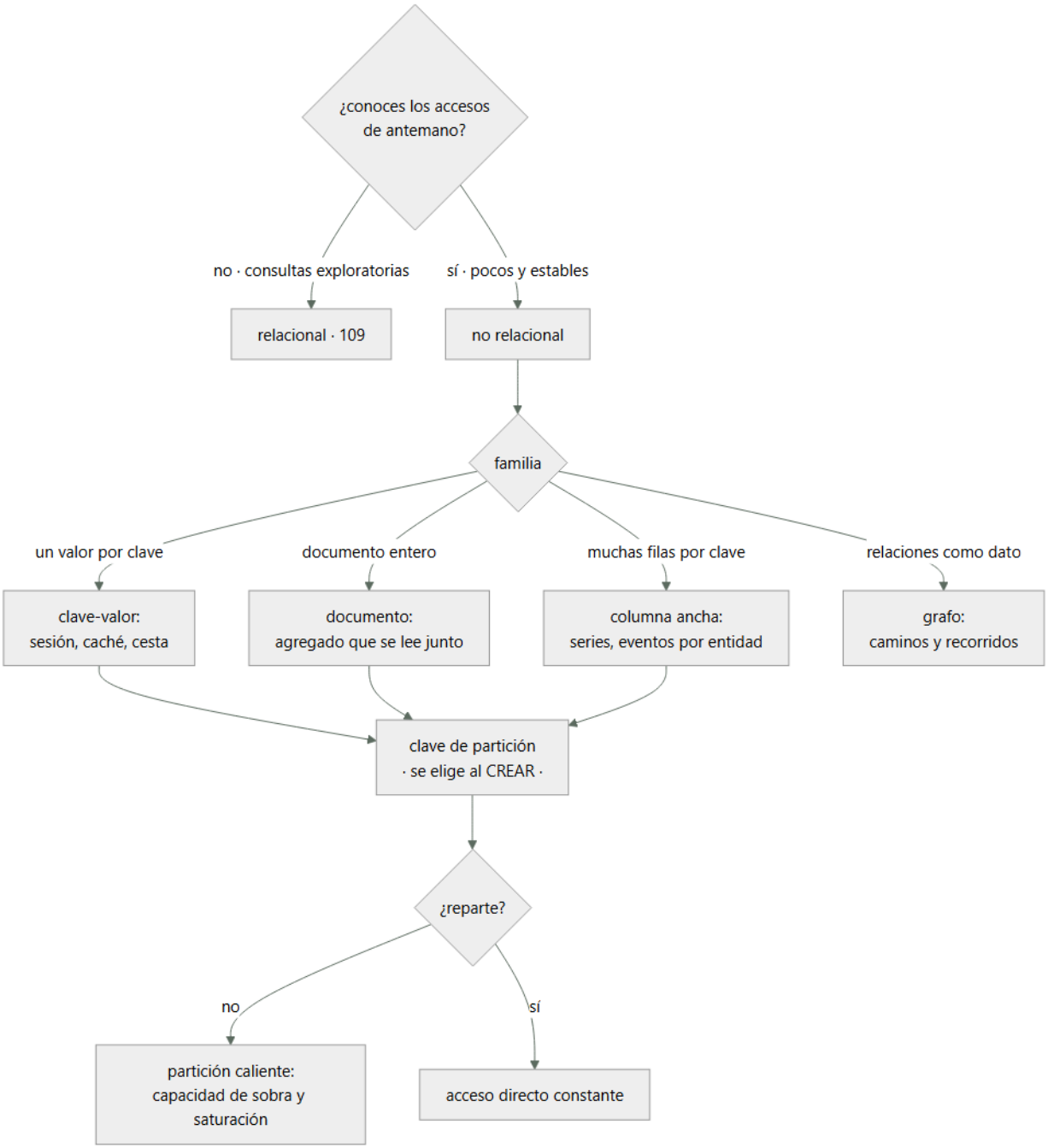
Concepto	Comprensión verificable
modelar por consulta	Diseñar la disposición de los datos a partir de los accesos conocidos. Lo contrario del modelo relacional, que normaliza primero y consulta después.
clave de partición	Campo que decide en qué partición vive cada elemento. Se elige al crear la tabla, determina el reparto de carga y no se cambia sin recrear.
partición caliente	Partición que recibe una proporción desmedida del tráfico. El sistema tiene capacidad de sobra y esa partición está saturada.
clave de ordenación	Segundo componente de la clave que ordena los elementos dentro de la partición. Es lo que permite recorrer rangos con un solo acceso.
consistencia eventual	Una lectura puede devolver un valor anterior durante un intervalo corto. Es más barata y más rápida; no siempre es aceptable.
recorrido completo	Leer la tabla entera porque la consulta no encaja con las claves. En estos sistemas es lento y se factura por cada elemento leído.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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 pregunta que decide, y no es el volumen

El criterio habitual —«relacional hasta que no escale»— es malo porque el volumen rara vez es el problema. Un motor relacional administrado con índices correctos sirve millones de filas sin dificultad.

La pregunta que sí decide:

```
¿sabes de antemano cómo vas a consultar estos datos?

  no, y van a aparecer consultas nuevas       → relacional
  sí, son pocos accesos y no van a cambiar    → no relacional
```

Y el motivo es que los dos modelos invierten el orden del diseño:

```
RELACIONAL      modelas los DATOS: entidades, relaciones, formas normales
                y luego consultas como quieras
                el precio: el motor tiene que resolver combinaciones en
                tiempo de consulta
```

```
NO RELACIONAL   modelas las CONSULTAS: cada acceso conocido tiene su
                disposición de datos
                el precio: una consulta que no previste puede ser inviable
```

Y hay tres motivos legítimos para elegir no relacional que sí se sostienen:

```
1. accesos por clave, muchísimos y muy simples
   sesiones, cestas, perfiles, caché persistente
   → el relacional también puede, y aquí no aporta nada su capacidad extra

2. escritura muy alta y sostenida sobre una entidad
   series temporales, eventos por dispositivo, registros
   → el reparto por partición es exactamente lo que hace falta

3. el propio dato es una red de relaciones que se recorren
   → esto es el grafo, y no lo hace bien ningún otro modelo
```

Y un motivo que no se sostiene y se usa mucho: «no queremos definir un esquema». El esquema no desaparece; se muda del motor al código, donde nadie lo valida.

```
sin esquema en el motor
→ conviven cinco formas del mismo documento, escritas en tres años
→ y cada lectura tiene que saber tratarlas todas
```

La forma sana de trabajar sin esquema declarado es versionar el documento y tener una ruta de migración, no ignorar el problema:

```
{"v": 3, "id": "c-1421", "nombre": "…", "direcciones": [ … ]}
```

2. Cuatro familias, por lo que resuelven

```
CLAVE-VALOR
  resuelve   dame el valor de esta clave
  no resuelve  cualquier consulta que no sea por la clave
  encaja en  sesión, cesta, preferencias, resultados de cálculo
  cuidado    el valor entero se lee y se escribe: no metas dentro cosas
             que cambian a ritmos distintos

DOCUMENTO
  resuelve   dame este agregado completo, y filtra por campos indexados
  no resuelve  combinaciones entre colecciones a gran escala
  encaja en  entidades que se leen y escriben juntas: pedido con sus líneas
  cuidado    si el documento crece sin límite —un pedido con 50.000 eventos—
             el modelo se rompe

COLUMNA ANCHA
  resuelve   dame las filas de esta entidad, en este rango de tiempo
  no resuelve  consultas que no empiecen por la clave de partición
  encaja en  series temporales, eventos por usuario o por dispositivo
  cuidado    borrar es escribir marcas de borrado; borrar mucho hace daño

GRAFO
  resuelve   recorridos: de aquí a allá pasando por N saltos
  no resuelve  agregaciones masivas sobre todo el grafo
  encaja en  permisos heredados, recomendaciones, detección de fraude,
             dependencias
  cuidado    es el único de los cuatro cuya alternativa relacional es
             realmente mala: una consulta de profundidad variable en
             tablas es dolorosa
```

Y una regla de tamaño que evita disgustos en las tres primeras familias:

```
si un elemento puede crecer sin límite, no es un elemento:
es una colección, y necesita su propia clave
```

El ejemplo clásico: el pedido con su historial de eventos dentro. Funciona hasta que un pedido tiene mil eventos, y entonces cada lectura del pedido lee los mil.

Y sobre el grafo, una precisión práctica: casi siempre convive con otro almacén. El grafo guarda las relaciones y los identificadores; los atributos completos viven en el sistema principal. Duplicar todo en el grafo es lo que hace que los grafos se abandonen.

3. La clave de partición: se elige al crear

Aquí está la decisión que la clase 108 predijo como dominante en esta parte, y es exactamente eso: se elige al crear la tabla, decide cómo se reparte todo y cambiarla es recrear y recargar.

La clave suele tener dos partes:

```
clave de PARTICIÓN     decide en qué partición vive el elemento
clave de ORDENACIÓN    ordena dentro de la partición

acceso directo         partición + ordenación exactas
recorrido de rango     partición exacta + rango de ordenación
cualquier otra cosa    recorrido completo
```

Y la propiedad que hay que exigirle a la clave de partición:

```
CARDINALIDAD ALTA        muchos valores distintos
REPARTO UNIFORME         ninguno concentra el tráfico
CONOCIDA EN LA CONSULTA  si no la sabes al consultar, no sirve
```

Las tres a la vez. Y los fallos típicos son incumplir una:

```
estado del pedido como clave     5 valores → 5 particiones → fallo de cardinalidad
fecha del día como clave         todo el tráfico de hoy en una → fallo de reparto
identificador interno aleatorio  reparte muy bien y no lo sabes al consultar
                                 → fallo de conocimiento
```

La partición caliente es lo que ocurre al fallar el reparto, y su síntoma engaña:

```
capacidad de la tabla        10.000 operaciones/s
límite por partición          1.000 operaciones/s
tráfico real                  3.000 operaciones/s

si el 80 % cae en una partición → 2.400 sobre un límite de 1.000
→ errores de limitación con la tabla al 30 % de su capacidad
```

Y las tres correcciones, en orden de preferencia:

```
1. cambiar la clave por una que reparta de verdad
   → recrear; es la ley 14 cobrando su precio

2. añadir un sufijo de reparto artificial
   clave = "producto#1421#3"  con 3 de 0..9
   → reparte, y ahora cada lectura tiene que consultar los 10 sufijos

3. poner delante un caché para el elemento caliente
   → resuelve las lecturas y no las escrituras
```

La segunda es la que más se usa y conviene entender su precio: multiplica por diez el coste de leer ese elemento. Es un intercambio, no una solución gratis.

Y un caso que aparece siempre y que las tres correcciones no cubren: el elemento genuinamente popular. Un producto en portada recibe cien mil lecturas por segundo y ninguna clave lo reparte, porque es una sola cosa. Ahí la respuesta es la tercera, y es la clase 111.

4. Modelar por consulta, y lo que cuesta después

Si el acceso decide la disposición, cada acceso nuevo necesita una disposición nueva. De ahí salen los índices secundarios y el diseño de tabla única:

```
ÍNDICE SECUNDARIO
  otra clave de partición sobre los mismos datos
  + permite un acceso que la clave principal no permite
  − se paga en escritura: cada escritura actualiza también el índice
  − y suele ser eventualmente consistente

TABLA ÚNICA
  varios tipos de entidad conviviendo con claves compuestas
  PK="CLI#1421"  SK="PERFIL"
  PK="CLI#1421"  SK="PEDIDO#2026-07-31#88"
  + un solo acceso trae el cliente y sus últimos pedidos
  − ilegible sin documentación, y la migración es un proyecto
```

Y la consecuencia honesta que hay que aceptar antes de empezar:

```
una consulta que no estaba prevista cuesta
  un índice nuevo (dinero y escritura), o
  un recorrido completo (mucho dinero y lentitud), o
  copiar los datos a otro sistema
```

La tercera opción es la que acaba adoptando casi todo el mundo, y es legítima: el almacén sirve al producto, y un segundo sistema sirve al análisis. Es la materia de la clase 112.

El coste, que en estos sistemas es parte del diseño. Se factura por operación y por datos leídos, así que un acceso mal modelado es directamente una factura:

```
leer un elemento por clave                    1 unidad
recorrer 100 elementos de una partición     ~ proporcional al tamaño leído
recorrido completo de 10 millones          decenas de miles de unidades
                                           y se paga aunque el filtro
                                           descarte casi todo
```

La última línea es la trampa: filtrar no ahorra. El filtro se aplica después de leer, así que se paga por todo lo leído.

Consistencia. Casi todos ofrecen dos niveles de lectura, y la elección es por acceso, no global:

```
eventual   más barata y más rápida; puede devolver un valor anterior
fuerte     ve la última escritura confirmada; cuesta más y tarda más
```

Y la regla es la misma de la clase 109, porque el problema es el mismo:

```
lectura que solo se muestra              eventual
lectura que decide una escritura         fuerte
```

Y las transacciones: existen, y casi siempre acotadas —a una partición, a un número de elementos, a una región—. Diseñar suponiendo transacciones amplias es el error que la clase 116 tendrá que reparar.

Y la lista de comprobación de la clase:

- [ ] están escritos los accesos conocidos antes de elegir el almacén
- [ ] la clave de partición tiene cardinalidad, reparte y se conoce al consultar
- [ ] se ha estimado el tráfico de la partición más caliente, no solo el total
- [ ] ningún elemento puede crecer sin límite
- [ ] los documentos llevan versión y hay ruta de migración
- [ ] cada índice secundario tiene un acceso que lo justifica
- [ ] no hay recorridos completos en el camino de una petición
- [ ] el nivel de consistencia está elegido por acceso
- [ ] está previsto dónde irán las consultas no previstas
- [ ] el coste por operación está estimado con el patrón real

Y el cierre que enlaza con la clase siguiente: el elemento genuinamente popular no lo reparte ninguna clave, y la respuesta es ponerle algo delante. Qué se gana, qué se rompe y por qué invalidar es difícil es la materia de la clase 111.

Ejemplo trabajado

CloudShop mueve tres conjuntos de datos a almacenes no relacionales. Uno sale bien, otro provoca un incidente de partición caliente y el tercero enseña lo que cuesta una consulta no prevista.

Caso 1: la sesión y la cesta. Sale bien y conviene entender por qué.

```
accesos conocidos     leer por identificador de sesión
                      escribir por identificador de sesión
                      caducar a las 72 h
accesos futuros       ninguno previsible
```

Un único acceso por clave, sin consultas exploratorias. Es el caso ideal.

```
latencia p99 de lectura                    14 ms          1,2 ms
escrituras por segundo en el pico           3.100         3.100
carga que quitó a la base principal           —          41 % de escrituras
caducidad                                proceso nocturno  automática
```

Y el detalle que evitó un problema conocido: la cesta y el histórico de navegación se guardaron por separado, aunque pertenecen a la misma sesión. El histórico cambia en cada clic y la cesta pocas veces; juntos habrían obligado a reescribir la cesta entera constantemente.

Caso 2: los eventos de pedido. Partición caliente en el segundo día.

Se eligió una tabla de columna ancha para el historial de eventos de cada pedido. La primera clave:

```
clave de partición   fecha (2026-07-31)
clave de ordenación  marca de tiempo + identificador de pedido
motivo               «así los informes diarios son un solo recorrido»
```

El segundo día en producción:

```
capacidad de la tabla                  20.000 escrituras/s
límite por partición                    1.000 escrituras/s
tráfico real                            2.900 escrituras/s
en la partición de hoy                  2.900   ← el 100 %
errores de limitación                   el 66 % de las escrituras
utilización de la tabla                 14 %
```

Catorce por ciento de utilización y dos tercios de las escrituras rechazadas. Es el síntoma que engaña del apartado tercero, en su forma más pura: la clave se eligió por la consulta de informes y no por el reparto de la escritura.

La corrección exigió recrear la tabla y recargar 210 millones de elementos:

```
clave de partición                     fecha              identificador de pedido
clave de ordenación                    tiempo + pedido    marca de tiempo
particiones activas                    1                  ~2,4 millones
escrituras rechazadas                  66 %               0 %
leer el historial de un pedido         recorrido           1 acceso
informe diario                         1 recorrido         ya no es posible
tiempo de la migración                   —                 31 h
```

Y la última fila es el intercambio real: se ganó la escritura y se perdió el informe. El informe se resolvió publicando los eventos también a almacenamiento de objetos, que es la clase 112.

Y la lección de la ley 14, medida: la decisión se tomó en veinte minutos de diseño y costó treinta y una horas de migración y dos días de errores.

Caso 3: el catálogo, y la consulta que nadie previó.

El catálogo se modeló por sus tres accesos conocidos:

```
por identificador de producto          acceso directo
por categoría, ordenado por ventas     índice secundario
por marca                              índice secundario
```

A los cuatro meses, el equipo de producto pidió: «productos con existencias por debajo de N en cualquier categoría».

```
opción A  índice nuevo sobre existencias
          → existencias cambian en cada venta
          → 4.100 escrituras/s adicionales al índice
          → coste estimado: 890 €/mes

opción B  recorrido completo cada 15 min
          → 2,1 millones de elementos leídos por ejecución
          → coste estimado: 1.240 €/mes
          → y el filtro no ahorra: se paga por todo lo leído

opción C  publicar los cambios a un segundo sistema para consultas libres
          → coste estimado: 210 €/mes, con retardo de ~1 min
```

Se eligió la C. Y la conclusión que se escribió en la decisión, que es la que este apartado defiende:

> «El almacén sirve al producto con sus accesos conocidos. Cualquier consulta nueva y exploratoria va al segundo sistema. Añadir un índice por cada pregunta nueva convierte la escritura en el cuello de botella.»

Y un cuarto caso que no se hizo: el grafo.

Se evaluó un almacén de grafo para las recomendaciones y se descartó, con el motivo escrito:

```
recorridos necesarios              2 saltos, siempre
alternativa relacional             una tabla de asociación y dos combinaciones
latencia medida en relacional      18 ms
coste de operar un sistema más     alto
decisión                           no; se reconsiderará si aparecen
                                   recorridos de profundidad variable
```

Es el criterio del apartado segundo: el grafo gana cuando la profundidad es variable. Con dos saltos fijos, no aporta lo bastante para justificar un sistema más.

Al año.

```
latencia p99 de sesión                        14 ms          1,2 ms
escrituras quitadas a la base principal          —           41 %
escrituras rechazadas por partición caliente     —           0 %
migraciones forzadas por la clave de partición   —           1 (31 h)
índices secundarios añadidos por consultas
  nuevas                                         —           0
consultas exploratorias                    en el almacén   en el segundo sistema
sistemas de datos en producción                  1             4
```

La última fila es el coste que no aparece en ninguna comparativa de rendimiento: se pasó de un sistema a cuatro, y cada uno tiene su copia, su vigilancia, su plan de recuperación y su forma de fallar.

La lección que esta clase traslada a la parte 09: la migración de treinta y una horas no la causó un problema de escala. La causó elegir la clave de partición por la consulta de informes en vez de por el reparto de la escritura, en una decisión de veinte minutos que no se podía deshacer. Es la primera confirmación explícita de la predicción de la clase 108: en los servicios con estado, las decisiones de creación son las caras.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/110-nosql-clave-valor-documento-columna-y-grafo/lab.py
```

El laboratorio selecciona el motor de práctica data 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-nosql 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 modelo de datos ligado a patrones de acceso. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye matriz-nosql 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
Errores de limitación con la tabla al 15 % de su capacidad	Partición caliente: la clave no reparte y el tráfico se concentra	Estima el tráfico de la partición más cargada, no el total; corrige la clave, y si no puedes, reparte con sufijo asumiendo que multiplica el coste de lectura.
Cada consulta nueva exige un índice y la escritura se satura	Se está usando un almacén modelado por consulta para consultas exploratorias	Publica los cambios a un segundo sistema para las consultas libres y deja el almacén para los accesos conocidos.
La factura crece con consultas que devuelven pocos resultados	Hay recorridos completos: el filtro se aplica tras leer, y se paga por todo lo leído	Rediseña el acceso para que use clave de partición y rango, o muévelo al segundo sistema.
Leer una entidad se vuelve lento con el tiempo	Un elemento crece sin límite porque contiene una colección dentro	Si algo puede crecer sin límite, dale su propia clave en vez de anidarlo.
El mismo documento existe en cinco formas distintas	Sin esquema declarado, el esquema se mudó al código y nadie lo valida	Versiona el documento, valida al escribir y ten una ruta de migración.
Un producto popular satura su partición y ninguna clave lo reparte	Es un solo elemento genuinamente caliente	Pon un caché delante para las lecturas; ninguna clave reparte lo que es una sola cosa.

Seguridad, ética y costo

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

Preguntas de comprobación

1. ¿Qué pregunta decide entre relacional y no relacional, y por qué no es el volumen?
2. ¿Qué tres propiedades debe cumplir una clave de partición a la vez?
3. ¿Por qué una tabla puede rechazar escrituras estando al 15 % de su capacidad?
4. ¿Qué precio tiene repartir con un sufijo artificial?
5. ¿Por qué filtrar no reduce el coste de un recorrido completo?

Referencias

- AWS (2025). DynamoDB: partition keys and best practices — cardinalidad, reparto y particiones calientes. <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-partition-key-design.html>
- Google Cloud (2025). Bigtable schema design — clave de fila, reparto y antipatrones. <https://cloud.google.com/bigtable/docs/schema-design>
- Azure (2025). Cosmos DB: partitioning and horizontal scaling — clave de partición lógica y física. <https://learn.microsoft.com/azure/cosmos-db/partitioning-overview>
- Kleppmann, M. (2017). Designing Data-Intensive Applications, caps. 2 y 6 — modelos de datos y particionado. <https://dataintensive.net/>
- Neo4j (2025). When to use a graph database — profundidad variable como criterio de elección. <https://neo4j.com/docs/getting-started/data-modeling/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 09 en PDF · Manual integral

Anterior	Índice	Siguiente
← 109 · Bases relacionales administradas y pooling	Parte 09 · Programa	111 · Caché, invalidación, TTL y consistencia →

Evaluación — 110 NoSQL: clave-valor, documento, columna y grafo

Preguntas

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

Reto verificable

Entrega matriz-nosql 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.

111 — Caché, invalidación, TTL y consistencia

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 4 Laboratorio: data · Estado: EXECUTABLECORE

Propósito

Poner un caché delante de lo que la clase 110 dejó sin resolver —el elemento genuinamente popular— y hacerlo entendiendo las tres cosas que casi nunca se dicen: que la proporción de aciertos no mide nada por sí sola, que invalidar es el problema difícil y tiene una solución sencilla que casi nadie usa, y que un caché del que el sistema no puede prescindir ha dejado de ser una optimización para convertirse en una dependencia con requisito de disponibilidad.

Resultados de aprendizaje

Al finalizar podrás:

1. Decidir qué cachear a partir del coste de los fallos, no de la proporción de aciertos.
2. Elegir el patrón de lectura y escritura, y saber qué garantiza cada uno.
3. Invalidar con claves versionadas en vez de perseguir cada ruta de escritura.
4. Prevenir avalancha, penetración y arranque en frío.
5. Distinguir un caché que optimiza de uno del que el sistema depende.

Conceptos centrales

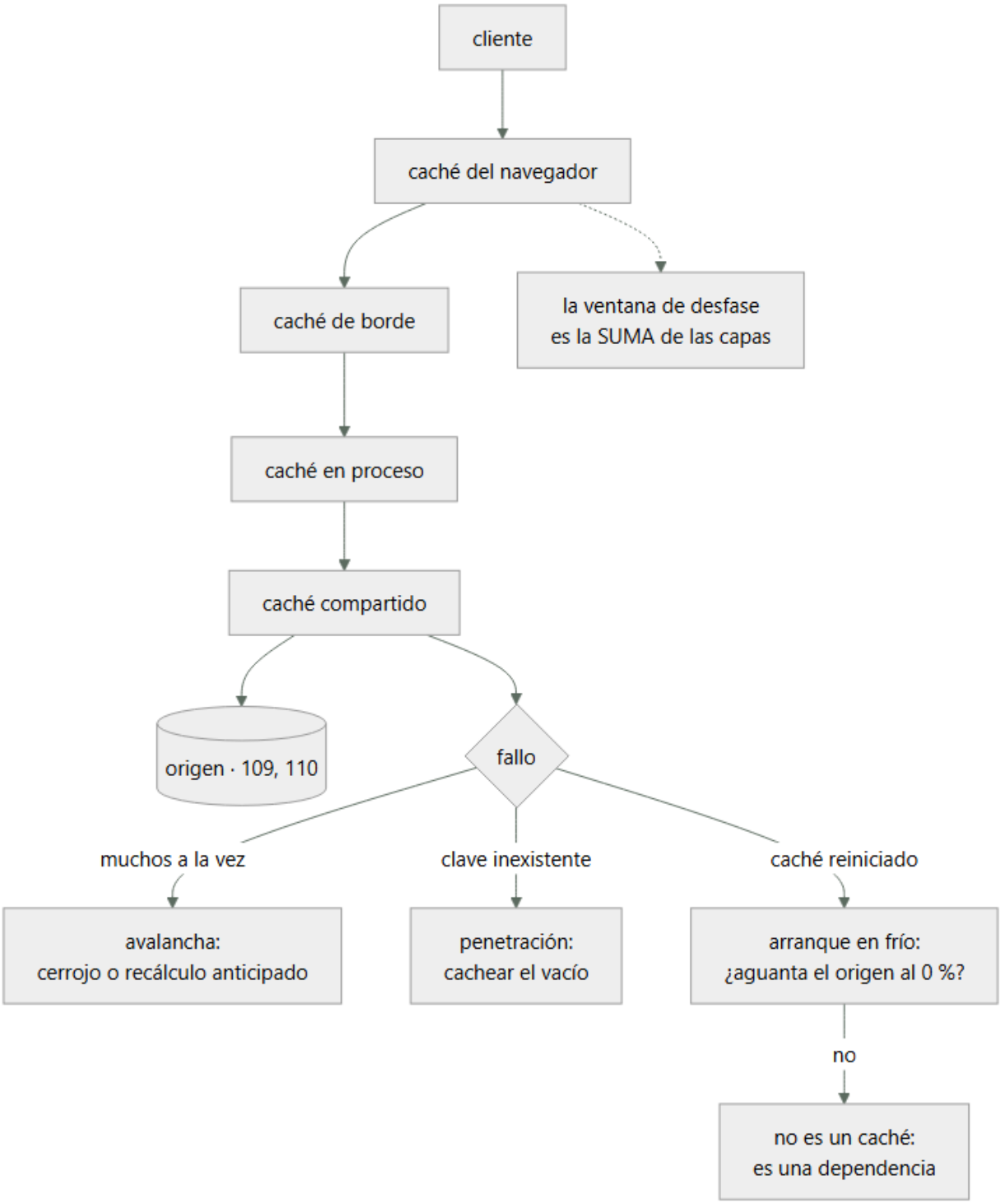
Concepto	Comprensión verificable
caché aparte	La aplicación consulta el caché, y si falla, lee el origen y guarda. El patrón más común y el que deja más decisiones en manos de quien escribe el código.
clave versionada	Incluir en la clave un número que cambia al modificar el dato. Invalidar deja de ser borrar: el valor viejo simplemente deja de consultarse.
avalancha	Muchas peticiones fallan a la vez sobre la misma clave —normalmente al caducar— y todas van al origen simultáneamente.
penetración	Peticiones por claves que no existen: nunca aciertan y siempre llegan al origen. Es el patrón que un atacante usa para saltarse el caché.
ventana de desfase	Tiempo máximo durante el que se puede servir un valor viejo. Con varias capas, se suman.
caché portante	Caché sin el cual el origen no aguanta el tráfico. Deja de ser una optimización y pasa a necesitar disponibilidad, réplica y plan de recuperación.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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

■ Flujo de razonamiento



Desarrollo

1. Qué compra un caché, y qué medida sirve

Un caché no hace que el sistema sea más rápido: hace que el caso frecuente sea barato, y que todos los casos sean más complicados. Conviene tenerlo claro porque decide qué merece la pena cachear.

```
merece la pena cuando
  el mismo dato se pide muchas veces
  calcularlo o traerlo es caro
  y se tolera servirlo algo desfasado

no merece la pena cuando
  cada petición pide algo distinto
```

```
  el origen ya es barato
  o el dato no puede estar desfasado ni un segundo
```

Y la medida habitual —proporción de aciertos— no dice nada sola:

```
caché A   95 % de aciertos; el 5 % restante cuesta 8 ms
caché B   70 % de aciertos; el 30 % restante cuesta 900 ms
```

B protege muchísimo más que A. Lo que hay que medir es lo que se ahorra, no la proporción:

```
carga que llega al origen, con y sin caché
latencia del percentil 99, no la media
  → la media la domina el acierto; el usuario sufre el fallo
coste del fallo: latencia y carga en el origen
proporción de valores servidos ya desfasados
```

Y las capas, que es lo primero que hay que dibujar, porque el desfase se suma:

```
navegador          60 s
borde             300 s
en proceso         30 s
compartido        600 s
                  ██████
peor caso         990 s ≈ 16 min de desfase posible
```

Casi nadie hace esta suma, y es la explicación de la mitad de los «he cambiado el precio y sigue saliendo el viejo».

Y una distinción entre dos capas que se confunden:

```
EN PROCESO      rapidísimo, y cada instancia tiene el suyo
                → 40 instancias = 40 versiones distintas del dato
                → e invalidar exige avisar a las 40
COMPARTIDO      un salto de red, y una sola verdad
                → invalidar es un sitio
```

La combinación de los dos es potente y es también la que más desfases raros produce: un dato invalidado en el compartido sigue vivo treinta segundos en cuarenta procesos.

2. Patrones, y qué garantiza cada uno

```
CACHÉ APARTE  (el más común)
  leer:     mirar caché → si falla, leer origen → guardar
  escribir: escribir origen → invalidar caché
  + simple, y el caché puede caerse sin romper nada
  − hay una ventana entre escribir e invalidar
  − y cada sitio del código que escriba tiene que acordarse de invalidar

LECTURA A TRAVÉS
  el caché sabe cómo traer del origen
  + el código no gestiona fallos
  − más difícil de razonar cuando algo va mal

ESCRITURA A TRAVÉS
  escribir caché y origen a la vez, de forma síncrona
  + el caché nunca queda viejo
  − toda escritura paga la latencia de los dos

ESCRITURA DIFERIDA
  escribir en caché y volcar al origen después
  + escrituras rapidísimas
  − si el caché se pierde, se pierden datos → esto no es un caché,
    es un almacén primario con otro nombre
```

Y el orden importa más de lo que parece en el patrón más usado:

```
mal   invalidar el caché → escribir el origen
      entre las dos, otra petición lee el valor VIEJO y lo vuelve a cachear
      → el caché queda viejo indefinidamente

bien  escribir el origen → invalidar el caché
      la ventana de desfase existe y es corta y se cierra sola
```

Y aun así la segunda tiene una carrera conocida y difícil de evitar: una lectura que empezó antes de la escritura puede guardar su valor viejo después de la invalidación. Con desfase corto se acepta; cuando no se puede aceptar, la respuesta es el apartado siguiente.

3. Invalidar: la solución que casi nadie usa

Las tres formas de invalidar, con lo que cuesta cada una:

```
1. CADUCIDAD
   el valor vive N segundos y desaparece
   + no hay nada que mantener
   − el dato está viejo hasta N segundos, siempre
   − y todas las claves caducan a la vez si se llenaron a la vez → avalancha

2. INVALIDACIÓN EXPLÍCITA
   cada escritura borra las claves afectadas
   + el desfase es mínimo
   − hay que acordarse en CADA ruta de escritura, incluidas las de
     mantenimiento, las migraciones y los procesos por lotes
   − y la clave que se olvida no da ningún error: simplemente sirve mal

3. CLAVE VERSIONADA
   la clave incluye una versión que cambia al modificar el dato
   + invalidar es cambiar un número; el valor viejo deja de consultarse
   + no hay ruta de escritura que se pueda olvidar
   − ocupa memoria hasta que caduque, y hay que tener política de expulsión
```

La tercera resuelve el problema de la segunda de raíz y se usa poco:

```
en vez de     producto:1421
usar          producto:1421:v7

y la versión sale de algo que ya cambia con el dato:
  una columna de versión, la marca de última modificación,
  o la etiqueta que devuelve el propio origen
```

Y su variante para invalidar grupos enteros sin recorrer claves:

```
catalogo:v3:producto:1421
catalogo:v3:categoria:12

cambiar catalogo a v4 invalida el catálogo ENTERO de una vez
→ y no hay que borrar nada
```

Y una advertencia sobre las caducidades: conviene que no sean todas iguales. Si mil claves se llenaron en el mismo segundo y caducan a la vez, mil peticiones van al origen a la vez. Añadir una variación aleatoria lo resuelve:

```
ttl = base + random.randint(-base // 10, base // 10)
```

Y qué no cachear, que es una lista corta y importante:

```
decisiones de autorización, salvo con caducidad muy corta y revocación
  → un permiso retirado que sigue vivo 10 minutos es un problema de seguridad
datos de un usuario bajo una clave que no incluya al usuario
  → es la fuga entre usuarios más común que existe
lo que se escribe mucho más de lo que se lee
```

La segunda merece detenerse: si la clave es perfil en vez de perfil:cliente:1421, el segundo usuario ve los datos del primero. Ocurre sobre todo en cachés en proceso y en cachés de borde mal configuradas, donde la clave se deriva de la ruta y no de la sesión.

4. Los tres fallos, y la pregunta que lo decide todo

Avalancha. Una clave muy popular caduca y las peticiones que llegan en ese instante fallan todas a la vez:

```
clave con 4.000 peticiones/s
coste de recalcularla                800 ms
peticiones que llegan durante ese cálculo    3.200
→ 3.200 consultas simultáneas al origen por el mismo dato
```

Las tres defensas, que se combinan:

```
cerrojo    solo una petición recalcula; las demás esperan o sirven el viejo
recálculo anticipado   se refresca ANTES de caducar, en segundo plano
variación de caducidad  evita que muchas claves caduquen juntas
```

La segunda es la mejor para las claves muy populares: el valor nunca llega a caducar porque siempre se renueva antes.

Penetración. Peticiones por claves que no existen: nunca aciertan y siempre llegan al origen.

```
peticiones a /producto/999999999  → no existe → no se cachea → al origen
→ repetido, es una forma trivial de saturar la base de datos
```

La defensa es cachear también el vacío, con caducidad corta:

```
guardar «no existe» durante 30 s
y validar el formato antes de consultar
```

Arranque en frío. Y aquí está la pregunta que decide la naturaleza del caché:

```
si el caché se vacía ahora mismo, ¿aguanta el origen?

  sí   → es una optimización; puede caerse y el sistema se degrada
  no   → NO es un caché: es una dependencia con requisito de disponibilidad
```

Y hay que responderla con números, no con intuición:

```
tráfico total                        12.000 peticiones/s
proporción de aciertos                    94 %
carga que llega al origen                720 peticiones/s
capacidad del origen                   1.500 peticiones/s
carga con el caché vacío              12.000 peticiones/s   ← 8× la capacidad
```

Con esos números, el caché es portante, y eso cambia todo lo que hay que hacer con él:

```
réplica y conmutación, como en la clase 109
plan de recalentamiento tras un reinicio
limitación de caudal hacia el origen, para que el frío no lo tumbe
y un ensayo: vaciar el caché en preproducción y medir
```

Y una salida intermedia muy útil: servir el valor viejo cuando el origen falla. Si el origen no responde, devolver lo caducado es casi siempre mejor que devolver un error.

```
valor fresco       se sirve
valor caducado y origen sano       se recalcula
valor caducado y origen caído      se sirve el viejo, y se registra
```

Y la lista de comprobación de la clase:

- está medido el coste del fallo, no solo la proporción de aciertos
- están dibujadas las capas y sumada la ventana de desfase total
- se escribe el origen antes de invalidar, nunca al revés
- la invalidación usa claves versionadas en vez de borrado por ruta
- las caducidades llevan variación aleatoria
- las claves muy populares se refrescan antes de caducar
- se cachea el vacío para evitar penetración
- ninguna clave de datos de usuario carece del identificador de usuario
- las decisiones de autorización no se cachean sin revocación
- está respondido con números si el origen aguanta el caché vacío
- si es portante: réplica, limitación hacia el origen y ensayo de vaciado

Y el cierre que enlaza con la clase siguiente: hasta aquí, los datos que sirven al producto. Los que no caben en ninguno de estos almacenes —los históricos, los grandes, los que se consultan de formas que nadie previó— necesitan otro sitio, y es la materia de la clase 112.

Ejemplo trabajado

CloudShop pone caché delante del catálogo para resolver el producto popular que la clase 110 dejó abierto. En nueve meses ocurren los tres fallos del apartado cuarto, más uno que no está en la lista y es el más grave.

Punto de partida.

```
lecturas del catálogo                  12.000 / s
capacidad del almacén                   1.500 / s
producto en portada                     4.100 / s sobre un solo elemento
latencia p99                               34 ms
```

Caché compartido, patrón de caché aparte, caducidad de 300 s.

```
aciertos                                    —            94 %
carga al origen                        12.000/s        720/s
latencia p99                              34 ms         6 ms
coste mensual del almacén                2.900 €        410 €
```

Fallo 1: avalancha, a los once días.

```
03:14  caducan a la vez las claves llenadas durante el despliegue de las 22:14
```

```
03:14  1.840 claves caducadas en el mismo segundo
03:14  la carga al origen pasa de 720/s a 9.100/s
03:15  el almacén limita; errores en el 41 % de las peticiones
03:19  se estabiliza al recachearse
```

La causa es la del apartado tercero: el caché se llenó de golpe tras un despliegue, así que caducó de golpe. Las tres defensas y su efecto:

```
pico de carga al origen                 9.100/s    2.300/s        810/s
claves caducadas en el mismo segundo     1.840        41            —
errores                                   41 %       0 %           0 %
```

El recálculo anticipado se aplicó solo a las 500 claves más pedidas: el 0,04 % de las claves recibía el 68 % del tráfico.

Fallo 2: penetración, y es deliberada.

```
14:20  llegan 3.400 peticiones/s a rutas de producto con identificadores
       inventados
14:20  ninguna acierta; todas llegan al origen
14:22  el origen satura; el catálogo entero deja de responder
```

No hacía falta ninguna sofisticación: bastaba pedir productos que no existen.

```
cachear el vacío                          no             sí, 30 s
validación de formato antes de consultar  no             sí
carga al origen con el mismo ataque    3.400/s          ~40/s
```

Fallo 3: el caché se reinicia.

Una actualización de mantenimiento vació el caché a las 11:40 de un martes:

```
carga al origen tras el vaciado          12.000/s
capacidad del origen                      1.500/s
tiempo de caída total                    9 minutos
tiempo hasta recuperar el 90 % de aciertos   14 min
```

La pregunta del apartado cuarto se respondió con números después del incidente, cuando debía haberse respondido antes:

```
carga con el caché vacío / capacidad del origen = 8×
→ el caché es portante
```

Y con esa respuesta, el tratamiento cambió por completo:

```
réplica del caché                              no             sí
conmutación probada                            no          trimestral
limitación de caudal hacia el origen           no        1.200/s, con cola
servir valor caducado si el origen falla       no             sí
recalentamiento tras reinicio                  no      las 500 claves top
ensayo de vaciado en preproducción             no          trimestral

caída al repetir el ensayo                   9 min          0 min
                                                        (degradación de 90 s)
```

El cuarto fallo, que no está en la lista y es el peor.

Un caché en proceso se añadió para las preferencias del usuario, con esta clave:

```
clave = "preferencias"
```

Sin el identificador de usuario.

```
usuarios que vieron preferencias de otra persona    no determinado
detectado por                          un cliente que llamó a soporte
tiempo hasta corregir                                2 h 10
tiempo que llevaba desplegado                        6 días
datos expuestos      dirección de envío y últimos 4 dígitos de la tarjeta
```

Y lo que se hizo para que no vuelva a pasar, que es más útil que la corrección:

```
regla en la revisión: toda clave de caché de datos de usuario debe
  contener el identificador del sujeto
prueba automática: dos sesiones distintas piden lo mismo y se compara
  que las respuestas difieren
comprobación estática de las claves construidas con literales
```

La prueba automática de la segunda línea detectó otros dos casos en el resto del sistema, ambos en cachés de borde donde la clave se derivaba de la ruta.

La ventana de desfase, que nadie había sumado.

Tras una queja recurrente de comercio —«cambio un precio y tarda en verse»— se dibujaron las capas:

```
navegador                    60 s
borde                       600 s
en proceso                   30 s
compartido                  300 s
                           ■■■■■■
peor caso                   990 s ≈ 16 min
```

Y la corrección no fue bajar caducidades, que habría multiplicado la carga, sino claves versionadas derivadas de la marca de modificación del producto:

```
desfase peor caso                        16 min            ~2 s
caducidad del borde                      600 s             86.400 s
carga al origen                          720/s             310/s
rutas de escritura que hay que recordar
invalidar                                   7                 0
```

La penúltima fila es lo contraintuitivo: al versionar las claves se pudo subir la caducidad, porque el valor viejo ya no molesta a nadie. Menos desfase y menos carga a la vez.

A los nueve meses.

```
carga al origen                            12.000/s          310/s
latencia p99                                 34 ms           5 ms
desfase peor caso                            16 min           ~2 s
rutas de escritura que invalidan a mano           7              0
avalanchas                                 1 / mes            0
caída por caché vacío                         9 min      0 (degradación 90 s)
fugas entre usuarios                               3              0
coste mensual del almacén                     2.900 €          380 €
caché clasificado como portante                   no             sí
```

La lección que esta clase traslada a la parte 09: el caché resolvió el elemento popular en una tarde, y los nueve meses siguientes fueron descubrir que era una dependencia crítica sin ninguno de los cuidados de una. La pregunta «¿aguanta el origen con el caché vacío?» costó nueve minutos de caída por no haberla respondido antes, y es la única de esta clase que no admite una respuesta aproximada.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/111-cache-invalidacion-ttl-y-consistencia/lab.py
```

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

1. Lee exercise.steps y formula una predicción antes de ejecutar.
2. Ejecuta la práctica y verifica todos los elementos de checks.
3. Provoca el caso de negativetest y explica la señal observada.
4. Materializa estrategia-cache 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 modelo de datos ligado a patrones de acceso. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye estrategia-cache 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
Picos de carga al origen a horas fijas	Muchas claves se llenaron a la vez y caducan a la vez	Añade variación aleatoria a la caducidad y refresca en segundo plano las claves más pedidas antes de que caduquen.
El origen se satura con peticiones por identificadores inexistentes	Penetración: lo que no existe nunca se cachea y siempre llega al origen	Cachea el resultado vacío con caducidad corta y valida el formato antes de consultar.
Un reinicio del caché tumba el sistema	El origen no tiene capacidad para el tráfico completo: el caché es portante	Calcula la relación entre tráfico total y capacidad del origen; si es mayor que uno, trata el caché como dependencia: réplica, limitación de caudal, recalentamiento y ensayo de vaciado.
Un usuario ve datos de otro	La clave de caché no incluye el identificador del sujeto	Exige el identificador en toda clave de datos de usuario y añade una prueba que compare respuestas de dos sesiones distintas.
Se cambia un dato y sigue viéndose el viejo mucho tiempo después	Hay varias capas de caché y sus ventanas de desfase se suman	Dibuja las capas, suma las caducidades y sustituye la caducidad por claves versionadas donde el desfase importe.
Una ruta de escritura olvidó invalidar y nadie se entera	La invalidación explícita depende de acordarse en cada sitio, y olvidarla no produce error	Usa claves versionadas: el valor viejo deja de consultarse sin que nadie tenga que borrarlo.

■ Seguridad, ética y costo

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

■ Preguntas de comprobación

1. ¿Por qué la proporción de aciertos no basta para evaluar un caché?
2. ¿Por qué se escribe el origen antes de invalidar y no al revés?
3. ¿Qué ventaja tiene una clave versionada frente a la invalidación explícita?
4. ¿Qué es una avalancha y con qué tres defensas se previene?
5. ¿Cómo se decide si un caché es una optimización o una dependencia?

Referencias

- Redis (2025). Key eviction and expiration — políticas de expulsión, caducidad y su efecto en la carga. <https://redis.io/docs/latest/develop/reference/eviction/>

- Nygard, M. (2018). Release It!, cap. 5 — caché, avalancha y degradación controlada. <https://pragprog.com/titles/mnee2/release-it-second-edition/>
- Facebook Engineering (2013). Scaling Memcache at Facebook — cerrojo de recálculo y avalancha en producción. <https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170update.pdf>
- MDN (2025). HTTP caching — capas de caché, validadores y control de desfase. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching>
- AWS (2025). ElastiCache: caching strategies and best practices — caché aparte, escritura a través y recalentamiento. <https://docs.aws.amazon.com/AmazonElastiCache/latest/dg/Strategies.html>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 09 en PDF · Manual integral

Anterior	Índice	Siguiente
← 110 · NoSQL: clave-valor, documento, columna y grafo	Parte 09 · Programa	112 · Object storage, data lake y formatos columnares →

Evaluación — 111 Caché, invalidación, TTL y consistencia

Preguntas

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

Reto verificable

Entrega estrategia-cache 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.

112 — Object storage, data lake y formatos columnares

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 4 Laboratorio: data · Estado: EXECUTABLECORE

Propósito

Construir el sitio donde van las consultas que la clase 110 dejó sin casa: las que nadie previó, las históricas y las que recorren millones de filas. La clase explica por qué el almacenamiento de objetos no es un sistema de ficheros y qué se rompe al tratarlo como tal, desarrolla el formato columnar con la aritmética de lo que ahorra, y se detiene en los dos errores que arruinan un lago de datos: millones de ficheros diminutos y una partición que no poda nada.

Resultados de aprendizaje

Al finalizar podrás:

1. Enumerar lo que el almacenamiento de objetos no hace y qué patrones se rompen por suponerlo.
2. Proteger los datos de un borrado propio, que la durabilidad no cubre.
3. Calcular lo que ahorra un formato columnar frente a uno por filas.
4. Particionar de modo que las consultas poden, sin generar ficheros diminutos.
5. Justificar cuándo hace falta un formato de tabla por encima de los ficheros.

Conceptos centrales

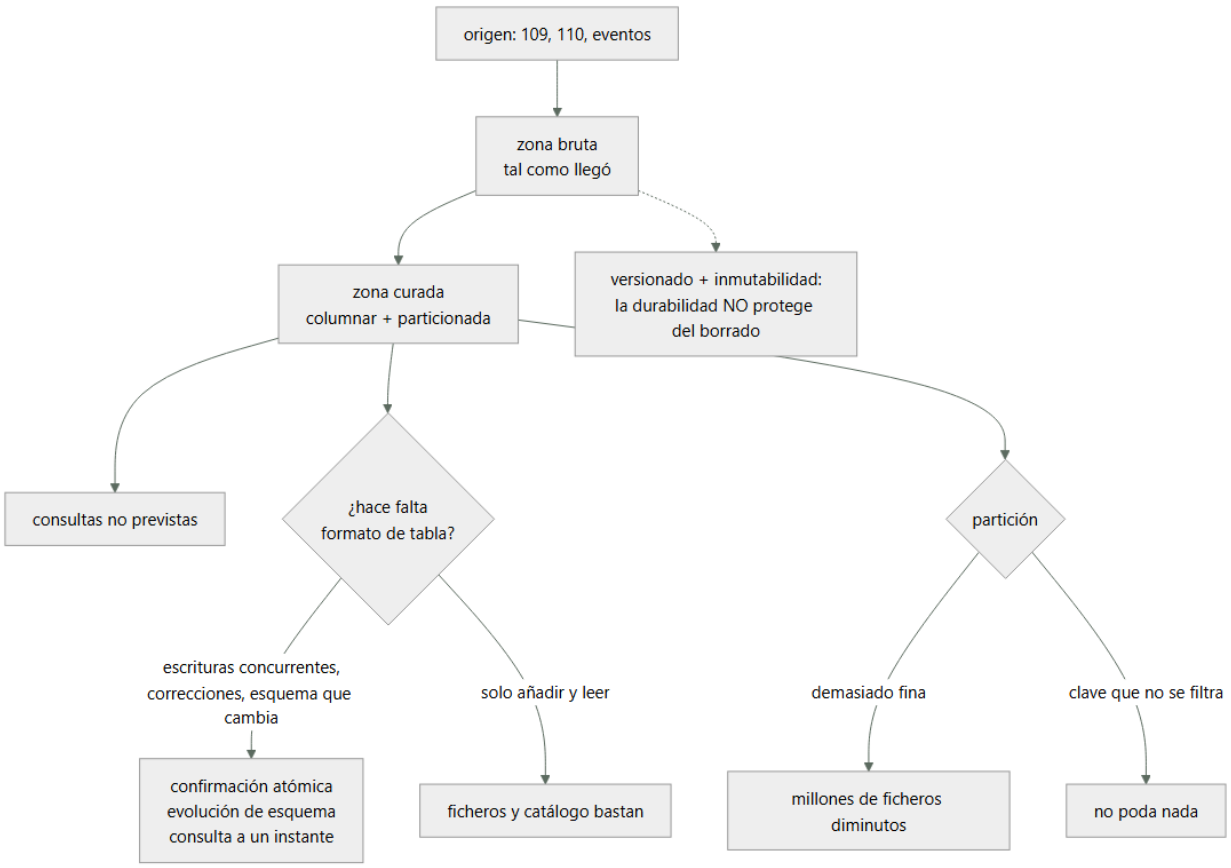
Concepto	Comprensión verificable
espacio de nombres plano	No hay directorios: hay claves con barras. Renombrar una carpeta es copiar y borrar cada objeto, uno a uno.
durabilidad frente a borrado	La durabilidad protege del fallo del soporte. No protege de que alguien —o algo— borre. Son problemas distintos con soluciones distintas.
formato columnar	Los valores de una misma columna se guardan juntos. Permite leer solo las columnas necesarias y comprimir mucho mejor.
poda por partición	Descartar ficheros enteros sin abrirlos porque la ruta indica que no contienen lo buscado. Es el mayor ahorro de un lago.
problema del fichero pequeño	Muchos objetos diminutos hacen que el tiempo se vaya en abrir ficheros en vez de en leer datos. Es el fallo más común de un lago.
formato de tabla	Capa de metadatos sobre los ficheros que aporta confirmación atómica, evolución de esquema y consulta a un instante pasado.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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

■ Flujo de razonamiento



Desarrollo

1. No es un sistema de ficheros

El almacenamiento de objetos parece un disco con carpetas y no lo es. Las diferencias no son cosméticas: cada una rompe un patrón habitual.

```
no hay directorios          las barras son parte de la clave
                            «renombrar una carpeta» = copiar y borrar N objetos

no se modifica una parte    para cambiar un byte se reescribe el objeto entero

no se añade al final        no hay escritura incremental sobre un objeto
                            → un registro que crece necesita objetos nuevos

listar es una operación     y cuesta; listar un prefijo con millones de
                            objetos es lento y se factura

la operación se factura     no solo los bytes: también cada petición
```

Y de esas cinco salen los errores típicos:

```
usarlo como sistema de ficheros con un adaptador
  → un renombrado de carpeta se convierte en horas
escribir un fichero por evento
  → millones de objetos y una factura dominada por peticiones
listar para saber qué hay
  → el listado es el cuello de botella, no la lectura
```

La tercera se resuelve con la idea que sostiene todo lo demás: no listar, sino tener un catálogo. Un fichero de manifiesto —o un formato de tabla, apartado cuarto— dice qué ficheros componen el conjunto, y la consulta no necesita preguntar al almacenamiento qué existe.

Y una propiedad que sí ha mejorado y conviene no arrastrar como mito: la escritura de un objeto nuevo es visible inmediatamente en los tres grandes proveedores desde hace años. Lo que sigue sin ser atómico es todo lo que abarque varios objetos:

```
escribir un objeto            atómico
escribir 200 objetos          NO es atómico
  → un lector puede ver 137 de los 200
```

Y ese es exactamente el problema que motiva el apartado cuarto.

2. Durabilidad no es protección

Los proveedores anuncian durabilidades de once nueves. Es cierto y responde a una sola pregunta: ¿se puede perder por fallo del soporte? No responde a ninguna de estas otras:

```
¿y si alguien lo borra?                    la durabilidad no interviene
¿y si un proceso lo sobrescribe con vacío?  tampoco
¿y si se cifra desde fuera?                tampoco
¿y si se borra el contenedor entero?       tampoco
```

Las cuatro defensas, y son distintas entre sí:

```
VERSIONADO           cada sobrescritura crea una versión; borrar es una marca
                     → recuperar es quitar la marca
                     → y hay que pagar el almacenamiento de las versiones

INMUTABILIDAD        el objeto no se puede borrar ni modificar hasta una fecha
                     → ni siquiera por quien administra
                     → es lo único que resiste una credencial comprometida

COPIA EN OTRA CUENTA copia a un destino con credenciales distintas
                     → protege del error y del compromiso de la cuenta origen

CICLO DE VIDA        mueve y caduca automáticamente
                     → y también puede borrar lo que no debía: revísalo
```

Y la segunda merece detenerse porque es la que cierra el caso que ninguna otra cierra. Si la credencial que administra el contenedor está comprometida, versionado y ciclo de vida no protegen: quien administra puede borrar versiones. La inmutabilidad con retención sí.

Y la comprobación que no puede faltar, la misma de la clase 088:

```
restaurar de verdad, con cronómetro, cada trimestre
→ una copia que nadie ha restaurado no es una copia
```

Las clases de almacenamiento, con la aritmética que se olvida:

```
frecuente                    alto           gratis           ninguna
esporádico                   medio          por GB          30 días
archivo                      bajo           por GB + espera 90-180 días
```

Y los tres errores que salen caros:

```
1. mover a archivo datos que se consultan
   → el coste de recuperación supera el ahorro

2. mover objetos pequeños
   → hay un mínimo facturable por objeto (típicamente 128 KB)
   → 40 millones de objetos de 4 KB se facturan como si fueran de 128 KB

3. borrar antes de la permanencia mínima
   → se paga igual el periodo completo
```

El segundo es el más frecuente y el más contraintuitivo: archivar muchos objetos pequeños puede costar más que dejarlos donde estaban. La solución es agrupar antes de archivar, que es la misma medicina del apartado siguiente.

3. Columnar: la aritmética

Un formato por filas guarda cada registro completo y seguido. Uno columnar guarda juntos todos los valores de una misma columna. Para analítica, la diferencia es enorme y se puede calcular.

```
tabla de eventos: 80 columnas, 2.000 millones de filas
consulta típica: 3 columnas, filtrando por fecha

POR FILAS (JSON o CSV)
  hay que leer las 80 columnas para quedarse con 3
  sin compresión efectiva: los tipos se mezclan
  → 1,4 TB leídos
```

```
COLUMNAR
  se leen 3 columnas de 80                          → ×0,04
  cada columna comprime bien porque los valores se parecen  → ×0,2
  estadísticas por bloque descartan bloques sin abrirlos    → ×0,3
  → ~3,4 GB leídos
```

Y como se factura por datos leídos, esa reducción es también la factura.

Los tres mecanismos, por separado:

```
PROYECCIÓN     leer solo las columnas pedidas
               → el ahorro es proporcional a columnas usadas / totales

COMPRESIÓN     valores del mismo tipo y parecidos juntos
               → un campo de estado con 5 valores distintos comprime
                 casi a nada; el mismo campo en JSON repite la etiqueta
                 en cada fila

ESTADÍSTICAS   cada bloque guarda mínimo y máximo por columna
               → si el filtro pide fecha > X y el máximo del bloque es
                 menor, el bloque no se abre
```

Y el tercero solo funciona si los datos están ordenados por la columna que se filtra. Escribir desordenado deja los mínimos y máximos tan amplios que ningún bloque se descarta:

```
ordenado por fecha      bloques descartados: 97 %
sin ordenar             bloques descartados:  4 %
```

Es el ajuste más rentable de un lago y el que menos se hace.

La partición, que actúa antes que todo lo anterior porque descarta ficheros sin abrirlos:

```
eventos/fecha=2026-07-31/pais=ES/parte-0001.parquet
```

Y sus dos fallos, opuestos:

```
DEMASIADO FINA    partición por hora y por cliente
                  → 8.760 × 40.000 = millones de ficheros diminutos
                  → el tiempo se va en abrir ficheros

QUE NO SE FILTRA  partición por identificador de pedido
                  → ninguna consulta filtra por eso → no poda nada
```

Y la regla de tamaño que resuelve el primero:

```
fichero objetivo: 128 MB a 1 GB
si las particiones quedan por debajo, particiona menos
y compacta periódicamente lo que llegue en trozos pequeños
```

Y la tensión que hay que resolver conscientemente: la ingesta quiere escribir seguido y pequeño; la consulta quiere ficheros grandes. La compactación es el proceso que reconcilia las dos, y es un trabajo programado que hay que operar, no algo que ocurra solo.

4. Cuándo hacen falta formatos de tabla

Con ficheros columnares particionados y un catálogo se puede llegar muy lejos. Lo que no se puede hacer es lo que este apartado enumera, y de ahí salen los formatos de tabla.

```
escribir 200 ficheros no es atómico
  → un lector ve el conjunto a medias

corregir o borrar filas concretas
  → obliga a reescribir particiones enteras a mano
  → y hay requisitos legales que exigen borrar filas concretas

cambiar el esquema
  → añadir una columna es fácil; renombrar o cambiar tipo, no

saber qué había ayer
  → si se sobrescribió, no hay forma

varios procesos escribiendo a la vez
  → se pisan
```

Un formato de tabla añade una capa de metadatos que resuelve las cinco:

```
CONFIRMACIÓN ATÓMICA   los ficheros nuevos no existen para el lector hasta
                       que se confirma una instantánea nueva
EVOLUCIÓN DE ESQUEMA   las columnas tienen identificador, no posición
                       → renombrar no rompe
BORRADO Y ACTUALIZACIÓN por fila, con reescritura acotada o marcas de borrado
CONSULTA A UN INSTANTE  cada confirmación es una instantánea consultable
CONCURRENCIA            control optimista: si dos confirman a la vez, una repite
METADATOS EN FICHEROS   no hay que listar el almacenamiento para saber qué hay
```

La última resuelve el problema del primer apartado y es, en la práctica, la que más se nota en consultas grandes.

Y el criterio para decidir:

```
solo se añade y solo se lee, y el esquema no cambia
  → ficheros columnares y catálogo bastan

hay correcciones, borrados por requisito, esquema vivo o
varios procesos escribiendo
  → formato de tabla
```

Y las dos advertencias que acompañan a los formatos de tabla:

```
las instantáneas se acumulan y ocupan
  → hay que caducar instantáneas y limpiar ficheros huérfanos
  → y ese trabajo, si no se ejecuta, no da ningún error (ley 13)

sigue haciendo falta compactar
  → el formato de tabla no elimina el problema del fichero pequeño
```

Y la organización del lago, que conviene fijar desde el primer día:

```
ZONA BRUTA      tal como llegó, sin transformar, inmutable
                → permite reprocesar cuando se descubre un error
ZONA CURADA     columnar, particionada, con esquema y tipos
ZONA DE CONSUMO agregados listos para cada uso
```

Y la razón de la primera zona, que se discute mucho: cuando se descubre un fallo en la transformación, lo único que permite arreglar el histórico es tener el dato original.

Y la lista de comprobación de la clase:

- nadie trata el almacenamiento como sistema de ficheros
- no hay listados de prefijos enormes en el camino de una consulta
- el versionado está activo y hay inmutabilidad donde importa
- existe copia en otra cuenta con credenciales distintas
- la restauración se ha probado con cronómetro este trimestre
- el ciclo de vida no archiva objetos pequeños sin agruparlos antes
- los datos de consulta están en formato columnar
- están ordenados por la columna que más se filtra
- las particiones se corresponden con los filtros reales
- el tamaño de fichero está entre 128 MB y 1 GB, y hay compactación
- si hay formato de tabla, la caducidad de instantáneas se ejecuta
- la zona bruta se conserva para poder reprocesar

Y el cierre que enlaza con la clase siguiente: los datos llegan a este lago desde algún sitio, y ese transporte tiene sus propias garantías. Qué se garantiza al entregar un mensaje, qué pasa cuando falla y dónde acaba lo que nadie pudo procesar es la materia de la clase 113.

Ejemplo trabajado

CloudShop construye el lago para recuperar los informes que perdió al cambiar la clave de partición en la clase 110. El primer intento funciona y cuesta veinte veces más de lo previsto; el ejercicio es corregirlo paso a paso midiendo cada cambio.

Primer intento: un fichero JSON por evento.

```
eventos al día                          41 millones
un objeto por evento
tamaño medio del objeto                      1,2 KB
objetos al mes                         1.230 millones

coste de almacenamiento                      58 €/mes
coste de peticiones de escritura            621 €/mes
consulta «pedidos por país del último mes»
```

```
datos leídos                                  1,4 TB
duración                                      47 min
coste por ejecución                            7,0 €
ejecuciones al día                                30
coste de consulta                          6.300 €/mes
```

Seis mil trescientos euros al mes en consultas. Y el diagnóstico es el del apartado tercero: se leen 80 columnas para usar 3, sin compresión y sin poder descartar nada.

Cambio 1: agrupar y pasar a columnar.

```
objetos al mes                       1.230 millones          4.100
tamaño medio del objeto                    1,2 KB          340 MB
datos leídos por consulta                  1,4 TB          38 GB
duración de la consulta                    47 min         2 min 10 s
coste de peticiones                       621 €/mes        2 €/mes
coste de consulta                       6.300 €/mes      170 €/mes
```

Cambio 2: particionar. Y aquí se comete el error de partición demasiado fina.

Primera partición: por hora y por país.

```
particiones al mes                  720 × 190 = 136.800
ficheros por partición                            3
tamaño medio de fichero                       2,4 MB
ficheros totales                            410.400

duración de la consulta                    4 min 40 s   ← EMPEORÓ
razón     el tiempo se va en abrir 410.400 ficheros
```

Particionar empeoró la consulta. Segunda partición, por día y país, con compactación:

```
particiones al mes                  136.800             5.700
ficheros totales                    410.400             6.900
tamaño medio de fichero              2,4 MB             210 MB
duración de la consulta             4 min 40 s         38 s
datos leídos                          38 GB             4,1 GB
```

Cambio 3: ordenar por la columna que se filtra.

```
bloques descartados por estadísticas          4 %             97 %
datos leídos                                4,1 GB           210 MB
duración                                      38 s             6 s
coste de consulta                          170 €/mes        11 €/mes
```

Seis segundos frente a los cuarenta y siete minutos iniciales, y once euros frente a seis mil trescientos.

El error del ciclo de vida, que costó dinero en la dirección contraria.

Antes de agrupar, alguien añadió una regla para archivar los objetos de más de 30 días:

```
objetos archivados                          890 millones
tamaño medio real                                1,2 KB
mínimo facturable por objeto                     128 KB
tamaño facturado                    890 M × 128 KB = 114 TB
tamaño real                                      1,07 TB

coste esperado del archivo                       4 €/mes
coste real                                     430 €/mes
```

Y al intentar deshacerlo apareció la segunda parte del apartado segundo:

```
permanencia mínima de la clase de archivo         90 días
borrar antes                          se factura igual el periodo completo
coste de la corrección                          1.290 € una vez
```

La regla se rehízo para archivar después de agrupar, no antes.

Cuándo hizo falta el formato de tabla.

Durante siete meses bastaron ficheros y catálogo. Tres cosas lo cambiaron:

1. una petición de borrado de datos de un cliente concreto
   sin formato de tabla: localizar y reescribir 340 particiones
   con formato de tabla: una sentencia de borrado, 11 min

2. un error en la transformación descubierto 4 meses después

```
   se reprocesó desde la ZONA BRUTA, que existía por decisión previa
   → sin ella, el histórico habría quedado mal para siempre

3. dos procesos escribiendo la misma tabla se pisaron
   consulta que leyó el conjunto a medias: 1 informe erróneo publicado
```

Tras adoptarlo:

```
borrado de filas concretas             reescribir       sentencia, 11 min
                                       340 particiones
lectura durante la escritura           parcial          instantánea consistente
cambio de nombre de columna            rompe            no rompe
consulta del estado de ayer            imposible        sí
listado del almacenamiento por consulta   sí            no, metadatos
```

Y el trabajo nuevo que trajo, y que la ley 13 hizo invisible durante dos meses:

```
instantáneas acumuladas antes de darse cuenta        4.100
espacio ocupado por instantáneas antiguas             2,1 TB
coste asociado                                       48 €/mes
caducidad de instantáneas                    no se ejecutaba
```

Nadie dio un error: el trabajo de limpieza simplemente no estaba programado.

Estado final.

```
objetos al mes                          1.230 millones      4.100
datos leídos por consulta                      1,4 TB       210 MB
duración de la consulta                        47 min         6 s
coste mensual de consultas                  6.300 €          11 €
coste mensual de peticiones                   621 €           2 €
coste de almacenamiento                        58 €          64 €
coste del error de archivo                      —       1.290 € una vez
versionado e inmutabilidad                     no            sí
copia en otra cuenta                           no            sí
restauración probada                        nunca        trimestral
compactación y caducidad programadas           no            sí
```

La lección que esta clase traslada a la parte 09: el lago pasó de 6.300 € a 11 € al mes sin cambiar de tecnología ni de proveedor. Los tres cambios que lo consiguieron —agrupar, particionar por lo que se filtra y ordenar por la columna del filtro— son decisiones de disposición de datos, exactamente las mismas que la clase 110 mostró en el almacén operativo. Y el error más caro fue el opuesto al que se teme: una optimización de coste aplicada a millones de objetos diminutos multiplicó la factura por cien, porque el mínimo facturable por objeto no se había mirado.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/112-object-storage-data-lake-y-formatos-columnares/lab.py
```

El laboratorio selecciona el motor de práctica data 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 diseno-data-lake 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 modelo de datos ligado a patrones de acceso. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

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

■ Criterio de aceptación

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

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
La factura está dominada por peticiones y no por almacenamiento	Se escribe un objeto por evento	Agrupa en ficheros de 128 MB a 1 GB antes de escribir, y compacta lo que llegue pequeño.
Particionar hizo la consulta más lenta	Partición demasiado fina: el tiempo se va en abrir ficheros diminutos	Reduce la granularidad hasta que cada partición tenga ficheros de tamaño objetivo, y compacta periódicamente.
La consulta lee casi todo aunque filtre por fecha	Los datos no están ordenados por la columna del filtro, así que las estadísticas por bloque no descartan nada	Ordena al escribir por la columna que más se filtra y verifica la proporción de bloques descartados.
Archivar datos antiguos aumentó el coste	Hay un mínimo facturable por objeto y se archivaron millones de objetos diminutos	Agrupa antes de archivar y comprueba la permanencia mínima antes de borrar.
Un borrado accidental no se puede deshacer pese a la durabilidad anunciada	La durabilidad protege del fallo del soporte, no del borrado	Versionado, inmutabilidad con retención donde importe y copia en otra cuenta con credenciales distintas.
Una consulta devuelve un conjunto a medias	Escribir varios objetos no es atómico y no hay capa de metadatos	Usa un formato de tabla con confirmación atómica, y programa la caducidad de instantáneas, que no falla si no se ejecuta.

■ 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 cinco cosas no hace el almacenamiento de objetos y qué patrón rompe cada una?
2. ¿Por qué once nueves de durabilidad no protegen de un borrado, y qué sí lo hace?
3. ¿Qué tres mecanismos hacen que un formato columnar lea menos, y cuál depende del orden de escritura?
4. ¿Cuáles son los dos fallos opuestos de la partición?
5. ¿Qué cinco cosas resuelve un formato de tabla que los ficheros sueltos no resuelven?

Referencias

- Apache Parquet (2025). File format specification — grupos de filas, estadísticas y codificaciones por columna. <https://parquet.apache.org/docs/file-format/>
- Apache Iceberg (2025). Table specification — instantáneas, confirmación atómica y evolución de esquema. <https://iceberg.apache.org/spec/>
- AWS (2025). S3 storage classes and lifecycle considerations — mínimos facturables, permanencia y coste de recuperación. <https://docs.aws.amazon.com/AmazonS3/latest/userguide/storage-class-intro.html>
- Google Cloud (2025). Cloud Storage: object versioning and retention policies — versionado, retención e inmutabilidad. <https://cloud.google.com/storage/docs/object-versioning>
- Azure (2025). Data Lake Storage best practices — tamaño de fichero, particionado y organización por zonas. <https://learn.microsoft.com/azure/storage/blobs/data-lake-storage-best-practices>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 09 en PDF · Manual integral

Anterior	Índice	Siguiente
← 111 · Caché, invalidación, TTL y consistencia	Parte 09 · Programa	113 · Colas, entrega, reintentos y dead-letter queues →

Evaluación — 112 Object storage, data lake y formatos columnares

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 diseno-data-lake 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.

113 — Colas, entrega, reintentos y dead-letter queues

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 4 Laboratorio: messaging · Estado: EXECUTABLECORE

Propósito

Separar en el tiempo quien produce trabajo de quien lo hace, y asumir con precisión lo que eso obliga a aceptar. La clase parte de una afirmación que conviene creerse antes de diseñar nada: la entrega exactamente una vez no existe sobre una red, y lo que se vende con ese nombre es otra cosa. De ahí salen las tres decisiones de la clase: cuánto tiempo se oculta un mensaje mientras se procesa, cuántas veces se reintenta y qué se hace con lo que nadie pudo procesar.

Resultados de aprendizaje

Al finalizar podrás:

1. Explicar qué garantiza cada semántica de entrega y por qué se diseña para al menos una vez.
2. Ajustar el tiempo de invisibilidad al tiempo real de proceso.
3. Configurar reintentos que no amplifiquen una caída.
4. Operar la cola de mensajes fallidos, que sin vigilancia no sirve de nada.
5. Decidir si hace falta orden y qué cuesta imponerlo.

Conceptos centrales

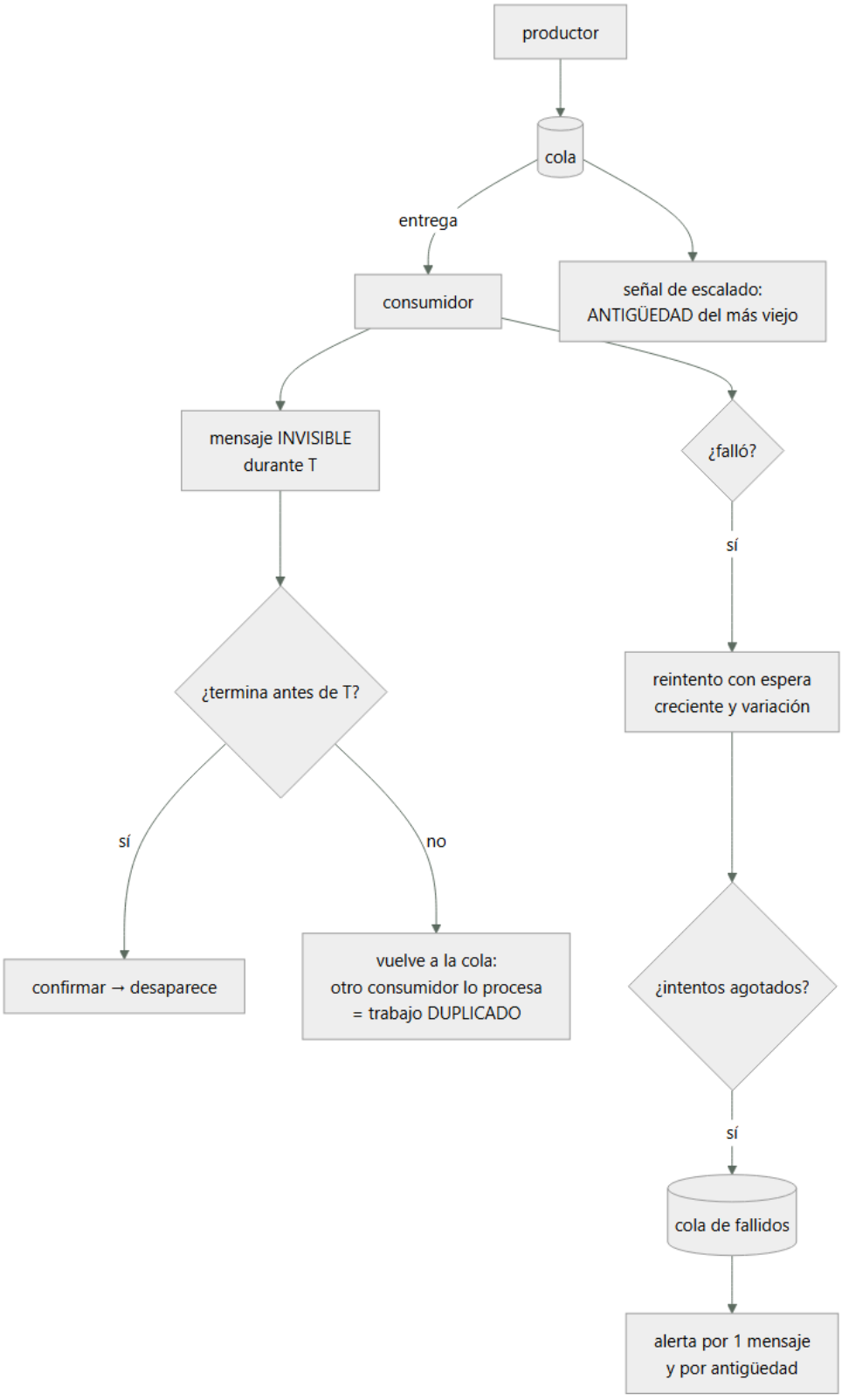
Concepto	Comprensión verificable
al menos una vez	Cada mensaje se entrega una o más veces. Es lo que ofrecen de verdad los sistemas de cola, y obliga a que el consumidor tolere repeticiones.
tiempo de invisibilidad	Periodo durante el que un mensaje entregado no se ofrece a otros consumidores. Si el proceso dura más, otro consumidor lo recoge y se duplica el trabajo.
confirmación	Señal del consumidor de que el mensaje se procesó. Antes de ella, el mensaje sigue vivo; después, desaparece.
mensaje venenoso	Mensaje que falla siempre. Sin límite de intentos, ocupa capacidad indefinidamente y puede bloquear a los que van detrás.
cola de fallidos	Destino de lo que agotó sus intentos. Es un sitio para inspeccionar y reprocesar, no un cementerio.
antigüedad del más viejo	Segundos que lleva esperando el mensaje más antiguo. Es la señal correcta del estado de una cola; la profundidad sola engaña.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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 de verdad se garantiza

Las tres semánticas, con lo que significan de verdad:

```
COMO MUCHO UNA VEZ
  se entrega y no se reintenta; si algo falla, el mensaje se pierde
  útil solo para datos que se pueden perder: telemetría de grano fino
```

```
AL MENOS UNA VEZ
  se reintenta hasta que se confirma
  → puede entregarse dos veces, y de hecho ocurre
  es lo que ofrecen los sistemas de cola de verdad

EXACTAMENTE UNA VEZ
  no existe como propiedad de la ENTREGA sobre una red
```

Y conviene entender por qué la tercera no existe, porque explica todo lo demás:

```
el consumidor procesa el mensaje y va a confirmar
la confirmación se pierde por la red

el emisor no puede distinguir:
  «no llegó y hay que reenviar»    de    «llegó y se perdió la respuesta»

cualquiera de las dos decisiones se equivoca en un caso
```

Lo que los proveedores llaman «exactamente una vez» es una de estas dos cosas, y las dos son útiles siempre que se sepa qué son:

```
deduplicación por ventana   se descarta un mensaje repetido con el mismo
                            identificador dentro de N minutos
                            → fuera de la ventana, se vuelve a entregar

proceso transaccional       leer, procesar y confirmar dentro de una
                            transacción del mismo sistema
                            → solo vale si el efecto está en ese sistema
```

Y de ahí la regla que gobierna esta clase entera y que la clase 116 desarrollará:

```
se diseña para AL MENOS UNA VEZ
y se hace que el consumidor pueda repetir sin efecto adicional
```

Y antes de nada, lo que cuesta la cola y suele olvidarse:

```
la respuesta al usuario deja de significar «hecho»
  → hay que decidir qué se le dice y cómo se entera del resultado
el error deja de aparecer donde se originó
  → hace falta correlacionar: el identificador de traza viaja en el mensaje
el sistema gana un estado nuevo: «pendiente»
  → y todo lo que consulte tiene que contemplarlo
```

2. El tiempo de invisibilidad, que es donde nacen los duplicados

El mecanismo es sencillo y su ajuste causa el error más común de esta clase:

```
el consumidor recibe el mensaje
el mensaje se oculta a los demás durante T
  si confirma antes de T   → desaparece
  si no                    → reaparece y otro consumidor lo toma
```

Y la consecuencia:

```
si el proceso tarda más que T, el trabajo se hace DOS veces
y nadie da ningún error: los dos consumidores creen que va bien
```

El ajuste correcto no se hace con la media:

```
tiempo de proceso, mediana        1,2 s
tiempo de proceso, percentil 99   14 s
tiempo de invisibilidad           30 s   ← al menos 2× el p99
```

Y para procesos de duración muy variable, alargar T no es la solución —porque un consumidor caído retendría el mensaje todo ese tiempo—, sino extender la invisibilidad mientras se trabaja:

```
T inicial corto, 30 s
y mientras el proceso sigue vivo, se renueva cada 15 s
→ si el consumidor muere, el mensaje vuelve en 30 s, no en 15 min
```

Y el otro par de errores, simétricos y ambos frecuentes:

```
CONFIRMAR ANTES DE TRABAJAR
  el mensaje desaparece y si el proceso falla, el trabajo se PIERDE
  → esto convierte una cola en «como mucho una vez» sin querer

CONFIRMAR DESPUÉS DE TRABAJAR
  correcto, y si el consumidor muere entre el trabajo y la confirmación,
```

```
el mensaje se repite
→ es el caso normal, y por eso hace falta poder repetir sin daño
```

Y una comprobación que conviene hacer explícitamente, porque casi nunca se hace:

```
matar un consumidor a mitad de proceso, a propósito
y comprobar qué ocurre con el mensaje y con sus efectos
```

Y el consumo por lotes, que multiplica el problema: si se reciben diez mensajes y falla el séptimo, ¿qué se confirma? La respuesta correcta es confirmar individualmente lo que salió bien; si el sistema solo permite confirmar el lote entero, los seis primeros se repetirán.

3. Reintentos que no empeoran la caída

El reintento inmediato es la peor opción posible, y es la que sale por defecto en mucho código:

```
la dependencia está saturada
→ los consumidores reintentan de inmediato
→ más carga sobre lo que ya está mal
→ la caída se alarga
```

Los tres parámetros, y los tres importan:

```
ESPERA CRECIENTE   1 s, 2 s, 4 s, 8 s…  da tiempo a que se recupere
VARIACIÓN          ±30 % aleatorio; sin ella, todos reintentan a la vez
LÍMITE DE INTENTOS 3 a 5; sin él, un mensaje venenoso vive para siempre
```

Y una distinción que ahorra la mayor parte de los reintentos inútiles:

```
ERROR TRANSITORIO     tiempo de espera agotado, 503, conflicto de bloqueo
                      → reintentar tiene sentido
ERROR PERMANENTE      mensaje mal formado, entidad que no existe,
                      validación fallida
                      → reintentar cinco veces no lo va a arreglar
                      → a la cola de fallidos directamente
```

Separarlos reduce mucho el ruido y acelera el diagnóstico: en la cola de fallidos quedan mezcladas dos cosas muy distintas si no se hace.

Y una tercera categoría que conviene tratar aparte: el mensaje que llega antes de tiempo. Un evento de pago que llega antes de que el pedido exista no es un error permanente ni transitorio: es orden. Reintentar con espera larga suele resolverlo, y si no, es señal de que hace falta el patrón de la clase 116.

La cola de fallidos. Su valor no está en existir, sino en lo que se hace con ella:

```
sirve para   inspeccionar el mensaje y el motivo del último fallo
             corregir la causa
             reprocesar cuando esté arreglado

no sirve     si nadie la mira
```

Y ahí vuelven dos leyes conocidas: un mensaje en la cola de fallidos no produce ningún error (ley 13) y una cola de fallidos con cuarenta mil elementos deja de mirarse (ley 15). Las dos alertas que lo evitan:

```
mensajes en la cola de fallidos ≥ 1        → avisar
antigüedad del más antiguo > 1 h           → avisar
```

La primera parece exagerada y no lo es: si llegar a la cola de fallidos es normal, el sistema tiene un problema de diseño, no de operación.

Y lo que hay que guardar con el mensaje para que sea reprocesable:

```
el cuerpo original, sin transformar
el motivo del último fallo y la traza
el número de intentos y las marcas de tiempo
el identificador de correlación
```

Y la herramienta de reproceso, que hay que escribir antes de necesitarla: devolver a la cola original en lotes controlados, no todo de golpe, porque devolver cuarenta mil mensajes a la vez reproduce la caída que los generó.

4. Ritmo, retraso y orden

Qué señal usar. La profundidad de la cola engaña:

```
50.000 mensajes y se consumen 20.000/s   → 2,5 s de retraso: bien
   200 mensajes y se consumen 2/s        → 100 s de retraso: mal
```

La señal correcta es la antigüedad del mensaje más viejo, que ya está en segundos y se compara directamente con lo que el negocio tolera.

Y el cálculo del tiempo de vaciado, que es lo que hay que saber durante un incidente:

```
vaciado = profundidad / (capacidad de consumo − ritmo de llegada)

120.000 mensajes, llegan 800/s, se consumen 1.000/s
→ 120.000 / 200 = 600 s = 10 min

y si llegan 1.100/s y se consumen 1.000/s
→ NUNCA se vacía: hay que escalar o parar la producción
```

Y el escalado del consumidor se hace con la antigüedad, no con la profundidad, por lo dicho arriba. Con un límite: el consumidor escala hasta donde aguanta su dependencia. Cien consumidores contra una base de datos de la clase 109 agotan sus conexiones; la cola solo mueve el cuello de botella.

El orden. La mayoría de los sistemas no lo garantizan, y conviene comprobar si de verdad hace falta:

```
necesita orden          actualizaciones sucesivas del mismo pedido
                        movimientos de un mismo saldo
no necesita orden       notificaciones, envío de correos, indexación
```

Y cuando hace falta, la respuesta casi nunca es «ordenar todo», sino ordenar por clave:

```
orden global      un solo consumidor: el ritmo lo marca el más lento
orden por clave   todos los mensajes de un pedido en el mismo grupo
                  → grupos distintos avanzan en paralelo
```

Y el precio del orden, que hay que aceptar conscientemente:

```
bloqueo por cabecera   un mensaje que falla detiene a los de su grupo
ritmo por grupo        limitado; el paralelismo lo da el número de grupos
escalado               no se puede repartir un grupo entre dos consumidores
```

El primero es el que más sorprende: un solo mensaje venenoso paraliza su clave entera hasta que agota intentos.

Y un recurso que evita imponer orden en muchos casos: que el mensaje lleve un número de versión y el consumidor descarte lo viejo. Si llega la versión 4 y luego la 3, se ignora la 3. Resuelve el problema sin ordenar nada.

Y la lista de comprobación de la clase:

- el consumidor tolera recibir el mismo mensaje dos veces
- el tiempo de invisibilidad es al menos el doble del p99 de proceso
- los procesos largos renuevan la invisibilidad mientras trabajan
- se confirma DESPUÉS de hacer el trabajo, e individualmente en lotes
- los errores permanentes van a fallidos sin reintentar
- los reintentos tienen espera creciente, variación y límite
- hay alerta por un solo mensaje en la cola de fallidos y por antigüedad
- existe herramienta de reproceso por lotes controlados
- el escalado usa la antigüedad del más viejo, no la profundidad
- está calculado hasta dónde puede escalar el consumidor sin romper su dependencia
- el orden se impone por clave y solo donde hace falta
- se ha matado un consumidor a mitad de proceso para ver qué ocurre

Y el cierre que enlaza con la clase siguiente: una cola entrega cada mensaje a un consumidor y lo borra. Cuando el mismo hecho interesa a varios sistemas —y hay que poder volver a leerlo dentro de un mes— hace falta otra cosa, y es la materia de la clase 114.

Ejemplo trabajado

CloudShop pasa el procesamiento de pedidos a una cola para dejar de hacerlo dentro de la petición del usuario. Funciona el primer día. Los cuatro problemas que aparecen después son los cuatro apartados de esta clase, en orden.

Punto de partida.

```
latencia de la petición del usuario       2,8 s          140 ms
fallos visibles al usuario por caída
de un sistema de pago                     4,1 %            0 %
pedidos procesados por segundo              85            1.200
```

Problema 1: pedidos duplicados, el 0,4 %.

```
pedidos con cobro duplicado en 3 semanas         214
```

```
tiempo de invisibilidad configurado              30 s
tiempo de proceso, mediana                      1,2 s
tiempo de proceso, percentil 99                  47 s   ← mayor que 30 s
```

El percentil 99 superaba el tiempo de invisibilidad: uno de cada cien mensajes se procesaba dos veces, y el sistema no daba ningún error porque los dos consumidores terminaban bien.

Y las llamadas al proveedor de pago lo confirmaban:

```
cobros solicitados                   52.180
pedidos                              51.966
diferencia                              214
```

Dos correcciones, y solo la segunda es de verdad:

```
duplicados por semana                    71            4                 0
retención tras muerte del consumidor     30 s          120 s             30 s
```

Subir la invisibilidad bajó los duplicados y empeoró otra cosa: un consumidor muerto retenía el mensaje dos minutos. La renovación durante el proceso resolvió las dos a la vez.

Y la corrección estructural quedó anotada para la clase 116: los duplicados residuales solo desaparecen cuando la operación se puede repetir sin efecto adicional.

Problema 2: la tormenta de reintentos alarga una caída de 4 minutos a 40.

```
09:12  el proveedor de pago devuelve 503
09:12  1.200 mensajes/s empiezan a fallar
09:12  reintento inmediato, sin espera ni límite
09:13  carga sobre el proveedor: ×6 respecto a lo normal
09:16  el proveedor se recupera internamente
09:16  la carga de reintentos le impide estabilizarse
09:52  se paran los consumidores a mano; el sistema se recupera
```

Cuatro minutos de fallo del proveedor, cuarenta de incidente. La configuración corregida y su ensayo:

```
espera entre intentos                    inmediata       1-2-4-8-16 s
variación                                no              ±30 %
límite de intentos                       ninguno         5
corte tras fallos consecutivos           no              sí

caída simulada de 4 min: duración del incidente
                                          40 min          5 min 10 s
carga de reintentos sobre el proveedor      ×6             ×1,3
```

Problema 3: la cola de fallidos con 41.000 mensajes que nadie miró.

```
mensajes en la cola de fallidos                41.216
el más antiguo                                 67 días
alertas configuradas sobre ella                 ninguna
pedidos afectados sin procesar                  1.847
el resto                        reintentos del mismo pedido y ruido
```

Es la ley 13 otra vez: 41.000 mensajes parados no producen ningún error. Y al inspeccionarlos apareció que el 71 % eran errores permanentes que se habían reintentado cinco veces cada uno:

```
validación fallida (campo obligatorio ausente)      18.400
entidad inexistente                                  8.900
mensaje mal formado                                  2.100
transitorios reales                                 11.816
```

Las correcciones:

```
errores permanentes                    5 reintentos      a fallidos directo
alerta por ≥ 1 mensaje                 no                sí
alerta por antigüedad > 1 h            no                sí
herramienta de reproceso               no había          por lotes de 500
mensajes en fallidos, estado normal    41.216            0-3
```

Y el primer reproceso enseñó por qué los lotes importan: devolver 11.816 mensajes de golpe reprodujo la saturación que los había generado. En lotes de 500 con pausa, se vaciaron en 40 minutos sin incidente.

Problema 4: el orden, que hacía falta solo para una cosa.

```
síntoma   un pedido cancelado y luego modificado quedaba activo
causa     los dos mensajes se procesaron en orden inverso
frecuencia  9 casos en 2 meses
```

La primera propuesta fue una cola con orden global:

```
mensajes por segundo                    1.200            85            1.140
consumidores en paralelo                   40             1               40
casos de orden invertido                    9             0                0
bloqueo por un mensaje venenoso            no       toda la cola     solo ese pedido
```

El orden global habría dividido el ritmo por catorce. Se ordenó por identificador de pedido.

Y para el resto de los consumidores, que no necesitan orden, se aplicó la alternativa del apartado cuarto:

```
el mensaje lleva número de versión del pedido
el consumidor descarta lo que sea anterior a lo que ya aplicó
→ resuelve el problema sin ordenar nada
```

La señal de escalado, corregida por el camino.

```
señal                              profundidad > 10.000   antigüedad > 60 s
escalados innecesarios al mes            14                   0
retrasos no detectados                    3                   0
límite de consumidores                sin límite      24 (por las conexiones
                                                       de la clase 109)
```

La última fila viene de un incidente encadenado: al escalar a 90 consumidores durante un retraso, se agotaron las conexiones de la base de datos y cayó también la parte síncrona. La cola movió el cuello de botella en lugar de eliminarlo.

A los cinco meses.

```
latencia de la petición                    2,8 s          140 ms
pedidos duplicados                       214 / 3 sem        0-2
duración de un incidente de 4 min           40 min        5 min 10 s
mensajes en cola de fallidos               41.216          0-3
alertas sobre la cola de fallidos               0            2
casos de orden invertido                    9 / 2 mes        0
señal de escalado                       profundidad      antigüedad
escalados innecesarios                     14 / mes          0
caídas encadenadas por escalar consumidores     1             0
```

La lección que esta clase traslada a la parte 09: los cuatro problemas tienen la misma raíz. La cola resolvió el acoplamiento en el tiempo y trasladó a la aplicación tres garantías que antes daba la llamada síncrona: que el trabajo se hace una vez, que el error se ve donde ocurre y que las cosas pasan en orden. Ninguna de las tres vuelve sola: hay que reconstruirlas. Y la más difícil —hacer que repetir no tenga efecto— es la única que esta clase no ha resuelto, solo aplazado.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/113-colas-entrega-reintentos-y-dead-letter-queues/lab.py
```

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

Evidencia esperada

El artefacto principal es un flujo que declara orden, entrega y manejo de errores. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

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

■ Criterio de aceptación

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

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
Trabajo duplicado sin ningún error visible	El proceso dura más que el tiempo de invisibilidad y el mensaje se reentrega	Ajusta la invisibilidad al doble del percentil 99 y renuévala mientras el proceso siga vivo.
Se pierden mensajes cuando un consumidor falla	Se confirma antes de hacer el trabajo	Confirma siempre después, y en lotes confirma individualmente lo que salió bien.
Una caída breve de una dependencia se convierte en un incidente largo	Reintento inmediato y sin límite, que amplifica la carga sobre lo que ya está mal	Espera creciente con variación, límite de intentos y corte tras fallos consecutivos.
La cola de fallidos acumula decenas de miles de mensajes	Ley 13: nada da error, y ley 15: con esa cifra ya nadie la mira	Alerta desde el primer mensaje y por antigüedad, y envía los errores permanentes sin reintentar.
Reprocesar la cola de fallidos vuelve a tumbar el sistema	Se devolvieron todos los mensajes de golpe	Reprocesa en lotes controlados con pausa entre ellos.
Escalar los consumidores tumba la base de datos	La cola movió el cuello de botella; el límite real es la dependencia	Fija el máximo de consumidores según lo que aguanta la dependencia y escala por antigüedad del mensaje más viejo.

■ 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 no existe la entrega exactamente una vez y qué se ofrece con ese nombre?
2. ¿Cómo se calcula el tiempo de invisibilidad y qué se hace con procesos largos?
3. ¿Qué diferencia hay entre error transitorio y permanente a efectos de reintento?
4. ¿Por qué se alerta desde el primer mensaje en la cola de fallidos?
5. ¿Por qué la antigüedad del mensaje más viejo es mejor señal que la profundidad?

Referencias

- AWS (2025). SQS: visibility timeout and dead-letter queues — invisibilidad, reentrega y destino de fallidos. <https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-visibility-timeout.html>

- Azure (2025). Service Bus: message transfers, locks and settlement — bloqueo, renovación y confirmación. <https://learn.microsoft.com/azure/service-bus-messaging/message-transfers-locks-settlement>
- Google Cloud (2025). Pub/Sub: subscription retry policy and dead lettering — espera creciente y reenvío. <https://cloud.google.com/pubsub/docs/handling-failures>
- Brooker, M. (2015). Exponential backoff and jitter — por qué la variación importa tanto como la espera. <https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/>
- Nygard, M. (2018). Release It!, cap. 4 — tormentas de reintentos y cortocircuitos. <https://pragprog.com/titles/mnee2/release-it-second-edition/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 09 en PDF · Manual integral

Anterior	Índice	Siguiente
← 112 · Object storage, data lake y formatos columnares	Parte 09 · Programa	114 · Pub/sub, streams, particiones y orden →

Evaluación — 113 Colas, entrega, reintentos y dead-letter queues

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 cola-resiliente con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

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

Escala

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

114 — Pub/sub, streams, particiones y orden

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 4 Laboratorio: messaging · Estado: EXECUTABLECORE

Propósito

Pasar de la cola —que entrega y borra— al registro ordenado que conserva, donde cada consumidor lleva su propia posición y puede volver atrás. Ese cambio resuelve tres cosas que la clase 113 no podía: varios consumidores independientes del mismo hecho, releer el pasado y reconstruir estado desde cero. Y trae dos decisiones que se toman al crear y no se deshacen: cuántas particiones y qué clave las elige, que juntas deciden el paralelismo máximo, el orden y dónde aparecerá la partición caliente.

Resultados de aprendizaje

Al finalizar podrás:

1. Distinguir cuándo hace falta un registro conservado y cuándo basta una cola.
2. Elegir número de particiones y clave, entendiendo lo que fijan para siempre.
3. Razonar sobre grupos de consumidores, reparto y lo que cuesta un rebalanceo.
4. Gestionar la posición de lectura sin perder ni repetir rangos enteros.
5. Usar retención y compactación para reconstruir estado.

Conceptos centrales

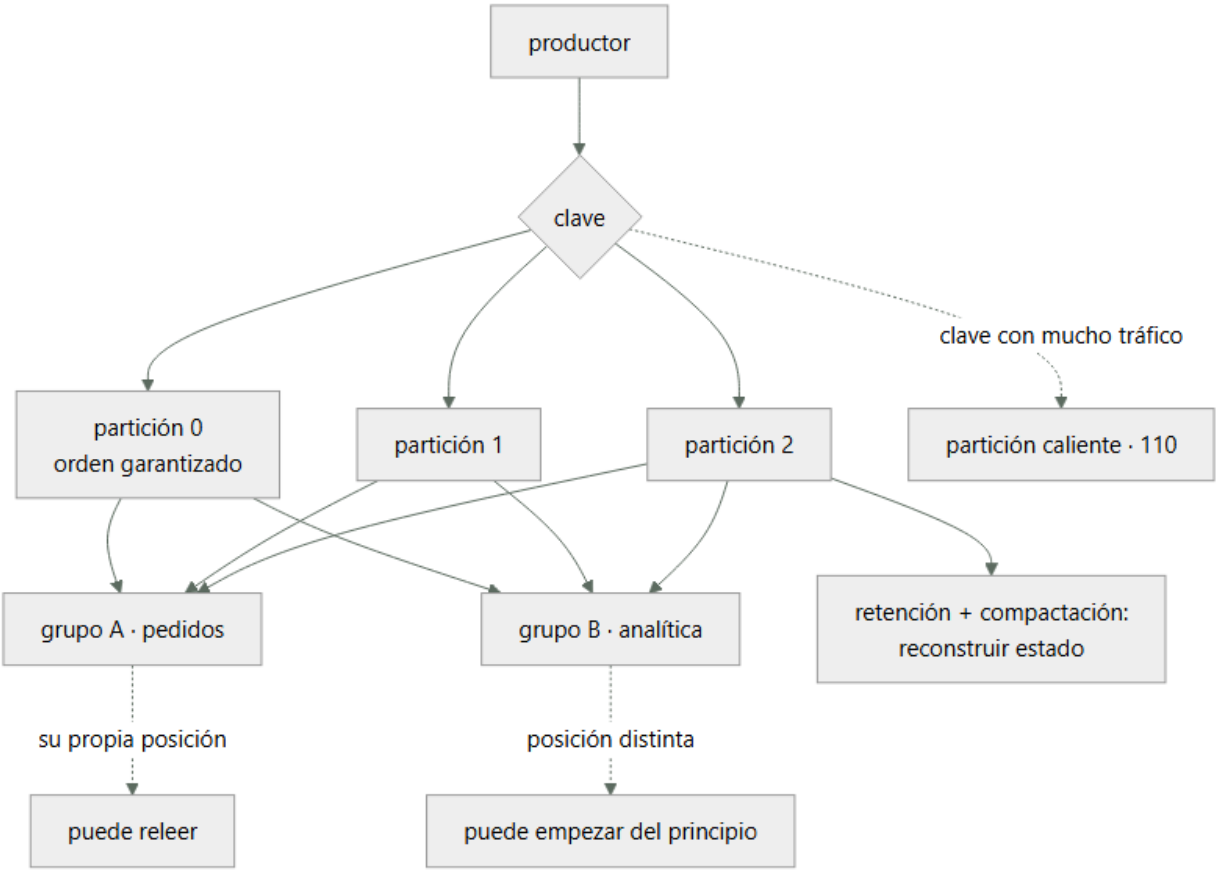
Concepto	Comprensión verificable
registro conservado	Secuencia ordenada e inmutable de mensajes que no se borran al leerse. Cada consumidor avanza por su cuenta.
partición	Subsecuencia ordenada del tema. Es a la vez la unidad de paralelismo y la única dentro de la cual hay orden.
posición de lectura	Índice del último mensaje procesado por un grupo. Confirmarla mal salta o repite rangos enteros, no un mensaje.
grupo de consumidores	Conjunto que se reparte las particiones. Ninguna partición la leen dos miembros a la vez, así que el paralelismo máximo es el número de particiones.
rebalanceo	Reparto de particiones cuando entra o sale un miembro. Durante él el consumo se detiene y suele haber reproceso.
compactación por clave	Conservar solo el último valor de cada clave. Convierte el registro en una fotografía reconstruible del estado actual.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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. Conservar cambia lo que se puede hacer

La diferencia con la clase 113 es una sola y tiene muchas consecuencias:

```
COLA        se entrega y se borra
            un mensaje, un consumidor
            lo procesado ya no existe

REGISTRO    se escribe y se conserva
            cada consumidor lleva su posición
            lo procesado sigue ahí
```

Y lo que eso permite:

```
varios consumidores independientes del mismo hecho
  facturación, analítica, notificación, búsqueda
  → y añadir uno nuevo no afecta a los demás

releer
  un consumidor con un defecto reprocesa desde una posición anterior
  → sin pedir nada al productor

empezar del principio
  un servicio nuevo reconstruye su estado leyendo el histórico
  → esto es lo que no se puede hacer con una cola
```

Y el criterio para elegir uno u otro:

```
cola        trabajo que alguien tiene que hacer una vez
            «envía este correo», «cobra este pedido»
registro    hecho que ya ocurrió y que interesa a varios
            «el pedido 1421 pasó a pagado»
```

La distinción se nota en el nombre del mensaje: un imperativo va a una cola; un hecho en pasado va a un registro.

Y lo que cuesta conservar:

```
el almacenamiento se paga por el periodo de retención
los mensajes viejos siguen ahí cuando el código ya cambió
  → un consumidor debe poder leer formatos anteriores
  → es la convivencia N/N+1 de la clase 102, aplicada a los datos
borrar un dato concreto es difícil: el registro es inmutable
  → y hay requisitos legales que lo exigen (clase 112)
```

La última línea tiene dos respuestas conocidas: compactar con una marca de borrado por clave, o guardar los datos personales fuera del registro y publicar solo referencias. La segunda es la que menos disgustos da.

2. Particiones: paralelismo y orden a la vez

La partición es la pieza central y hace dos trabajos que conviene ver separados:

```
es la unidad de ORDEN          dentro de una partición hay orden total
                               entre particiones no hay ninguno
es la unidad de PARALELISMO    cada partición la lee un solo miembro del grupo
                               → paralelismo máximo = número de particiones
```

Y de ahí sale la primera decisión irreversible en la práctica:

```
¿cuántas particiones?
  pocas     techo de paralelismo bajo
  muchas    más metadatos, más ficheros, rebalanceos más lentos
            y más coste por partición en los servicios administrados
```

Y por qué aumentar el número después es un problema:

```
partición = f(clave) mod N

si N cambia, la misma clave pasa a otra partición
→ los mensajes viejos de esa clave están en la partición vieja
→ y el orden por clave, que era la garantía, se rompe en la frontera
```

Así que ampliar es posible y no conserva la garantía de orden para las claves ya existentes. Es la ley 14 en su forma más limpia: una decisión de creación con consecuencias que no se pueden deshacer sin migrar.

La regla práctica al dimensionar:

```
particiones ≥ consumidores máximos previstos
y con margen: 2 a 3 veces el paralelismo actual
pero no cientos «por si acaso»
```

La clave, que es la segunda decisión y hereda todo lo de la clase 110:

```
misma clave → misma partición → orden garantizado entre esos mensajes

clave = identificador de pedido    orden por pedido, reparto bueno
clave = país                       5 valores → 5 particiones útiles
clave = nula (reparto rotatorio)   máximo reparto, ningún orden
```

Y la partición caliente vuelve exactamente igual que en la clase 110: una clave con mucho tráfico satura su partición mientras las demás están ociosas, y no se puede repartir porque repartirla rompería el orden que justificaba la clave.

```
si una clave concentra demasiado tráfico:
  ¿de verdad necesita orden esa clave?
    no  → quítale la clave y reparte
    sí  → hay un límite físico, y hay que reducir el trabajo por mensaje
```

3. Grupos, rebalanceo y posición

El grupo de consumidores reparte las particiones entre sus miembros:

```
12 particiones, 4 consumidores    → 3 particiones cada uno
12 particiones, 12 consumidores   → 1 cada uno
12 particiones, 20 consumidores   → 8 CONSUMIDORES OCIOSOS
```

La tercera línea es la que sorprende a quien viene de las colas: añadir consumidores por encima del número de particiones no aumenta nada. El autoescalado por retraso tiene ahí su techo, y hay que declararlo.

El rebalanceo ocurre cuando entra o sale un miembro, y su coste es real:

```
se detiene el consumo de todo el grupo
se reparte de nuevo
cada consumidor empieza desde la última posición confirmada
  → lo procesado y no confirmado se REPITE
```

Y el error más frecuente es provocarlo sin querer:

```
el consumidor tarda más de lo permitido entre lecturas
→ el coordinador lo da por muerto
→ rebalanceo
→ el consumidor vuelve y provoca otro rebalanceo
→ y en ese bucle el grupo no avanza
```

Lo que lo corrige:

```
leer lotes más pequeños, para volver antes a pedir
separar el latido del proceso, si el cliente lo permite
subir el tiempo máximo entre lecturas al p99 real del proceso
y usar reparto incremental, que no detiene a todo el grupo
```

La posición de lectura. Aquí la diferencia con una cola es importante y peligrosa: no se confirma un mensaje, se confirma una posición, y todo lo anterior queda dado por hecho.

```
confirmar ANTES de procesar   si falla, se saltan mensajes: se PIERDEN
confirmar DESPUÉS de procesar si falla, se repite el lote: al menos una vez
confirmar cada mensaje        seguro y lento
confirmar por lotes           rápido, y repite el lote entero al fallar
```

Y el caso que rompe a mucha gente: procesar en paralelo dentro de un lote y confirmar la posición mayor. Si el mensaje 7 falla y el 9 va bien, confirmar 9 da por hecho el 7.

```
si se procesa en paralelo, la posición confirmable es
la del último mensaje SIN huecos anteriores
```

Y la señal de estado, que aquí tiene dos formas y hay que mirar las dos:

```
retraso en mensajes    cuántos faltan por leer
retraso en tiempo      cuánto hace que se produjo el que se está leyendo
```

La segunda es la que se compara con lo que el negocio tolera; la primera dice si se está recuperando.

4. Retención, compactación y reconstrucción

La retención es lo que convierte el registro en algo más que un transporte.

```
por tiempo    conserva N días
por tamaño    conserva N GB por partición
permanente    con almacenamiento por capas, el histórico completo
```

Y lo que la retención permite, que es lo que justifica todo el modelo:

```
un consumidor con un defecto retrocede su posición y reprocesa
un servicio nuevo empieza en el principio y construye su estado
un incidente se investiga leyendo lo que pasó de verdad
```

Y la advertencia correspondiente: la retención es también el plazo de recuperación. Si se conservan tres días y un defecto se descubre a los cinco, no hay nada que reprocesar.

```
retención ≥ tiempo típico hasta descubrir un defecto
→ en la práctica, 7 días es poco y 30 suele ser suficiente
```

La compactación por clave es distinta de la retención por tiempo y resuelve otro problema:

```
conserva el ÚLTIMO mensaje de cada clave, para siempre
descarta los anteriores

→ el tema deja de ser un histórico y pasa a ser una fotografía
→ leerlo entero reconstruye el estado actual de todas las claves
```

Y para qué sirve de verdad:

```
reconstruir un caché tras perderlo (clase 111)
dar estado inicial a un servicio nuevo sin consultar a la base
mantener una copia local de una tabla de referencia
```

Y el detalle imprescindible: para borrar una clave se publica un mensaje con valor vacío. Sin eso, una clave borrada en el origen vive para siempre en el tema compactado.

Y dos cosas que conviene no mezclar:

```
tema de HECHOS       histórico, retención por tiempo, no compactar
                     «el pedido 1421 pasó a pagado»
tema de ESTADO       compactado, última versión por clave
```

```
                    «el pedido 1421 está así»
```

Mezclarlas es un error frecuente: compactar un tema de hechos borra historia que alguien necesitaba.

Y un último aviso sobre los reintentos, que enlaza con la clase 113: reintentar un mensaje fuera del registro rompe el orden. Si el mensaje 5 falla y se manda a una cola de reintentos mientras el 6 y el 7 se procesan, se ha perdido la garantía que justificaba la partición. Las dos salidas honestas:

```
detenerse en el fallo y no avanzar esa partición
  → conserva el orden y bloquea a los de detrás
enviar a un tema de reintentos y ACEPTAR que se pierde el orden
  → hay que declararlo, no descubrirlo
```

Y la lista de comprobación de la clase:

- ■ está justificado por qué esto es un registro y no una cola
- ■ el número de particiones cubre el paralelismo máximo previsto
- ■ está entendido que ampliar particiones rompe el orden por clave
- ■ la clave reparte y no concentra tráfico en una partición
- ■ el máximo de consumidores útiles está declarado y no se escala más allá
- ■ el tiempo máximo entre lecturas es mayor que el p99 de proceso
- ■ la posición se confirma después de procesar y sin huecos
- ■ se vigilan retraso en mensajes y retraso en tiempo
- ■ la retención cubre el tiempo típico hasta descubrir un defecto
- ■ los temas de hechos no están compactados
- ■ el borrado por clave publica un mensaje vacío
- ■ está declarado si los reintentos rompen el orden

Y el cierre que enlaza con la clase siguiente: si varios sistemas leen los mismos hechos, el formato de esos hechos es un contrato entre equipos que no se ven. Cómo se define, cómo se cambia sin romper a quien lee el histórico y quién es su dueño es la materia de la clase 115.

Ejemplo trabajado

CloudShop publica los hechos de pedido en un registro conservado para que tres equipos los consuman por su cuenta. El diseño inicial es razonable y contiene dos decisiones de creación que costarán una migración.

Por qué un registro y no una cola.

```
consumidores del mismo hecho          facturación, analítica, búsqueda
con una cola                          3 colas, y el productor tiene que
                                      publicar en las tres
añadir un cuarto consumidor           tocar al productor
reprocesar tras un defecto            pedir al productor que reenvíe
```

Y con registro, los cuatro problemas desaparecen: el productor publica una vez y cada consumidor avanza a su ritmo.

Decisión 1: seis particiones. Se queda corta a los cinco meses.

```
al diseñar                    1.200 mensajes/s, 4 consumidores
particiones elegidas          6
mes 5                         4.900 mensajes/s
consumidores necesarios       11
consumidores útiles           6   ← techo
consumidores ociosos          5
retraso en tiempo             de 2 s a 4 min 20 s
```

Y al ampliar a 24 particiones apareció lo del apartado segundo:

```
mensajes de un mismo pedido antes de ampliar    en la partición 3
mensajes del mismo pedido después              en la partición 17
ventana en la que el orden por pedido no está garantizado
                                                 la retención: 30 días
casos de orden invertido detectados             41
```

La migración honesta consistió en publicar en un tema nuevo con 24 particiones, consumir de los dos durante treinta días y retirar el viejo. Doce minutos de decisión inicial, cinco semanas de migración.

Decisión 2: la clave. La primera fue el país.

```
clave = país
valores distintos                    190
particiones que reciben tráfico       6
tráfico de España                    61 % del total
carga de la partición de España      61 %
```

```
carga de la partición menos usada    0,4 %
retraso de la partición de España    9 min
retraso del resto                    < 1 s
```

Es la partición caliente de la clase 110, en otro sistema y con el mismo diagnóstico. Y la pregunta del apartado segundo, respondida:

```
¿necesita orden el conjunto de un país?    no
¿qué necesita orden?                       los mensajes de un mismo pedido
→ clave = identificador de pedido
```

```
particiones con tráfico                      6            24
desviación entre la más y la menos cargada   ×150         ×1,3
retraso máximo                               9 min        2 s
casos de orden invertido                     0            0
```

El bucle de rebalanceo, que paralizó el consumo 40 minutos.

```
02:10  un consumidor tarda 6 min en un lote (una dependencia lenta)
02:15  el coordinador lo da por muerto: tiempo máximo entre lecturas, 5 min
02:15  rebalanceo; el grupo se detiene
02:16  el consumidor vuelve → otro rebalanceo
02:16  … se repite
02:50  se detecta y se corrige a mano
```

```
tamaño del lote                          500 mensajes      50 mensajes
tiempo de proceso por lote, p99          6 min             38 s
tiempo máximo entre lecturas             5 min             5 min
reparto                                  detiene el grupo  incremental
rebalanceos por semana                   14                0-1
minutos de consumo detenido por semana   47                < 1
```

El error de posición que perdió 9.000 mensajes.

Un consumidor procesaba el lote en paralelo con veinte hilos y confirmaba la posición mayor al terminar:

```
lote de 500, mensajes 12.000 a 12.499
el mensaje 12.203 falla y se registra el error
el resto termina bien
se confirma la posición 12.499
→ el 12.203 nunca se procesa y nadie vuelve a él
```

```
mensajes perdidos así en 3 semanas                 9.140
detectado por    un descuadre en facturación de 21.400 €
corrección       confirmar la posición del último SIN huecos anteriores
mensajes perdidos tras la corrección                   0
reproceso        se retrocedió la posición 3 semanas y se repitió
                 → posible SOLO porque la retención era de 30 días
```

La última línea es el argumento del apartado cuarto: con siete días de retención, esos 21.400 € no se habrían podido recuperar.

La compactación, para reconstruir el caché de la clase 111.

El caché portante de la clase 111 tardaba catorce minutos en recalentarse contra la base de datos. Con un tema de estado compactado:

```
tiempo de recalentamiento              14 min           1 min 50 s
carga sobre la base durante el proceso  12.000/s        0
tamaño del tema (2,1 M de productos)      —             3,4 GB
```

Y el fallo que se cometió al montarlo: se compactó también el tema de hechos.

```
hechos conservados antes de compactar        4.100 millones
después                                      2,1 millones (uno por pedido)
histórico perdido                            todo lo intermedio
recuperado desde                             la zona bruta del lago (clase 112)
```

Se recuperó porque la zona bruta existía. Sin ella, el histórico de pedidos se habría perdido de forma irreversible.

A los ocho meses.

```
consumidores independientes                      3            6
particiones                                      6           24
clave                                          país       pedido
desviación de carga entre particiones          ×150          ×1,3
retraso en tiempo, p99                       4 min 20 s      1,8 s
```

```
rebalanceos por semana                          14            0-1
mensajes perdidos por confirmación con huecos  9.140           0
retención                                     30 días        30 días
temas de hechos compactados por error            1             0
recalentamiento del caché                     14 min         1 min 50 s
```

La lección que esta clase traslada a la parte 09: los dos problemas caros —ampliar particiones y cambiar la clave— no fueron fallos de operación ni de código. Fueron dos decisiones tomadas en la primera media hora de diseño, cuando el sistema tenía la cuarta parte del tráfico, y ninguna de las dos se podía deshacer sin migrar. Es la tercera confirmación seguida de la predicción de la clase 108, y ya se puede afinar: en los sistemas con estado, las decisiones baratas de tomar son las caras de cambiar, y se toman antes de tener datos para tomarlas bien.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/114-pub-sub-streams-particiones-y-orden/lab.py
```

El laboratorio selecciona el motor de práctica messaging 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 stream-particionado 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 flujo que declara orden, entrega y manejo de errores. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye stream-particionado 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
Añadir consumidores no mejora el retraso	Ya hay tantos consumidores como particiones; el resto están ociosos	Declara el máximo útil, no escales por encima, y dimensiona las particiones con margen desde el principio.
El orden por clave se rompe tras ampliar particiones	La asignación de clave a partición depende del número de particiones	Publica en un tema nuevo con el tamaño correcto y consume de los dos durante la retención; ampliar en sitio no conserva la garantía.
Una partición acumula retraso y las demás están ociosas	La clave concentra el tráfico, como en la clase 110	Elige una clave con reparto real; si la que concentra no necesita orden, quítale la clave.
El grupo entra en un bucle de rebalanceos y no avanza	El proceso de un lote dura más que el tiempo máximo entre lecturas	Reduce el tamaño del lote, ajusta el tiempo máximo al percentil 99 real y usa reparto incremental.
Se pierden mensajes sin ningún error	Se confirmó una posición por delante de un mensaje que falló	Confirma la posición del último mensaje sin huecos anteriores, nunca la mayor del lote.
Se pierde histórico que alguien necesitaba	Se compactó un tema de hechos, que conserva solo el último por clave	Separa temas de hechos (retención por tiempo) de temas de estado (compactados) y no mezcles las políticas.

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 cosas permite un registro conservado que una cola no permite?
2. ¿Por qué el número de particiones limita el paralelismo y por qué ampliarlo rompe el orden por clave?
3. ¿Qué ocurre durante un rebalanceo y cómo se evita un bucle de rebalanceos?
4. ¿Qué posición se puede confirmar si se procesa un lote en paralelo?
5. ¿Para qué sirve la compactación por clave y qué tema no debe compactarse nunca?

Referencias

- Apache Kafka (2025). Design: topics, partitions and consumer groups — orden por partición y reparto del grupo. <https://kafka.apache.org/documentation/#design>
- Apache Kafka (2025). Log compaction — retención por clave y borrado con mensaje vacío. <https://kafka.apache.org/documentation/#compaction>
- Google Cloud (2025). Pub/Sub: ordering keys and message retention — orden por clave y relectura. <https://cloud.google.com/pubsub/docs/ordering>
- AWS (2025). Kinesis Data Streams: shards and resharding — paralelismo por fragmento y coste de reparticionar. <https://docs.aws.amazon.com/streams/latest/dev/kinesis-using-sdk-java-resharding.html>
- Kleppmann, M. (2017). Designing Data-Intensive Applications, cap. 11 — registros, posiciones y reproceso. <https://dataintensive.net/>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 09 en PDF · Manual integral

Evaluación — 114 Pub/sub, streams, particiones y orden

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 stream-particionado 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.

115 — Arquitectura dirigida por eventos y contratos

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 4 Laboratorio: architecture · Estado: EXECUTABLECORE

Propósito

Diseñar el contenido de los hechos que viajan por el registro de la clase 114, que es donde se decide si una arquitectura de eventos desacopla de verdad o solo cambia el acoplamiento de sitio. La clase distingue tres cosas que se llaman igual y no lo son, establece que el esquema del evento es una interfaz pública con más obligaciones que una API, porque el registro conserva mensajes más viejos que cualquier versión del código, y afronta el precio que nadie anuncia: cuando nadie orquesta, nadie sabe responder dónde está el pedido 1421.

Resultados de aprendizaje

Al finalizar podrás:

1. Distinguir notificación, transferencia de estado y comando disfrazado.
2. Redactar eventos como hechos, con envoltura estándar y contenido acotado.
3. Evolucionar un esquema conservando compatibilidad en las dos direcciones.
4. Elegir entre coreografía y orquestación sabiendo qué se paga.
5. Evitar que el esquema interno de una base de datos se convierta en contrato público.

Conceptos centrales

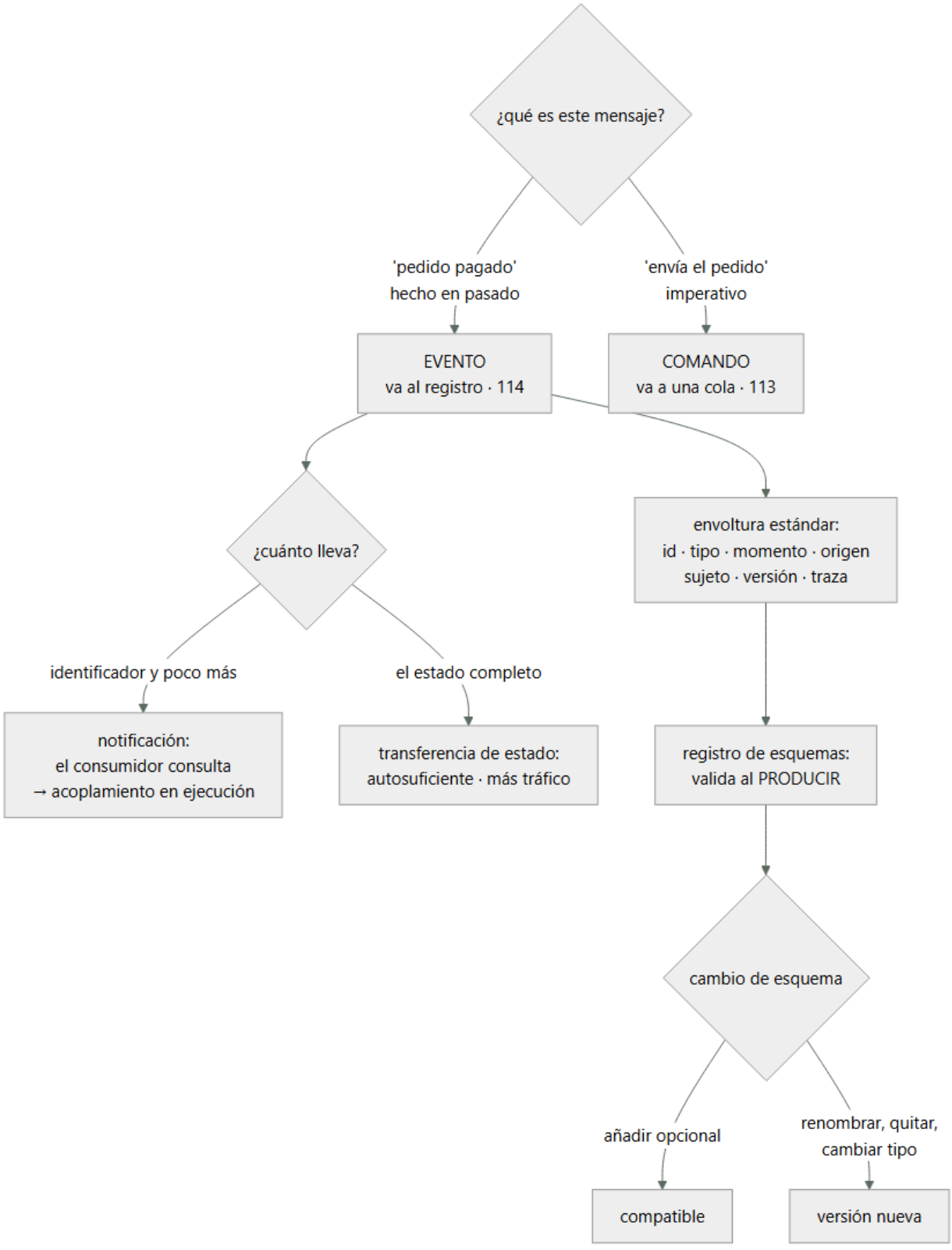
Concepto	Comprensión verificable
notificación	Aviso mínimo de que algo pasó; el consumidor consulta al origen para saber más. Poco tráfico, y devuelve el acoplamiento en tiempo de ejecución.
transferencia de estado	El evento lleva el estado resultante completo. El consumidor no necesita preguntar nada, y a cambio recibe mucho más dato.
comando disfrazado	Mensaje en imperativo con forma de evento. El productor decide qué debe hacer el consumidor, que es justo lo que se quería evitar.
compatibilidad en dos direcciones	Un consumidor nuevo debe leer mensajes viejos y uno viejo debe tolerar mensajes nuevos. Con registros conservados hacen falta las dos.
envoltura	Metadatos comunes a todos los eventos: identificador, tipo, momento, origen, sujeto, versión y traza. Se estandariza una vez para toda la organización.
coreografía	Cada servicio reacciona a los hechos sin que nadie dirija. Escala organizativamente y hace muy difícil saber el estado de un proceso completo.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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

■ Flujo de razonamiento



Desarrollo

1. Tres cosas que se llaman evento

La mayor parte de los problemas de una arquitectura de eventos vienen de mezclar estas tres:

```
NOTIFICACIÓN
  {"tipo":"pedido.pagado", "pedido":"1421"}
  + mínimo tráfico; el productor no expone su modelo
```

```
  − el consumidor tiene que llamar al origen para saber algo
  − y entonces el origen tiene que estar disponible: acoplamiento en ejecución

TRANSFERENCIA DE ESTADO
  {"tipo":"pedido.pagado", "pedido":{ … estado completo … }}
  + el consumidor es autosuficiente y puede procesar en diferido
  + soporta que el origen esté caído
  − más tráfico, y expone parte del modelo del productor

COMANDO DISFRAZADO
  {"tipo":"debe.enviarse.el.pedido", …}
  → no es un evento: es una orden con otro nombre
  → el productor está decidiendo qué hace el consumidor
```

La tercera es la que arruina el diseño sin que se note. La prueba para detectarla:

```
¿el nombre está en pasado y describe algo que YA ocurrió?
  sí → evento; el productor no sabe quién lo consume ni le importa
  no → comando; va a una cola, con un destinatario concreto
```

Y la elección entre las dos primeras, que es real y se decide por caso:

```
notificación              cuando el dato es grande, cambia mucho
                          o es sensible
transferencia de estado   cuando el consumidor debe poder trabajar sin
                          el origen, o el proceso es diferido
```

Y una tercera vía muy usada y muy útil: notificación con lo suficiente. Se incluye lo que el 90 % de los consumidores necesita y se deja el resto tras una consulta. La forma sana de decidirlo es preguntar a los consumidores, que es lo que el apartado tercero convierte en obligación.

La envoltura, que conviene estandarizar una vez para toda la organización:

```
{
  "id": "e-9f2c41…",
  "tipo": "pedido.pagado",
  "version": 2,
  "momento": "2026-08-03T09:14:22Z",
  "origen": "servicio-pedidos",
  "sujeto": "pedido/1421",
  "traza": "00-a91c…-b7f2…-01",
  "clave_idempotencia": "pago-1421-3",
  "datos": { … }
}
```

Y cada campo está por un motivo concreto:

```
id                  permite detectar repeticiones (clase 113)
tipo y versión      permiten evolucionar sin romper
momento             el del HECHO, no el de la publicación: son distintos
sujeto              permite filtrar sin abrir el contenido
traza               enlaza el evento con la petición que lo originó
clave_idempotencia  la desarrolla la clase 116
```

La diferencia entre momento del hecho y momento de publicación importa más de lo que parece: en un reproceso, el segundo cambia y el primero no.

2. El esquema es una interfaz pública

Un evento publicado lo leen equipos que no conoces, con código que no controlas, y —esto es lo específico de la clase 114— mensajes escritos hace treinta días siguen ahí. Eso obliga a más que una API.

```
en una API          conviven la versión N y la N+1 unos días
en un registro      conviven mensajes de TODAS las versiones publicadas
                    durante todo el periodo de retención
                    y consumidores que aún no se han actualizado
```

De ahí que hagan falta las dos direcciones a la vez:

```
HACIA ATRÁS   el consumidor NUEVO lee mensajes VIEJOS
              → necesario para reprocesar el histórico
HACIA DELANTE el consumidor VIEJO tolera mensajes NUEVOS
              → necesario porque no se puede desplegar a todos a la vez
```

Y las reglas que las conservan:

```
SE PUEDE
  añadir un campo OPCIONAL con valor por defecto
  añadir un valor nuevo a una enumeración, si los consumidores
    tienen un caso «desconocido»
  añadir un tipo de evento nuevo

NO SE PUEDE
  quitar un campo
  renombrar un campo
  cambiar el tipo de un campo
  convertir en obligatorio uno que era opcional
  cambiar el significado de un campo sin cambiar su nombre  ← el peor
```

El último es el más dañino porque ninguna herramienta lo detecta: el esquema sigue validando y los consumidores interpretan mal. Si importe pasa de incluir impuestos a no incluirlos, hay que cambiar el nombre.

Y cuando hay que hacer algo de la lista prohibida, el procedimiento es el de la clase 102 —expandir y contraer— aplicado a eventos:

```
1. publicar el campo nuevo junto al viejo
2. esperar a que todos los consumidores usen el nuevo, y comprobarlo
3. dejar de publicar el viejo, cuando ya no queden mensajes con él
   dentro de la retención
```

El paso 3 tiene aquí una condición extra: hay que esperar además a que expire la retención, o un reproceso encontrará mensajes sin el campo nuevo.

El registro de esquemas es lo que convierte estas reglas en algo que se cumple:

```
guarda el esquema de cada tipo y versión
valida en el momento de PRODUCIR, no de consumir
rechaza los cambios incompatibles antes de publicarlos
```

La segunda línea es la importante: validar al consumir llega tarde, porque el mensaje malo ya está en el registro y seguirá ahí durante toda la retención.

Y una obligación organizativa que hace posible todo lo anterior: el productor tiene que saber quién le consume. No para pedirle permiso, sino para poder avisar y para poder comprobar el paso 2.

```
catálogo por tema: quién produce, quién consume, desde cuándo
→ es el catálogo de la clase 095, con una columna más
```

3. Coreografía, orquestación y el pedido 1421

Con hechos publicados, hay dos formas de que un proceso de negocio avance:

```
COREOGRAFÍA
  cada servicio escucha y reacciona; nadie dirige
  + los equipos avanzan sin coordinarse
  + añadir un participante no toca a nadie
  − nadie conoce el proceso completo
  − y nadie puede responder «¿dónde está el pedido 1421?»

ORQUESTACIÓN
  un componente conoce los pasos y los invoca
  + el proceso está en un sitio, legible y consultable
  + los fallos y compensaciones tienen dueño
  − ese componente acopla a todos los participantes
  − y crece hasta contener la lógica de negocio de todos
```

Y el criterio práctico, que no es elegir una:

```
entre equipos y entre dominios        coreografía
dentro de un proceso de negocio con
pasos, plazos y compensaciones        orquestación
```

La segunda es la clase 119, y las compensaciones son la 116.

Y el precio de la coreografía hay que pagarlo explícitamente, porque no se paga solo:

```
sin nada        nadie sabe en qué paso está un pedido, ni por qué se paró
con lo mínimo   la traza viaja en la envoltura, y hay una vista que
                reúne todos los eventos de un sujeto
```

Y esa vista —«dame todo lo que le ha pasado al pedido 1421»— es obligatoria en una arquitectura coreografiada. Sin ella, el diagnóstico de cualquier incidente consiste en preguntar a seis equipos.

```
lo mínimo que hace falta
  identificador de traza en todos los eventos del proceso
  campo sujeto uniforme: «pedido/1421»
  un sitio donde consultar por sujeto y ver la secuencia con tiempos
  y qué se esperaba y no llegó
```

La última línea es la difícil: en coreografía, que un paso no ocurra no produce ningún error. Es la ley 13 en versión de negocio, y la única defensa es un vigilante de plazos:

```
si «pedido.pagado» no va seguido de «pedido.preparado» en 30 min
→ alguien tiene que enterarse
```

Y dos antipatrones frecuentes de la coreografía:

```
BUCLE DE EVENTOS      A reacciona a B y B reacciona a A
                      → se detecta contando saltos en la traza
CASCADA INVISIBLE     un evento dispara once consumidores y nadie lo sabía
                      → el catálogo de consumidores lo hace visible
```

4. No publiques tu base de datos

Hay una forma muy cómoda de empezar a publicar eventos: leer el registro de cambios de la base de datos y publicar cada fila modificada. Es potente y tiene una trampa grande.

```
lo que se publica          filas de tus tablas
lo que eso significa       tu esquema interno pasa a ser contrato público
consecuencia              no puedes renombrar una columna sin romper
                          a cuatro equipos
```

Y además de eso:

```
el consumidor recibe cambios de FILAS, no hechos de negocio
  → «cambió la columna estado de 3 a 4»
  → y tiene que saber qué significa 4, que es conocimiento tuyo

un hecho de negocio suele tocar varias tablas
  → llegan varios eventos y ninguno es el hecho

se publican también las correcciones y las migraciones
  → una migración que toca 4 millones de filas publica 4 millones de eventos
```

Dónde sí encaja la captura de cambios:

```
sí   replicar hacia un lago o un almacén analítico (clase 112)
     alimentar un índice de búsqueda
     sacar datos de un sistema heredado que no se puede modificar

no   como contrato de eventos de negocio entre equipos
```

Y la forma correcta cuando se quiere garantizar que el evento se publica si y solo si el cambio se guardó: escribir el evento en la misma transacción que el cambio, en una tabla propia, y publicarlo desde ahí. Eso es el patrón de la clase 116, y aquí basta saber que existe y que es distinto de publicar filas.

```
captura de cambios     publica lo que cambió en las tablas
tabla de salida        publica el HECHO que tú redactaste,
                       con la garantía de que se guardó con el cambio
```

Y el tamaño del evento, que decide más de lo que parece:

```
muy pequeño     obliga a consultar; devuelve el acoplamiento
muy grande      lleva datos que nadie usa, y los sensibles viajan
                por donde no deberían
adecuado        lo que necesitan la mayoría de los consumidores conocidos
```

Y sobre los datos personales dentro de eventos, una advertencia con consecuencias legales: un registro conservado es inmutable y difícil de purgar. Publicar identificadores y dejar los datos personales en el origen evita el problema; publicarlos dentro lo garantiza.

Y la lista de comprobación de la clase:

- ■ todos los eventos tienen nombre de hecho en pasado
- ■ ningún «evento» dice al consumidor lo que tiene que hacer
- ■ la envoltura es la misma en toda la organización

- el momento del hecho está separado del de publicación
- el esquema está en un registro y se valida al producir
- los cambios conservan compatibilidad en las dos direcciones
- ningún campo cambia de significado sin cambiar de nombre
- hay catálogo de quién produce y quién consume cada tema
- existe una vista por sujeto con la secuencia completa y sus tiempos
- hay vigilancia de pasos que no ocurren dentro de su plazo
- no se publican filas de tablas como contrato de negocio
- los datos personales no viajan dentro de eventos conservados

Y el cierre que enlaza con la clase siguiente: queda el problema que las clases 113 y 114 han ido aplazando —cómo se garantiza que un hecho se publica exactamente si el cambio se guardó, y cómo se repite una operación sin duplicar su efecto—, y es la materia de la clase 116.

Ejemplo trabajado

CloudShop lleva ocho meses publicando eventos de pedido. Tres equipos consumen. El ejercicio son los cuatro problemas que aparecieron y lo que cada uno obligó a cambiar en el contrato.

Problema 1: un cambio de esquema rompe a cuatro equipos en veinte minutos.

```
cambio     el campo «importe» pasa de céntimos a unidades
motivo     «era confuso»
revisión   aprobada por el equipo productor
consumidores avisados                          0
consumidores que existían                      4
```

```
14:02  se despliega el productor
14:04  facturación emite facturas con importes 100 veces menores
14:11  analítica publica un panel con ingresos del 1 %
14:19  detectado por una persona de finanzas
14:40  revertido
facturas erróneas emitidas                     1.204
rectificativas necesarias                      1.204
```

El esquema seguía validando: el tipo era el mismo y solo cambió el significado, que es el caso que ninguna herramienta detecta.

Las tres correcciones:

```
registro de esquemas                          no            sí
validación de compatibilidad en la canalización no          sí
cambio de significado                  mismo nombre   nombre nuevo
                                                      (importe_centimos)
catálogo de consumidores por tema              no            sí
```

Y la comprobación de compatibilidad detectó, al implantarla, otros seis cambios pendientes de despliegue que habrían roto a alguien.

Problema 2: nadie sabe dónde está el pedido 1421.

```
reclamación de un cliente: «hice el pedido hace 3 días y no llega»
equipos consultados para responder                    6
tiempo hasta poder responder                      2 h 40
causa encontrada    un evento se procesó y el consumidor de
                    preparación estaba parado desde hacía 19 h
quién lo sabía      nadie: un consumidor parado no da error
```

Es la ley 13 en versión de negocio. Lo que se construyó:

```
vista por sujeto: todos los eventos de «pedido/1421» con tiempos
vigilante de plazos: 7 pares de eventos con su tiempo máximo
alerta cuando el segundo no llega
```

```
tiempo hasta responder «¿dónde está?»          2 h 40         40 s
procesos parados detectados por el vigilante      —      11 en 6 meses
de ellos, detectados antes por otro medio         —            2
```

Nueve de once solo se detectaron por el vigilante de plazos.

Problema 3: la captura de cambios que se convirtió en contrato.

El equipo de búsqueda pidió los cambios del catálogo y se le dio acceso al flujo de captura de cambios de la base de datos.

```
lo que recibía            filas de la tabla «productos»
a los 5 meses             3 equipos consumiendo lo mismo
intento de renombrar
una columna interna       bloqueado: rompía a los 3
migración de 4,1 M de
filas para un ajuste      publicó 4,1 M de eventos
                          → el consumidor de búsqueda tardó 9 h en digerirlos
                          → y reindexó todo el catálogo sin motivo
```

La corrección fue publicar hechos redactados, no filas:

```
lo que viaja                          filas de la tabla     producto.publicado
                                                            producto.retirado
                                                            producto.reprecio
esquema interno como contrato               sí                    no
eventos por migración de 4,1 M filas    4,1 millones             0
equipos bloqueados por renombrar             3                    0
```

Y la captura de cambios se conservó para lo que sí encaja: alimentar el lago de la clase 112.

Problema 4: el comando disfrazado.

```
tema publicado           «pedido.debe.enviarse»
consumidor               un solo servicio, el de logística
qué pasó                 logística cambió su proceso y necesitaba dos pasos
                         → pidió al equipo de pedidos que publicara
                           «pedido.debe.empaquetarse» primero
```

El equipo de pedidos estaba tomando decisiones sobre el proceso de logística. El diagnóstico con la prueba del apartado primero:

```
¿está en pasado y describe algo ocurrido?   no: es un imperativo
→ no es un evento
```

```
lo que publica pedidos             pedido.debe.enviarse    pedido.pagado
quién decide los pasos de logística     pedidos              logística
cambios en logística que exigen
tocar pedidos                            todos               ninguno
consumidores del hecho                     1                   3
```

La última fila fue la sorpresa: al publicar el hecho en vez de la orden, dos equipos más lo consumieron sin que nadie tuviera que hacer nada.

El tamaño del evento, decidido preguntando.

```
propuesta inicial       notificación mínima: solo el identificador
consultado a los 3 consumidores conocidos:
  facturación   necesita importe, impuestos y cliente
  analítica     necesita todo lo anterior y el canal
  búsqueda      no consume este evento
decisión        incluir lo que necesitan los dos, y nada más
tamaño medio    1,8 KB
consultas al origen tras el cambio    de 12.000/día a 40/día
```

A los seis meses.

```
registro de esquemas                            no             sí
cambios incompatibles llegados a producción    1 grave      0 (6 bloqueados)
catálogo de productores y consumidores         no             sí
vista por sujeto                                no             sí
tiempo hasta responder «¿dónde está?»        2 h 40          40 s
procesos parados detectados                     0        11 / 6 meses
comandos disfrazados de evento                  3              0
temas que publican filas de tablas              2         0 (1 al lago)
consultas al origen por evento pequeño      12.000/día      40/día
```

La lección que esta clase traslada a la parte 09: los cuatro problemas eran de contrato, no de infraestructura. El registro de la clase 114 funcionaba perfectamente en los cuatro casos. Y el más caro —mil doscientas facturas rectificativas— lo causó un cambio que todas las validaciones automáticas consideraban compatible, porque el tipo del campo no cambió: solo su significado. Contra eso, la única defensa que funcionó fue organizativa: saber quién consume y avisarle.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/115-arquitectura-dirigida-por-eventos-y-contratos/lab.py
```

El laboratorio selecciona el motor de práctica architecture 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 catalogo-eventos 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 diagrama acompañado por decisiones y trade-offs. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye catalogo-eventos 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 cambio de esquema que valida correctamente rompe a los consumidores	Cambió el significado del campo sin cambiar su nombre, y eso ninguna herramienta lo detecta	Si cambia el significado, cambia el nombre; y mantén catálogo de consumidores para avisar.
Nadie puede decir en qué paso está un proceso	Coreografía sin vista por sujeto ni vigilancia de plazos	Identificador de traza y sujeto uniformes en la envoltura, vista por sujeto y alerta cuando un paso esperado no llega a tiempo.
No se puede renombrar una columna interna sin romper a otros equipos	Se publican filas de tablas, así que el esquema interno es contrato público	Publica hechos de negocio redactados; deja la captura de cambios para alimentar el lago.
Una migración de datos genera millones de eventos y satura a los consumidores	Los eventos salen del registro de cambios de la base, no de hechos de negocio	Publica hechos, y si hay que reprocesar, hazlo por un canal separado y anunciado.
El productor tiene que cambiar cada vez que cambia el proceso del consumidor	Está publicando comandos disfrazados de eventos	Publica hechos en pasado; si el mensaje dice qué hacer, va a una cola con destinatario.
Un reproceso del histórico falla porque faltan campos	Se retiró un campo antes de que expirara la retención del registro	En expandir y contraer sobre registros conservados, espera además a que caduquen los mensajes con el formato antiguo.

■ 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 prueba distingue un evento de un comando disfrazado?
2. ¿Por qué un registro conservado exige compatibilidad en las dos direcciones?
3. ¿Qué cambio de esquema es el más peligroso y por qué no lo detecta ninguna herramienta?
4. ¿Qué es obligatorio construir si se elige coreografía?
5. ¿Cuándo encaja la captura de cambios y cuándo no debe usarse?

Referencias

• CloudEvents (2025). Specification — envoltura común: identificador, tipo, origen, sujeto y momento. <https://github.com/cloudevents/spec/blob/main/cloudevents/spec.md>

• Fowler, M. (2025). What do you mean by event-driven? — notificación, transferencia de estado y sus consecuencias. <https://martinfowler.com/articles/201701-event-driven.html>

• Confluent (2025). Schema Registry: compatibility types — compatibilidad hacia atrás, hacia delante y completa. <https://docs.confluent.io/platform/current/schema-registry/fundamentals/avro.html>

• Richardson, C. (2025). Saga and event choreography trade-offs — coreografía frente a orquestación. <https://microservices.io/patterns/data/saga.html>

• Debezium (2025). Change data capture: when to use it — límites de publicar cambios de filas como contrato. <https://debezium.io/documentation/reference/stable/architecture.html>

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

Descargar: Parte 09 en PDF · Manual integral

Anterior	Índice	Siguiente
← 114 · Pub/sub, streams, particiones y orden	Parte 09 · Programa	116 · Sagas, outbox, idempotencia y deduplicación →

Evaluación — 115 Arquitectura dirigida por eventos 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 catalogo-eventos 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.

116 — Sagas, outbox, idempotencia y deduplicación

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 4 Laboratorio: distributed · Estado: EXECUTABLECORE

Propósito

Resolver lo que las clases 113, 114 y 115 fueron aplazando. Son dos problemas distintos y se confunden todo el rato: guardar un cambio y publicar su hecho no se puede hacer de forma atómica, y un consumidor que recibe el mismo mensaje dos veces no debe producir dos efectos. La clase da la solución de cada uno —tabla de salida e idempotencia real, no aproximada— y luego afronta el caso general: cuando una operación abarca varios servicios y no existe ninguna transacción que los cubra.

Resultados de aprendizaje

Al finalizar podrás:

1. Explicar por qué la doble escritura no puede ser atómica y qué pierde cada atajo.
2. Implantar la tabla de salida y su gemela en el consumidor.
3. Elegir la técnica de idempotencia adecuada a cada efecto, incluidos los externos.
4. Diseñar una secuencia con compensaciones, y ordenar los pasos por reversibilidad.
5. Enunciar qué se consigue de verdad: efecto una sola vez, no entrega una sola vez.

Conceptos centrales

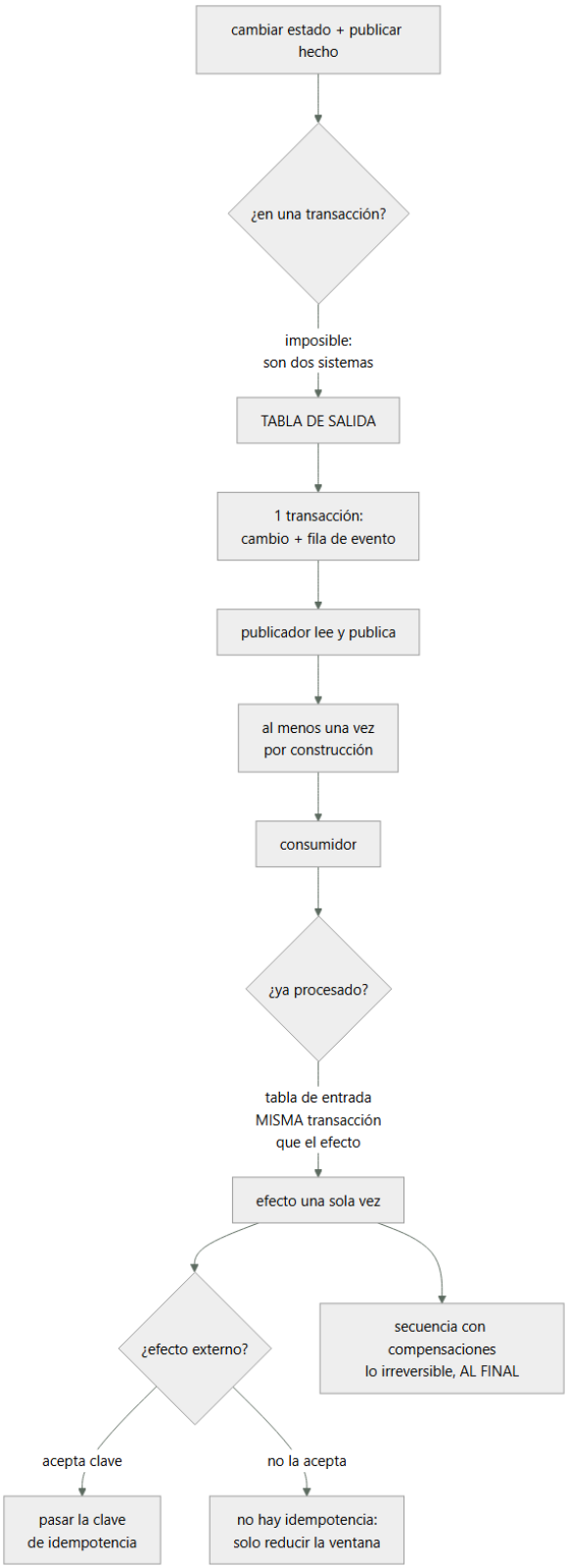
Concepto	Comprensión verificable
doble escritura	Guardar en la base y publicar en el intermediario como dos operaciones. Entre las dos puede fallar todo, y no hay transacción que las cubra.
tabla de salida	El evento se escribe en la misma transacción que el cambio, en una tabla propia. Un publicador la lee y publica después.
tabla de entrada	El consumidor registra el identificador de lo procesado en la misma transacción que el efecto. Es lo que hace la deduplicación fiable.
idempotencia	Repetir la operación no añade efecto. No es «detectar duplicados»: es que repetir sea inofensivo.
compensación	Operación de negocio que anula el efecto de un paso anterior. No es una vuelta atrás: es un hecho nuevo y visible.
efecto una sola vez	Entrega al menos una vez más proceso idempotente. Es lo máximo alcanzable, y es suficiente.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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 doble escritura no tiene atajo

El problema aparece en cuanto un servicio guarda algo y publica un hecho:

```
BEGIN
  UPDATE pedidos SET estado='pagado' WHERE id=1421
COMMIT
```

```
publicar("pedido.pagado", 1421)      ← si falla aquí, el hecho no existe
```

Y el orden inverso no mejora nada:

```
publicar("pedido.pagado", 1421)
BEGIN … COMMIT                        ← si falla aquí, el hecho es MENTIRA
```

Las dos son la misma imposibilidad: son dos sistemas distintos y no hay transacción que los abarque. Y los atajos habituales pierden algo cada uno:

```
publicar dentro de la transacción
  el intermediario no participa; si la transacción se deshace,
  el mensaje ya salió

reintentar la publicación en memoria
  si el proceso muere, se pierde

transacción distribuida de dos fases
  existe, y casi ningún intermediario moderno la soporta;
  y bloquea recursos mientras dura

no hacer nada y esperar que no pase
  pasa; con 10.000 operaciones al día, varias veces al mes
```

La solución es la tabla de salida, y consiste en no tener dos sistemas en el momento crítico:

```
BEGIN;
  UPDATE pedidos SET estado='pagado' WHERE id=1421;
  INSERT INTO salida (id, tipo, sujeto, datos, creado)
  VALUES ('e-9f2c', 'pedido.pagado', 'pedido/1421', '{…}', now());
COMMIT;
```

Y después, un publicador lee la tabla y publica:

```
lee las filas no publicadas, en orden
publica
marca como publicadas

si muere entre publicar y marcar → publica otra vez
→ AL MENOS UNA VEZ, por construcción
→ y por eso el consumidor tiene que ser idempotente: apartado tercero
```

Lo operativo, que decide si esto funciona:

```
ORDEN         leer por identificador creciente; publicar en orden por sujeto
LIMPIEZA      borrar lo publicado con cierta antigüedad, o la tabla crece
              sin fin → y no borrarlo no da ningún error (ley 13)
RETRASO       vigilar la antigüedad de la fila no publicada más vieja
              → es la señal, igual que en la clase 113
COMPETENCIA   varios publicadores requieren bloqueo o reparto por sujeto
```

Y dos formas de leer la tabla:

```
sondeo periódico          simple, y añade latencia y carga
captura de cambios sobre
la tabla de salida        latencia baja y sin carga de sondeo
                          → aquí sí encaja la captura de cambios (clase 115),
                            porque la tabla la escribes tú a propósito
```

Y la simétrica en el consumidor, la tabla de entrada, que es lo que hace que la deduplicación sea fiable y no aproximada:

```
BEGIN;
  INSERT INTO procesados (id_mensaje) VALUES ('e-9f2c');  -- falla si ya está
  UPDATE inventario SET reservado = reservado + 1 WHERE sku='X';
COMMIT;
```

Si la inserción falla por duplicado, la transacción entera se deshace y no hay segundo efecto. La clave está en que las dos cosas estén en la misma transacción: comprobar antes y actuar después deja una ventana en la que dos consumidores pasan la comprobación.

2. Idempotencia de verdad

Idempotente no significa «detecta duplicados»: significa que repetir no añade efecto. Las técnicas, de más robusta a menos:

```
1. OPERACIÓN NATURALMENTE IDEMPOTENTE
   estado = 'pagado'            repetir no cambia nada
   frente a  saldo = saldo - 10  repetir descuenta otra vez
   → siempre que se pueda, expresar el efecto como asignación
     y no como incremento

2. RESTRICCIÓN ÚNICA SOBRE LA CLAVE NATURAL
   UNIQUE (pedido_id, tipo_movimiento)
   → la base impide el segundo efecto, sin código
   → es la más fuerte porque no depende de que nadie se acuerde

3. ESCRITURA CONDICIONAL POR VERSIÓN
   UPDATE … WHERE id=1421 AND version=7
   → si otro ya aplicó, afecta a 0 filas y se detecta
   → sirve también contra el desorden: descarta lo viejo (clase 113)

4. TABLA DE DEDUPLICACIÓN CON CLAVE
   la del apartado anterior, en la misma transacción que el efecto
   → general y aplicable a casi todo
   → cuidado con su caducidad: ver más abajo
```

Y el detalle de la cuarta que casi siempre se hace mal: la caducidad de la tabla de deduplicación es una ventana de corrección, no de limpieza.

```
caducidad de 24 h
→ un mensaje reentregado a las 30 h se procesa otra vez
→ ¿puede eso ocurrir? con reprocesos de la clase 114, sí
```

La regla: la caducidad debe superar la retención del registro y cualquier ventana de reproceso previsible.

Y de dónde sale la clave, porque generarla mal invalida todo:

```
bien   el identificador del mensaje, si el productor lo genera una vez
       y lo repite en los reenvíos
bien   una clave derivada del negocio: «pago-1421-intento-3»
mal    generarla en el consumidor  → cada intento tiene una distinta
mal    usar la marca de tiempo     → cambia en cada reenvío
```

Los efectos externos, que es el caso duro y el que se pasa por alto:

```
cobrar en un proveedor de pago
enviar un correo
llamar a la API de un transportista
```

Aquí la idempotencia no depende de ti:

```
si el proveedor acepta una clave de idempotencia
  → pásasela, derivada del negocio, y él deduplica
  → esto es lo que hace correcta la operación, no tu tabla

si NO la acepta
  → no hay idempotencia posible
  → solo se puede reducir la ventana: registrar «voy a llamar» antes,
    y conciliar después con el estado del proveedor
```

Y la conciliación de la última línea es trabajo real y periódico: comparar lo que crees haber hecho con lo que el proveedor dice que ocurrió. Casi nadie la monta hasta el primer descuadre.

Y la conclusión que ordena la parte 09 entera:

```
entrega al menos una vez  (clases 113 y 114, y no hay alternativa)
+ proceso idempotente     (esta clase)
= EFECTO UNA SOLA VEZ
```

3. Cuando no hay transacción que abarque

Un pedido toca inventario, pago, envío y notificación, y cada uno vive en su servicio con su base. No existe ninguna transacción que abarque los cuatro.

Lo que sí existe es una secuencia de transacciones locales, cada una con su compensación:

```
paso 1  reservar inventario        compensar: liberar la reserva
paso 2  cobrar                     compensar: devolver
paso 3  crear el envío             compensar: cancelar el envío
paso 4  notificar al cliente       compensar: NO SE PUEDE
```

Y de ahí las tres verdades incómodas de este patrón:

```
1. COMPENSAR NO ES DESHACER
   una devolución no es que el cobro no ocurriera: es otro hecho,
   visible para el cliente y para la contabilidad

2. HAY PASOS QUE NO SE COMPENSAN
   un correo enviado, una llamada a un tercero, un envío ya recogido
   → hay que ORDENAR los pasos: lo irreversible, lo más tarde posible

3. NO HAY AISLAMIENTO
   entre el paso 1 y el 4, otro proceso ve el estado intermedio
   → un inventario reservado y un pago pendiente son visibles
```

La tercera es la que produce errores raros, y la defensa es hacer explícito el estado intermedio:

```
estados de negocio, no booleanos
  pendiente_de_pago, reservado, confirmado, cancelado
y quien consulte tiene que saber tratarlos
```

Y el tiempo, que es la parte que se olvida: un paso puede no responder nunca. Cada paso necesita un plazo y una decisión al vencerlo:

```
si el paso 2 no responde en 5 min
  → ¿se compensa el paso 1 o se sigue esperando?
  → hay que decidirlo por adelantado, y por paso
```

Y el peor caso de todos, que hay que mirar de frente: la compensación también puede fallar. La respuesta honesta no es más automatismo:

```
reintentar la compensación, con límite
y si sigue fallando → cola de intervención humana, con todo el contexto
→ una secuencia atascada tiene que ser VISIBLE, no silenciosa
```

Cómo se coordina, que enlaza con la clase 115:

```
COREOGRAFIADA   cada servicio reacciona al hecho anterior
                + sin componente central
                − nadie conoce la secuencia entera; compensar en cadena
                  es difícil de seguir

ORQUESTADA      un componente conoce los pasos y las compensaciones
                + la secuencia está escrita en un sitio
                + los plazos y los atascos tienen dueño
                − acopla a los participantes
```

Y el criterio: si hay compensaciones y plazos, orquestar. La coreografía es buena para propagar hechos; para procesos con vuelta atrás, tener la secuencia en un sitio vale más que la independencia. Es la clase 119.

Y la lista de comprobación de la clase:

- no hay ninguna doble escritura sin tabla de salida
- el publicador vigila la antigüedad de la fila más vieja sin publicar
- la tabla de salida se limpia, y ese trabajo está programado
- el consumidor registra lo procesado en la MISMA transacción que el efecto
- la caducidad de la deduplicación supera la retención y la ventana de reproceso
- la clave de idempotencia la genera el productor y se repite en los reenvíos
- los efectos externos reciben clave de idempotencia; si no la aceptan, hay conciliación periódica
- los pasos irreversibles están al final de la secuencia
- cada paso tiene plazo y decisión al vencerlo
- los estados intermedios son estados de negocio y quien consulta los conoce
- una compensación que falla acaba en una cola visible, no en silencio

Y el cierre que enlaza con la clase siguiente: todo esto supone procesos que se ejecutan en algún sitio con control sobre su tiempo de vida. Cuando ese sitio es una función efímera con límites de duración, concurrencia y arranque, varias de estas garantías se complican, y es la materia de la clase 117.

Ejemplo trabajado

CloudShop arrastra tres problemas de las clases anteriores: eventos que no se publicaron, cobros duplicados y pedidos que se quedan a medias. Los tres se resuelven aquí, y el tercero enseña lo que cuesta no tener transacciones.

Problema 1: los eventos que nunca existieron.

Una auditoría comparó pedidos pagados con eventos publicados:

```
pedidos con estado 'pagado' en 6 meses            412.880
eventos 'pedido.pagado' publicados                412.301
faltan                                                579
```

Quinientos setenta y nueve pedidos cobrados que facturación nunca supo que existían. Y ninguno produjo un error: la transacción de la base había ido bien y la publicación había fallado después.

```
causas de los 579
  reinicio del proceso entre confirmar y publicar        341
  intermediario no disponible unos segundos              203
  errores de red                                          35
```

Con tabla de salida:

```
eventos perdidos en 6 meses                    579               0
latencia de publicación (sondeo cada 2 s)      inmediata        ~1,1 s
latencia con captura de cambios sobre la tabla     —            ~90 ms
filas acumuladas sin limpieza a los 2 meses        —        41 millones
```

La última fila es el trabajo nuevo que apareció, y tardó dos meses en verse porque no daba ningún error. Se programó la limpieza y una alerta por antigüedad de la fila más vieja sin publicar, que en cuatro meses detectó tres publicadores parados.

Problema 2: los cobros duplicados, de la clase 113.

La clase 113 dejó residuo tras ajustar la invisibilidad:

```
duplicados residuales                          0-2 por semana
causa                          reentrega tras muerte del consumidor
```

El primer intento fue una tabla de deduplicación consultada antes de cobrar:

```
SELECT … FROM procesados WHERE id = ?     -- si existe, salir
… cobrar …
INSERT INTO procesados …
```

Y siguió habiendo duplicados, menos:

```
duplicados por semana                      0-2          0-1                0
```

La razón es la del apartado primero: entre la consulta y la inserción caben dos consumidores. Con la inserción y el efecto local en la misma transacción, el problema local desapareció.

Pero el cobro es un efecto externo, y ahí la transacción local no alcanza:

```
la transacción local puede deshacerse DESPUÉS de que el proveedor cobró
→ y el proveedor ya cobró
```

La corrección real fue pasar la clave al proveedor:

```
cobros duplicados en 6 meses                47                    0
lo que hace el proveedor con el repetido   cobra      devuelve el cobro original
```

Y para el proveedor de mensajería, que no acepta clave, se montó la conciliación:

```
comparación diaria entre envíos que creemos haber creado y los suyos
descuadres encontrados el primer día                        94
descuadres al mes 6                                        0-3
tiempo hasta detectar un descuadre     de nunca a menos de 24 h
```

Problema 3: los pedidos a medias.

```
pedidos en estado inconsistente al mes                     31
  inventario reservado y pago fallido                      19
  pago hecho y envío no creado                              9
  envío creado y pedido cancelado                           3
cómo se detectaban       reclamaciones de clientes
cómo se arreglaban       a mano, por el equipo de soporte
```

La tercera línea es la peor: envío creado y pedido cancelado significa mercancía que salió de un pedido que no existe.

Se escribió la secuencia con compensaciones y se reordenó por reversibilidad:

```
ORDEN ORIGINAL                  ORDEN CORREGIDO
1 crear pedido                  1 crear pedido (pendiente)
2 notificar al cliente          2 reservar inventario
3 reservar inventario           3 cobrar
```

```
4 cobrar                          4 crear envío
5 crear envío                     5 notificar al cliente   ← irreversible, al final
```

El cambio de la notificación del segundo al quinto lugar eliminó por sí solo una categoría entera de problemas:

```
correos de confirmación de pedidos que luego fallaban
  antes:  118 en 6 meses
  después:  0
```

Y los plazos por paso, con su decisión:

```
paso          plazo     al vencer
reservar      30 s      reintentar 3 veces, luego cancelar
cobrar         5 min    compensar la reserva y marcar pendiente_de_pago
crear envío   15 min    reintentar; a los 15 min, cola de intervención
notificar      1 h      reintentar; no bloquea el pedido
```

Y lo que ocurrió con las compensaciones que fallan:

```
compensaciones ejecutadas en 6 meses              1.412
  correctas                                       1.397
  fallidas tras 3 intentos                           15
  → a cola de intervención humana con todo el contexto
tiempo medio de resolución de esas 15            22 min
pedidos inconsistentes detectados por clientes       0
```

Quince casos en seis meses que una persona resolvió en veintidós minutos de media, en lugar de treinta y uno al mes descubiertos por reclamaciones.

Y el estado intermedio visible, que causó un incidente propio.

```
síntoma   el panel de existencias mostraba menos stock del real
causa     las reservas de secuencias en curso se contaban como vendidas
          y algunas se compensaban minutos después
efecto    se dejaron de vender 340 unidades que estaban disponibles
```

Es la falta de aislamiento del apartado tercero. La corrección fue hacer explícitos los estados:

```
estados de inventario             disponible/vendido   disponible /
                                                       reservado_en_curso /
                                                       vendido
el panel muestra                  disponible           disponible +
                                                       reservado_en_curso,
                                                       separados
unidades no vendidas por el error     340                 0
```

A los seis meses.

```
eventos perdidos                          579 / 6 meses        0
cobros duplicados                          47 / 6 meses        0
descuadres con el transportista           no se sabía         0-3 / mes
pedidos inconsistentes                     31 / mes           0
correos de confirmación erróneos          118 / 6 meses        0
unidades bloqueadas por estado intermedio     340              0
casos que requieren intervención humana        —          15 / 6 meses
tiempo medio de esa intervención               —            22 min
trabajo nuevo: limpieza de la tabla de salida  —          programado
```

La lección que esta clase traslada a la parte 09: ninguno de los tres problemas se resolvió con una garantía del intermediario. Se resolvieron moviendo el punto donde ocurre la atomicidad: la tabla de salida mete el hecho en la misma transacción que el cambio, la tabla de entrada mete la deduplicación en la misma transacción que el efecto, y la clave de idempotencia traslada el mismo truco al proveedor externo. Y donde ese truco no se puede aplicar —el transportista que no acepta clave, la compensación que falla— la respuesta honesta no es más automatismo: es conciliar periódicamente y hacer visible lo que necesita una persona.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/116-sagas-outbox-idempotencia-y-
deduplicacion/lab.py
```

El laboratorio selecciona el motor de práctica distributed 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 transaccion-distribuida 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 traza de consistencia, reintento o fallo parcial. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye transaccion-distribuida 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
Hay cambios guardados cuyo evento nunca se publicó, sin ningún error	Doble escritura: la base y el intermediario son dos sistemas y no hay transacción que los cubra	Tabla de salida: el evento se escribe en la misma transacción que el cambio y un publicador lo envía después.
La deduplicación reduce los duplicados pero no los elimina	Se comprueba antes y se registra después: entre las dos caben dos consumidores	Registra el identificador procesado en la misma transacción que el efecto, y deja que la restricción única falle.
Se cobra dos veces aunque el consumidor sea idempotente localmente	El efecto está en un sistema externo y la transacción local no lo abarca	Pasa una clave de idempotencia derivada del negocio al proveedor; si no la acepta, monta conciliación periódica.
Se envía una confirmación de un pedido que luego se cancela	Un paso irreversible está antes que pasos que pueden fallar	Ordena la secuencia por reversibilidad y deja lo irreversible al final.
Otros procesos ven datos a medias mientras la secuencia avanza	Una secuencia con compensaciones no tiene aislamiento	Haz explícitos los estados intermedios como estados de negocio y adapta lo que consulta.
La tabla de salida crece sin límite	Ley 13: no limpiar lo publicado no produce ningún error	Programa la limpieza y alerta por antigüedad de la fila más vieja sin publicar.

■ 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 guardar y publicar no puede ser atómico, y qué pierde cada atajo?
2. ¿Por qué la deduplicación debe estar en la misma transacción que el efecto?
3. ¿De dónde debe salir la clave de idempotencia y por qué no vale generarla en el consumidor?
4. ¿Qué se hace cuando un proveedor externo no acepta clave de idempotencia?
5. ¿Qué tres cosas distinguen una compensación de una vuelta atrás?

Referencias

- Richardson, C. (2025). Transactional outbox pattern — publicar el hecho en la misma transacción que el cambio. <https://microservices.io/patterns/data/transactional-outbox.html>
- Richardson, C. (2025). Saga pattern — transacciones locales con compensación, coreografiadas y orquestadas. <https://microservices.io/patterns/data/saga.html>
- Stripe (2025). Idempotent requests — clave de idempotencia en un efecto externo y su ventana. <https://docs.stripe.com/api/idempotentrequests>
- Helland, P. (2012). Idempotence is not a medical condition — idempotencia como propiedad del efecto, no de la entrega. <https://queue.acm.org/detail.cfm?id=2187821>
- Garcia-Molina, H. y Salem, K. (1987). Sagas — el artículo original sobre transacciones largas con compensación. <https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 09 en PDF · Manual integral

Anterior	Índice	Siguiente
← 115 · Arquitectura dirigida por eventos y contratos	Parte 09 · Programa	117 · Serverless: límites, cold starts y concurrencia →

Evaluación — 116 Sagas, outbox, idempotencia y deduplicación

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 transaccion-distribuida 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.

117 — Serverless: límites, cold starts y concurrencia

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 4 Laboratorio: serverless · Estado: EXECUTABLECORE

Propósito

Ejecutar código sin gestionar servidores, entendiendo que todo lo que hay que aprender se deduce de una sola renuncia: el proceso puede desaparecer y reaparecer, y una instancia atiende una petición a la vez. De ahí salen la aritmética de la concurrencia, que es también la del coste; el arranque en frío, que importa en unos casos y en otros no; los límites que obligan a partir el trabajo; y el conflicto con lo que la clase 109 dejó claro sobre las conexiones.

Resultados de aprendizaje

Al finalizar podrás:

1. Calcular la concurrencia y el coste a partir del tráfico y la duración.
2. Reducir el arranque en frío y decidir si en este caso importa.
3. Diseñar dentro de los límites de duración, memoria y estado.
4. Proteger las dependencias de la concurrencia, y a otras funciones entre sí.
5. Reconocer cuándo este modelo deja de ser la elección adecuada.

Conceptos centrales

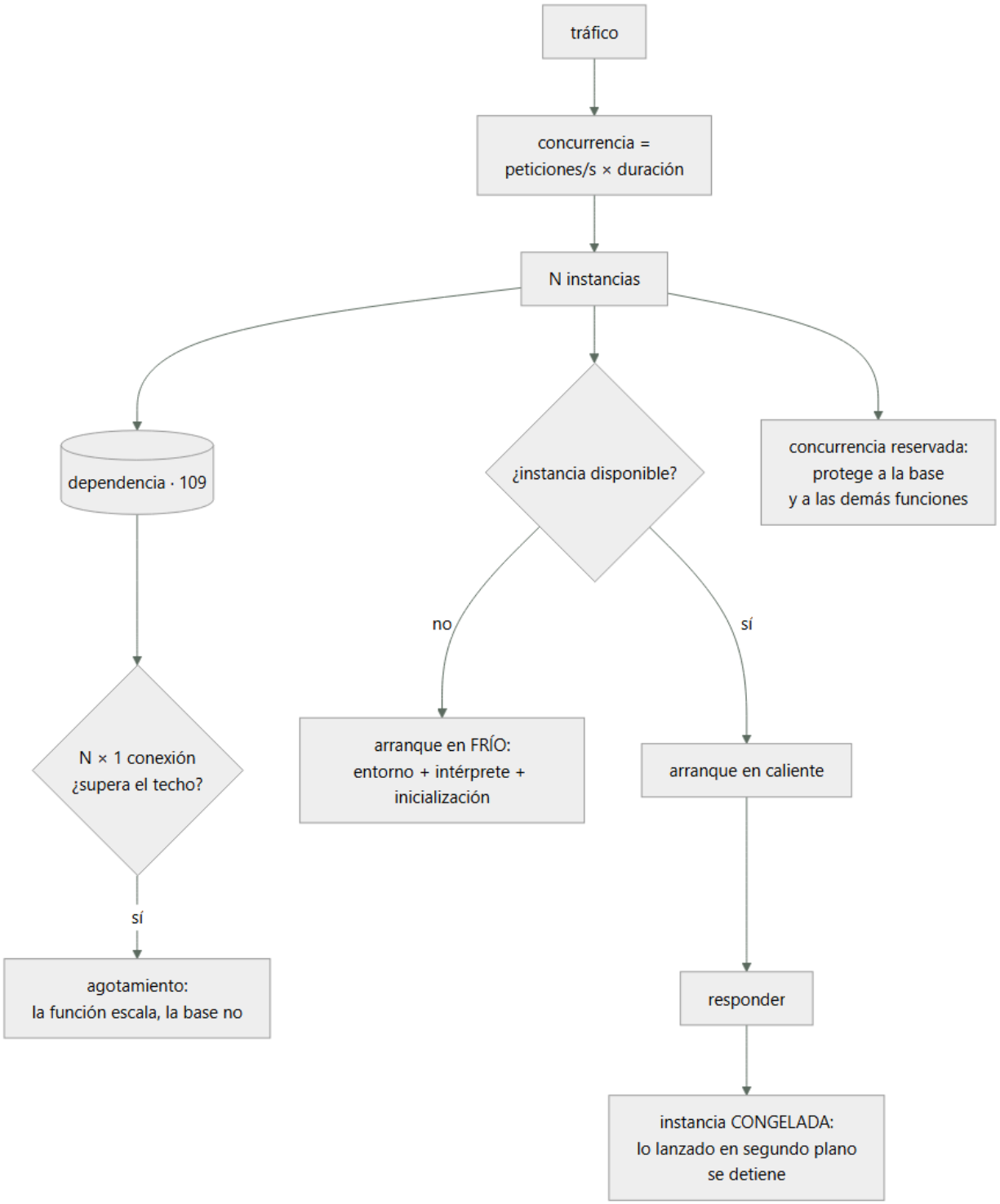
Concepto	Comprensión verificable
una petición por instancia	Cada ejecución ocupa su instancia entera. La concurrencia es el número de instancias vivas, no de hilos.
arranque en frío	Tiempo añadido cuando hay que crear una instancia nueva: preparar el entorno, cargar el intérprete e inicializar el código.
concurrencia reservada	Límite máximo de instancias de una función. Sirve para dos cosas opuestas: garantizarle capacidad y evitar que se coma la de los demás.
congelación tras responder	Al devolver la respuesta, la instancia se congela. El trabajo lanzado en segundo plano se detiene a mitad y puede reanudarse mucho después.
memoria acoplada al procesador	En la mayoría de las plataformas, la memoria asignada determina también la potencia. Subir memoria puede abaratar, no encarecer.
punto de cruce de coste	Nivel de uso a partir del cual pagar por ejecución sale más caro que mantener capacidad encendida.
límite de tiempo	Duración máxima de una ejecución. Es un límite duro que obliga a partir el trabajo o a cambiar de modelo.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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 aritmética que lo explica todo

En un servidor tradicional, un proceso atiende muchas peticiones a la vez. Aquí no:

```
una instancia = una petición a la vez
```

Y de eso sale la única fórmula que hay que recordar:

```
concurrencia = peticiones por segundo × duración media

800 peticiones/s × 0,120 s   →    96 instancias
800 peticiones/s × 1,400 s   → 1.120 instancias
```

Mismo tráfico, doce veces más instancias, solo por tardar más. Y eso decide tres cosas a la vez:

```
cuántas instancias hacen falta
cuántas conexiones abre contra su base de datos    (apartado cuarto)
cuánto cuesta                                       (memoria × tiempo)
```

El coste se factura por memoria asignada y tiempo, y de ahí sale un resultado contraintuitivo:

```
función con 512 MB, tarda 1.200 ms   → 0,6 GB·s
la misma con 1.024 MB, tarda 480 ms  → 0,49 GB·s
```

Más memoria, menos coste, porque la memoria lleva procesador acoplado. La consecuencia práctica: probar tres o cuatro tamaños y medir, en vez de asignar el mínimo por prudencia. Lo que casi nunca compensa es el mínimo.

Y el punto de cruce, que hay que calcular antes de mover nada:

```
uso esporádico o muy variable   → pagar por ejecución sale barato
uso alto y constante            → una instancia encendida sale más barato

ejemplo con números redondos:
  10 peticiones/s constantes, 200 ms, 512 MB
  → 1 GB·s por segundo, 24 h al día
  → comparado con dos contenedores pequeños siempre encendidos,
    el segundo suele ganar por un factor de 3 a 10
```

Y la conclusión que conviene tener escrita: este modelo es excelente para lo irregular y caro para lo constante. No es una cuestión de gusto: es una cuenta.

Y el otro lado, que a menudo compensa lo anterior:

```
lo que se deja de pagar: capacidad ociosa, parcheado, escalado, guardia
de la capa de ejecución
→ y eso no aparece en la factura del cómputo
```

2. El arranque en frío, y cuándo importa

Cuando no hay instancia libre, hay que crear una, y eso tiene partes con costes muy distintos:

```
preparar el entorno            plataforma; poco control
cargar el intérprete o máquina virtual   depende del lenguaje
inicializar TU código          dependencias, clientes, configuración
primera ejecución              caminos aún no optimizados
```

Y el tercero es el que suele dominar y el único que controlas del todo:

```
leer configuración de un servicio remoto al arrancar    +300 a 900 ms
crear clientes de SDK dentro del manejador     se repite en CADA petición
un paquete de 180 MB frente a uno de 12 MB     diferencia grande
```

Las correcciones, por orden de eficacia:

```
1. inicializar FUERA del manejador
   clientes, conexiones y configuración se crean una vez por instancia
   y se reutilizan en todas las peticiones que atienda

2. reducir el paquete
   quitar dependencias que no se usan; empaquetar solo lo necesario

3. no consultar servicios remotos en la inicialización
   o cachear el resultado con su caducidad

4. instancias mínimas siempre listas
   → resuelve el frío y se paga esté o no en uso
```

Y la pregunta previa a todo esto, que ahorra mucho trabajo inútil:

```
¿este arranque en frío lo sufre un usuario?
  sí, es una API síncrona → importa, y afecta al percentil 99
  no, procesa una cola     → no importa: 800 ms más en un trabajo
                             asíncrono no los nota nadie
```

Y dos matices que evitan diagnósticos equivocados:

```
la proporción de arranques en frío baja con el tráfico
  → con tráfico alto y constante, casi todo está caliente
  → el problema es de tráfico irregular, no de volumen

una instancia se recicla cada cierto tiempo aunque haya tráfico
```

```
  → siempre habrá algún frío; el objetivo no es cero
```

Y el efecto secundario útil de inicializar fuera del manejador: la instancia conserva estado entre peticiones. Eso permite reutilizar conexiones y cachés locales, y a la vez es una fuente de errores:

```
variable global que acumula datos entre peticiones
→ crece hasta agotar la memoria
→ y si guarda datos de un usuario, el siguiente los ve (clase 111)
```

3. Los límites que obligan a diseñar distinto

```
DURACIÓN MÁXIMA        de minutos a unas pocas horas según plataforma
                       → un trabajo que puede crecer no cabe
MEMORIA                acoplada al procesador
TAMAÑO DEL MENSAJE     límites de entrada y de respuesta
DISCO TEMPORAL         existe, es efímero y se comparte entre peticiones
                       de la MISMA instancia
SIN ESTADO LOCAL       nada sobrevive garantizado entre ejecuciones
SIN TRABAJO DE FONDO   la instancia se congela al responder
```

La última merece el detalle porque produce errores desconcertantes:

```
return respuesta            ← la instancia se CONGELA aquí
// … la tarea lanzada en segundo plano se detiene a mitad
// … y puede reanudarse minutos después, en la siguiente petición
// … o no reanudarse nunca
```

Síntomas típicos: registros que aparecen tarde y mezclados con otra petición, métricas que se pierden, escrituras que a veces se completan. La regla es que todo termine antes de responder; lo que no pueda, va a una cola.

Y el límite de duración obliga a partir el trabajo, que es un patrón que conviene conocer:

```
mal   una función que procesa 4 millones de filas
bien  una función que procesa un lote y encola el resto
      con marcador de posición y comprobación de tiempo restante

if tiempo_restante() < 20s:
    encolar(siguiente_marcador); return
```

Y el disco temporal, que se comparte entre peticiones de la misma instancia:

```
escribir siempre con nombre único por petición
y borrar al terminar
→ si no, una petición lee el fichero de la anterior
```

Los reintentos automáticos, que es lo que enlaza esta clase con la 116. En las invocaciones asíncronas y en las que consumen de una cola o un registro, la plataforma reintenta sola:

```
fallo de la función        → se reintenta, típicamente dos veces más
tiempo agotado             → se reintenta; y la ejecución anterior
                             puede seguir viva un rato
agotados los intentos      → destino de fallidos, si está configurado
```

De ahí que la conclusión de la clase 116 se aplique aquí con más fuerza: todo manejador asíncrono tiene que ser idempotente, porque no hace falta que nadie lo reintente a mano; lo hace la plataforma.

4. Concurrencia: proteger a la base y a las demás

La función escala en segundos hasta miles de instancias. Sus dependencias no.

```
1.200 instancias concurrentes
× 1 conexión cada una
= 1.200 conexiones contra una base con techo de 400
```

Es exactamente el incidente de la clase 109, con un agravante: aquí la escalada es mucho más rápida y no hay agrupador local que valga, porque cada instancia es un proceso aparte.

Las tres respuestas, y suelen hacer falta las tres:

```
agrupador externo                          clase 109
concurrencia reservada en la función       el límite duro que la contiene
reutilizar la conexión entre peticiones    inicializando fuera del manejador
```

Y la concurrencia reservada hace dos trabajos opuestos que conviene distinguir:

```
TECHO      esta función no pasará de N instancias
           → protege a su base de datos
```

```
            → y protege a las demás funciones de la cuenta

SUELO       esta función tiene N garantizadas
            → nadie más se las puede quitar
```

La segunda existe porque la concurrencia suele ser un recurso compartido de la cuenta o el proyecto, y ahí está el incidente clásico:

```
una función de procesamiento por lotes se dispara
consume toda la concurrencia disponible
→ la API de cara al cliente empieza a recibir limitaciones
→ y el incidente parece de la API, que no ha cambiado nada
```

Es un compartimento estanco, exactamente el mismo razonamiento que la clase 060 aplicó a los hilos y las conexiones: aislar para que un consumidor no se lleve la capacidad de todos.

Y el otro efecto de poner techo, que hay que aceptar conscientemente: cuando se alcanza, se rechazan peticiones. En una cola eso solo retrasa; en una API síncrona es un error visible. Por eso el techo se calcula con la capacidad de la dependencia, no con un número redondo.

Y lo que hay que vigilar:

```
concurrencia usada frente al límite
limitaciones por segundo
proporción de arranques en frío
duración, p50 y p99
errores y reintentos por origen
coste por millón de invocaciones, por función
```

Cuándo este modelo no es la elección, que conviene decir con claridad:

```
tráfico alto y constante          sale caro; ver el punto de cruce
trabajos largos                   no caben en el límite de duración
latencia crítica en el p99        el frío es difícil de eliminar del todo
inicialización muy pesada         se paga en cada instancia nueva
necesidad de estado local         no hay
conexiones persistentes salientes vive mal con instancias efímeras
```

Y la combinación que suele ganar: lo irregular y por eventos en funciones; lo constante y de cara al cliente en contenedores (partes 05 y 06). No es una decisión de todo o nada.

Y la lista de comprobación de la clase:

- ■ está calculada la concurrencia con tráfico × duración
- ■ se ha probado con varios tamaños de memoria y se ha medido el coste
- ■ está calculado el punto de cruce frente a capacidad encendida
- ■ los clientes y conexiones se crean fuera del manejador
- ■ no hay consultas remotas en la inicialización
- ■ está decidido si el arranque en frío afecta a alguien
- ■ nada se lanza en segundo plano después de responder
- ■ los trabajos largos se parten con marcador y control de tiempo restante
- ■ el disco temporal usa nombres únicos por petición
- ■ todo manejador asíncrono es idempotente (clase 116)
- ■ hay concurrencia reservada como techo, calculada con la dependencia
- ■ las funciones críticas tienen concurrencia garantizada

Y el cierre que enlaza con la clase siguiente: estas funciones y servicios acaban expuestos como una API que consumen otros, dentro y fuera de la organización. Cómo se publica, se limita, se versiona y se cobra es la materia de la clase 118.

Ejemplo trabajado

CloudShop mueve tres cargas a funciones. Una encaja perfectamente, otra provoca dos incidentes y la tercera se devuelve a contenedores tras hacer la cuenta. Los tres casos son la misma decisión con datos distintos.

Caso 1: procesar imágenes al subirlas. Encaja.

```
tráfico                   irregular: 0 la mayor parte del día,
                          picos de 400/min al cargar catálogos
duración                  1,8 s por imagen
latencia percibida        ninguna: es asíncrono

capacidad para el pico              4 instancias siempre    automática
coste mensual                            310 €               19 €
```

```
tiempo de respuesta al pico          escalado en 90 s       inmediato
arranques en frío                           —              38 % de ejecuciones
efecto de los arranques en frío             —                ninguno
```

Treinta y ocho por ciento de arranques en frío y no importa, porque es la pregunta del apartado segundo: nadie espera.

Y el ajuste de memoria, que sorprendió al equipo:

```
                  512 MB     4,1 s     2,10           100 %
                1.024 MB     1,8 s     1,84            88 %
                2.048 MB     1,1 s     2,25           107 %
```

El mínimo no era el más barato; el punto óptimo estaba en el medio.

Caso 2: consumir eventos de pedido. Dos incidentes.

Incidente A: 1.400 conexiones contra una base de 400.

```
11:20  llega un lote de 90.000 eventos acumulados
11:20  la función escala a 1.400 instancias en 40 s
11:21  la base rechaza conexiones
11:21  la parte SÍNCRONA del sistema también falla: comparten base
11:34  se para la función a mano
```

La cuenta que no se había hecho:

```
concurrencia = 3.000 mensajes/s × 0,47 s = 1.410 instancias
conexiones                                  1.410
techo de la base                              400
```

```
conexión creada                       en el manejador  fuera, reutilizada
agrupador externo                         no             sí
concurrencia reservada                    ninguna        60
conexiones máximas                        1.410          24
tiempo en digerir 90.000 eventos          fallo         11 min
efecto sobre la parte síncrona            caída           ninguno
```

Sesenta instancias tardan once minutos en lugar de fallar en cuarenta segundos. El techo no ralentizó el sistema: lo hizo posible.

Incidente B: una función se come la concurrencia de la cuenta.

```
09:05  un trabajo de reindexación se dispara por un error de configuración
09:06  consume 980 de las 1.000 unidades de concurrencia de la cuenta
09:06  la API de cara al cliente empieza a recibir limitaciones
09:06  errores del 34 % en el sitio web
09:22  se identifica; la API no había cambiado nada
```

Dieciséis minutos buscando en el sitio equivocado.

```
concurrencia garantizada a la API          no          200 reservadas
techo de la función de reindexación        no          50
alerta por concurrencia usada > 70 %       no             sí
incidentes por vecino ruidoso           1 / 4 meses       0
```

El trabajo de fondo que se congelaba.

Y un problema que costó tres semanas de diagnóstico:

```
síntoma   el 2 % de los eventos no llegaba a analítica
          y sus registros aparecían mezclados con otra petición
causa     el envío a analítica se lanzaba en segundo plano
          y el manejador respondía sin esperarlo
          → la instancia se congelaba a mitad del envío
```

```
envío a analítica                    en segundo plano   antes de responder
eventos perdidos                          2 %              0 %
registros mezclados entre peticiones      sí               no
duración añadida                           —              +14 ms
```

Catorce milisegundos por hacerlo bien.

Caso 3: la API de catálogo. Se hace la cuenta y se devuelve a contenedores.

```
tráfico                      1.100 peticiones/s constantes, 24 h
duración                     90 ms
memoria                      512 MB
concurrencia                 99 instancias permanentes
```

```
coste mensual de cómputo                 4.180 €           390 €
latencia p50                              38 ms            31 ms
latencia p99                             410 ms            84 ms
causa del p99                      arranques en frío         —
instancias mínimas para evitarlo    +1.900 €/mes            —
```

Diez veces más caro y con peor percentil 99. Es tráfico constante, que es justo donde este modelo pierde. Se devolvió a contenedores sin discusión, porque la cuenta la resolvía.

Y la conclusión que se escribió en la decisión:

> «Funciones para lo irregular y lo dirigido por eventos. Contenedores para lo constante y de cara al cliente. La frontera la decide la cuenta de concurrencia por duración, y se revisa cuando el tráfico cambie de forma.»

A los seis meses.

```
cargas en funciones                              3             2
coste mensual total de las movidas             310 €          29 €
coste evitado al devolver la API a contenedores   —         3.790 €/mes
conexiones máximas desde funciones             1.410           24
caídas por agotar conexiones                     1             0
incidentes por vecino ruidoso                    1             0
eventos perdidos por trabajo de fondo          2 %             0
concurrencia reservada configurada           ninguna     en las 2 funciones
manejadores asíncronos idempotentes           0 de 2        2 de 2
```

La lección que esta clase traslada a la parte 09: las tres decisiones se resolvieron con la misma fórmula aplicada tres veces. Concurrencia igual a peticiones por duración explicó por qué el procesado de imágenes es barato, por qué la función de eventos abrió mil cuatrocientas conexiones y por qué la API constante costaba diez veces más. Y el problema más difícil de diagnosticar no fue ninguno de los tres: fue la instancia que se congela al responder, que no aparece en ninguna métrica y solo se ve como datos que faltan.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/117-serverless-limites-cold-starts-y-concurrencia/lab.py
```

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

Evidencia esperada

El artefacto principal es una función con límites, reintentos e idempotencia. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

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

■ Criterio de aceptación

- [ ] lab.py termina con código 0 y genera JSON válido.
- [ ] La entrega conecta al menos tres requisitos con mecanismos verificables.
- [ ] Existe una prueba positiva y una prueba negativa con evidencia.

- [ ] Seguridad, costo y operación aparecen como decisiones, no como anexos.
- [ ] Se declara una limitación y una condición que obligaría a revisar el diseño.
- [ ] Otra persona puede repetir el recorrido sin conocimiento tácito.

■■ Errores frecuentes

Síntoma	Causa probable	Corrección
La base de datos agota conexiones en cuanto llega un lote	Concurrencia igual a peticiones por duración: miles de instancias, una conexión cada una	Crea la conexión fuera del manejador, pon agrupador externo y fija concurrencia reservada calculada con el techo de la base.
Una API que no ha cambiado empieza a recibir limitaciones	Otra función consumió la concurrencia compartida de la cuenta	Reserva concurrencia garantizada a lo crítico y pon techo a los trabajos por lotes.
Se pierden datos enviados en segundo plano y los registros salen mezclados	La instancia se congela al responder y el trabajo pendiente se detiene	Termina todo antes de responder; lo que no quepa, encólalo.
El percentil 99 es diez veces la mediana	Arranques en frío en una ruta síncrona	Inicializa fuera del manejador, reduce el paquete, evita consultas remotas al arrancar y valora instancias mínimas o cambiar de modelo.
La factura es mucho mayor de lo esperado	Tráfico constante, que es donde pagar por ejecución pierde frente a capacidad encendida	Calcula el punto de cruce y prueba varios tamaños de memoria: el mínimo casi nunca es el más barato.
Efectos duplicados sin que nadie reintente a mano	La plataforma reintenta sola las invocaciones asíncronas	Haz idempotente todo manejador asíncrono, con la técnica de la clase 116.

■ Seguridad, ética y costo

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

■ Preguntas de comprobación

1. ¿Cómo se calcula la concurrencia y qué tres cosas decide?
2. ¿Por qué asignar la memoria mínima suele salir más caro?
3. ¿En qué casos importa el arranque en frío y en cuáles no?
4. ¿Qué le pasa al trabajo lanzado en segundo plano después de responder?
5. ¿Para qué dos cosas opuestas sirve la concurrencia reservada?

Referencias

- AWS (2025). Lambda: concurrency, reserved and provisioned — techo, suelo y comportamiento al escalar. <https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html>
- AWS (2025). Lambda execution environment lifecycle — inicialización, congelación tras responder y reciclado. <https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html>
- Google Cloud (2025). Cloud Run functions: cold starts and minimum instances — coste del arranque y mitigaciones. <https://cloud.google.com/run/docs/tips/general>
- Azure (2025). Azure Functions: performance and scale considerations — límites, planes y conexiones salientes. <https://learn.microsoft.com/azure/azure-functions/functions-best-practices>
- Jonas, E. y otros (2019). Cloud programming simplified: a Berkeley view on serverless computing — límites del modelo y cuándo no encaja. <https://arxiv.org/abs/1902.03383>

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

Descargar: Parte 09 en PDF · Manual integral

Anterior	Índice	Siguiente
← 116 · Sagas, outbox, idempotencia y deduplicación	Parte 09 · Programa	118 · API management, cuotas, versiones y monetización →

Evaluación — 117 Serverless: límites, cold starts y concurrencia

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 funcion-operable con diagrama o configuración, evidencia de ejecución, alternativa descartada, riesgo residual y criterio de rollback.

Criterio de aceptación

• [ ] Resultado reproducible por una segunda persona.

• [ ] Evidencia vinculada a cada afirmación técnica.

• [ ] Prueba negativa o de fallo incluida.

• [ ] Costos y permisos expresados con unidades y alcance.

• [ ] Limitaciones declaradas sin presentar la simulación como producción.

Escala

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

118 — API management, cuotas, versiones y monetización

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 4 Laboratorio: api · Estado: EXECUTABLECORE

Propósito

Publicar como producto lo que hasta ahora eran servicios internos: con clientes que no controlas, límites que hay que hacer cumplir, versiones que no puedes retirar cuando quieres y, si se cobra, un contador que se convierte en un sistema financiero. La clase separa lo que sí va en la puerta de entrada de lo que nunca debe acabar ahí, desarrolla los algoritmos de limitación con su modo de fallo concreto, y sostiene que el problema de versionar no es el mecanismo, sino la retirada.

Resultados de aprendizaje

Al finalizar podrás:

1. Delimitar qué corresponde a la puerta de entrada y qué no.
2. Elegir el algoritmo de limitación y la dimensión sobre la que se aplica.
3. Responder correctamente a un cliente limitado, para no provocar una tormenta.
4. Versionar con un procedimiento de retirada que funcione de verdad.
5. Medir el consumo con la fiabilidad que exige facturar por él.

Conceptos centrales

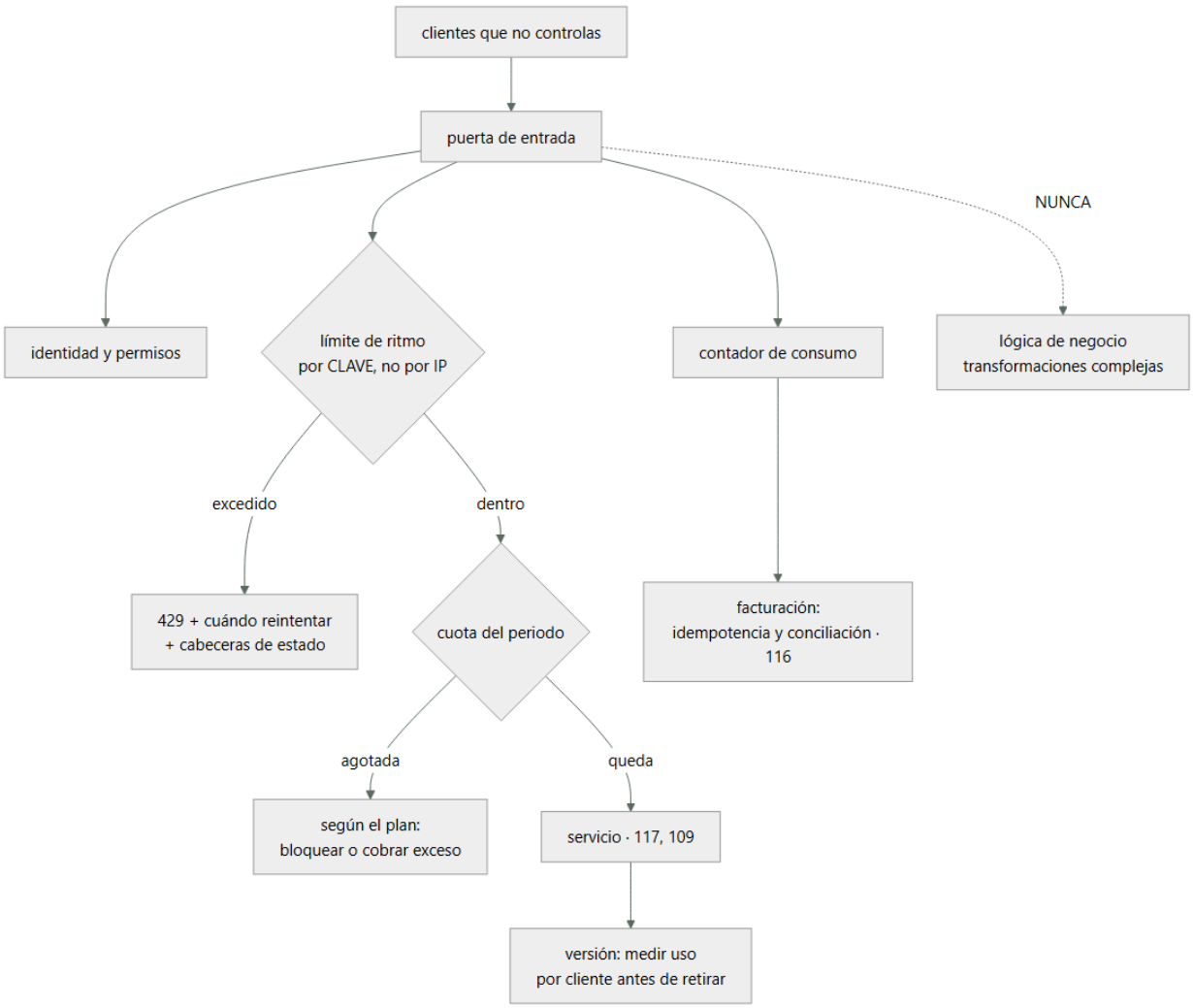
Concepto	Comprensión verificable
puerta de entrada	Punto único de acceso que aplica lo transversal: identidad, límites, encaminamiento y observabilidad. Nunca lógica de negocio.
límite de ritmo	Peticiones por unidad de tiempo. Protege la capacidad del sistema en el corto plazo.
cuota	Volumen total en un periodo largo. Es una condición comercial, no una protección técnica; son cosas distintas.
cubo de credenciales	Algoritmo que acumula permisos hasta un máximo y los gasta por petición. Permite ráfagas acotadas sin romper el ritmo medio.
retirada de versión	Proceso para dejar de servir una versión: anuncio, medición de uso por cliente, fecha y aplicación. Sin medición no se puede hacer.
contador de consumo	Registro de lo consumido por cliente. Si se factura con él, necesita idempotencia y conciliación como cualquier sistema de dinero.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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

■ Flujo de razonamiento



Desarrollo

1. Qué va en la puerta y qué no

La puerta de entrada resuelve lo que es igual para todas las API y no debe resolver nada específico de una.

```
SÍ
  autenticación y validación de credenciales
  autorización gruesa: ¿este cliente puede llamar a esta operación?
  límites de ritmo y cuotas
  encaminamiento por ruta, versión o cliente
  observabilidad: identificador de correlación, métricas por cliente
  respuestas de error uniformes
  terminación de cifrado y comprobaciones de tamaño

NO
  reglas de negocio
  composición de varias llamadas en una
  transformaciones que dependan del significado del dato
  autorización fina que dependa del estado del recurso
```

Y el motivo de la lista de la derecha es el modo de fallo conocido de este componente: la puerta se convierte en el sitio donde acaba todo lo que no tiene dueño claro, y entonces:

```
cambiar una regla de negocio exige tocar la infraestructura compartida
un error afecta a todas las API a la vez
nadie puede probar el sistema sin desplegar la puerta
y la lógica queda fuera del repositorio del servicio que la posee
```

Y una consecuencia que hay que asumir desde el principio: la puerta es un punto único de fallo con latencia añadida. Eso exige tratarla como lo que es:

```
latencia que añade                  medirla; 5-15 ms es razonable,
                                    80 ms no
qué pasa si se cae                  ¿hay camino alternativo para lo crítico?
su propia disponibilidad            mayor que la de cualquier servicio
                                    que protege
cambios de configuración            despliegue con las reglas de la parte 08,
                                    no ediciones en una consola
```

La última línea es donde más se peca: la puerta suele tener una interfaz cómoda que invita a cambiar cosas a mano, y eso es exactamente la deriva que la clase 103 eliminó del resto del sistema.

Y sobre los errores, una decisión pequeña con mucho efecto: un formato de error uniforme para todas las API, con código estable, mensaje legible e identificador de correlación. Los clientes programan contra los errores tanto como contra los éxitos.

2. Limitar bien

Primero, dos cosas que se confunden:

```
LÍMITE DE RITMO   100 peticiones/segundo
                  protege la capacidad; es técnico
CUOTA             1.000.000 de peticiones/mes
                  es una condición comercial
```

Un cliente puede estar dentro de su cuota mensual y aun así tener que ser limitado en un pico.

Los algoritmos, con su comportamiento real:

```
VENTANA FIJA
  contador que se reinicia cada minuto
  simple, y permite el DOBLE en la frontera:
    100 peticiones en el segundo 59 y 100 más en el 61
    → 200 en dos segundos con un límite de 100 por minuto

VENTANA DESLIZANTE
  cuenta los últimos 60 s reales
  correcto, y cuesta más de mantener

CUBO DE CREDENCIALES
  se rellena a ritmo constante hasta un máximo
  cada petición gasta una
  → permite ráfagas de hasta el tamaño del cubo
  → y limita el ritmo medio
  es el que mejor se ajusta a tráfico real

CUBO CON FUGA
  la salida es constante y lo que sobra se encola o se descarta
  → suaviza hacia el servicio de detrás
```

El del cubo es el que suele elegirse: el tráfico real llega a ráfagas, y un límite que no las tolera rechaza peticiones que el sistema podía atender.

Sobre qué dimensión se limita, que importa tanto como el algoritmo:

```
por clave de API o cliente     lo correcto
por usuario final              cuando un cliente sirve a muchos
por operación                  las caras merecen límite propio
por dirección IP               MAL como dimensión principal:
                               muchos clientes comparten salida,
                               y uno solo puede tener muchas
```

Y en sistemas distribuidos, el contador es compartido, y ahí hay un compromiso:

```
contador central exacto       correcto y añade latencia y dependencia
reparto local aproximado      cada nodo lleva su parte del límite
                              → si hay 10 nodos, cada uno permite 1/10
                              → y con tráfico desigual se limita de más
local con sincronización       lo habitual: local, ajustado cada pocos segundos
```

Qué responder al limitar, que es lo que evita el problema de la clase 113:

```
429
Retry-After: 30
X-RateLimit-Limit / Remaining / Reset
```

Sin la segunda cabecera, los clientes reintentan de inmediato y amplifican el problema que la limitación pretendía contener. Y conviene documentar que reintentar antes de tiempo puede contar contra el límite: es la única forma de que se respete.

Y dos figuras que conviene tener por encima del límite normal:

```
límite de emergencia    aplicable a un cliente concreto en un incidente
modo degradado          servir menos datos o desde caché en vez de rechazar
```

3. Versionar es fácil; retirar no

Los tres mecanismos y su realidad:

```
EN LA RUTA        /v1/pedidos
                  visible, trivial de encaminar y probar
                  «impuro» según algunos, y es el que usa casi todo el mundo

EN UNA CABECERA   Accept: application/vnd.tienda.v2+json
                  elegante, más difícil de probar y de depurar

SIN VERSIÓN       solo cambios compatibles, siempre
                  ideal, y no sobrevive a un cambio de modelo
```

Y la recomendación práctica: versión mayor en la ruta y todo lo demás compatible. Las versiones mayores deben ser raras; si hay una cada trimestre, el problema es el diseño, no el versionado.

Y las reglas de compatibilidad son las mismas que la clase 115 fijó para los eventos:

```
SE PUEDE   añadir campos opcionales en la respuesta
           añadir parámetros opcionales
           añadir operaciones
NO SE PUEDE  quitar o renombrar campos
           cambiar tipos
           hacer obligatorio lo que era opcional
           cambiar el significado sin cambiar el nombre
           cambiar códigos de error o de estado
```

Y una obligación que conviene documentar para los clientes: ignorar los campos que no conozcan. Un cliente que falla al recibir un campo nuevo convierte cualquier mejora en un cambio incompatible.

La retirada, que es el problema de verdad. No se puede retirar lo que alguien sigue usando, y para saberlo hace falta medirlo:

```
peticiones por versión y por cliente, con fecha del último uso
→ sin esa medición no hay retirada posible, solo esperanza
```

Y el procedimiento que funciona, que es el de la clase 106 con más margen porque los clientes son externos:

```
1. anunciar con fecha, en la documentación y por correo
2. cabeceras de aviso en cada respuesta de la versión vieja
   Deprecation: true / Sunset: <fecha>
3. contactar uno a uno con los que sigan usándola
4. cortes de prueba: apagar 1 h en una fecha anunciada
   → es lo que hace que los rezagados se enteren de verdad
5. apagar
```

El paso 4 es el que más funciona y el que menos se hace. Y el paso 3 exige lo que la clase 115 pedía para los eventos: saber quién consume.

Y una decisión que evita quedarse atrapado: cuántas versiones se soportan a la vez, escrito antes de publicar la primera. Dos suele ser suficiente; tres ya es un coste de mantenimiento notable en cada cambio.

4. Si se cobra, es un sistema financiero

En cuanto la factura de un cliente depende de un contador, ese contador deja de ser una métrica y pasa a ser un dato contable.

```
métrica          se puede perder un 0,1 % y no pasa nada
contador de
facturación      un 0,1 % es una factura mal emitida
```

Y eso trae exactamente los problemas de la clase 116:

```
un reintento no debe contar dos veces      → idempotencia por identificador
un evento perdido no debe dejar de contar  → tabla de salida
```

```
el contador y lo servido deben cuadrar     → conciliación periódica
```

Y una decisión que hay que tomar explícitamente y escribir en el contrato:

```
¿se cuenta la petición limitada?          normalmente no
¿se cuenta la que devuelve error 5xx?     no; es culpa tuya
¿se cuenta la que devuelve 404?           depende, y hay que decirlo
¿se cuenta por petición o por dato?       por dato es más justo y más difícil
```

Los planes y su comportamiento al agotarse, que es una decisión de producto con consecuencias técnicas:

```
bloquear al agotar       previsible para el cliente, y le rompe el servicio
cobrar el exceso         no rompe, y produce facturas sorpresa
avisar y degradar        avisar al 80 %, y al 100 % servir con menos ritmo
```

La tercera es la que menos conflictos genera, y exige avisar de verdad: al 50 %, al 80 % y al 100 %, por el canal que el cliente lea.

La experiencia de quien integra, que decide la adopción tanto como la funcionalidad. La medida útil es una:

```
tiempo desde que alguien llega a la documentación
hasta su primera llamada correcta
```

Es la versión externa de la fricción de la clase 107, y lo que la baja:

```
documentación generada de la especificación, siempre al día
ejemplos ejecutables, con credencial de prueba inmediata
entorno de pruebas con datos, no vacío
errores que explican qué hacer, no solo qué falló
registro de cambios y avisos de retirada en un sitio fijo
```

Y la especificación como fuente: si la documentación se escribe aparte, diverge; si se genera de lo que el servicio publica y se valida en la canalización, no puede divergir.

Y la lista de comprobación de la clase:

- ■ no hay lógica de negocio en la puerta de entrada
- ■ la configuración de la puerta se despliega, no se edita a mano
- ■ está medida la latencia que añade y qué pasa si se cae
- ■ el límite de ritmo usa cubo de credenciales, no ventana fija
- ■ la dimensión de limitación es la clave del cliente, no la IP
- ■ las respuestas limitadas llevan cuándo reintentar y estado del límite
- ■ límite de ritmo y cuota están separados y documentados
- ■ hay medición de uso por versión y por cliente
- ■ está escrito cuántas versiones se soportan a la vez
- ■ el procedimiento de retirada incluye cortes de prueba anunciados
- ■ el contador de facturación es idempotente y se concilia
- ■ está documentado qué peticiones cuentan y cuáles no
- ■ se mide el tiempo hasta la primera llamada correcta

Y el cierre que enlaza con la clase siguiente: la clase 116 dejó una conclusión pendiente —que los procesos con pasos, plazos y compensaciones se entienden mejor cuando la secuencia está escrita en un sitio—. Cómo se ejecuta esa secuencia de forma que sobreviva a reinicios, esperas de días y fallos parciales es la materia de la clase 119.

Ejemplo trabajado

CloudShop abre una API para sus socios logísticos y comercios afiliados. Empieza con cuatro clientes y llega a ciento noventa. Los cuatro problemas del camino son los cuatro apartados de esta clase.

Problema 1: la puerta acumuló lógica hasta ser intocable.

A los ocho meses:

```
reglas de transformación en la puerta                    41
de ellas, que dependían del significado del dato         29
equipos que podían desplegar la puerta                    1
tiempo medio para un cambio de un campo en una respuesta  6 días
incidentes causados por un cambio en la puerta        3 en 8 meses
  → los 3 afectaron a TODAS las API a la vez
```

Seis días para cambiar un campo, porque el cambio no estaba en el repositorio del servicio dueño.

```
reglas en la puerta                          41            12
las 12 restantes                              —      identidad, límites,
                                                     correlación, errores
lógica devuelta a los servicios               —            29
```

```
tiempo para cambiar un campo                6 días         < 1 día
equipos bloqueados por la puerta               4              0
```

Problema 2: la ventana fija, y el socio que tumbó el catálogo.

```
límite publicado                        600 peticiones/minuto
algoritmo                               ventana fija

11:59:58  un socio envía 600 peticiones
12:00:01  envía otras 600
→ 1.200 peticiones en 3 segundos, ambas «dentro del límite»
→ el servicio de catálogo satura; 4 minutos de degradación
```

Es el modo de fallo del apartado segundo, exactamente.

```
ráfaga máxima permitida                  2× el límite    tamaño del cubo (100)
ritmo medio garantizado                  irregular       10/s
peticiones rechazadas de tráfico legítimo   4,1 %          0,3 %
incidentes por ráfaga en frontera          2 / 8 meses      0
```

La penúltima fila es la que convenció: el cubo rechazó menos tráfico legítimo además de contener mejor las ráfagas, porque el tráfico real llega a golpes.

Y la respuesta a los limitados, que era un 429 pelado:

```
cabecera de cuándo reintentar                  no             sí
cabeceras de límite, restante y reinicio       no             sí
reintentos inmediatos tras un 429             89 %           7 %
carga de reintentos durante una limitación    ×4,2           ×1,1
```

Y la dimensión, que también estaba mal:

```
limitación por dirección IP
→ tres socios que salían por el mismo proveedor compartían límite
→ y uno de ellos, con veinte salidas, tenía veinte veces su límite
corregido a clave de cliente
reclamaciones por limitación injusta: de 11 a 0
```

Problema 3: la versión 1 que no se pudo retirar en dos años.

```
v2 publicada                                   mes 9
anuncio de retirada de v1                      mes 12, para el mes 18
uso de v1 en el mes 18                         31 % de las peticiones
medición por cliente                           no existía
→ no se sabía a quién avisar, así que no se apagó
```

Se montó la medición y apareció el reparto real:

```
clientes que aún usaban v1                          64 de 190
de ellos, con menos de 100 peticiones/mes           49
de ellos, con más de 100.000 peticiones/mes          3   ← el 27 % del tráfico
```

Tres clientes eran casi todo el problema. Con esa información el procedimiento fue viable:

```
mes 20  cabeceras de obsolescencia y fecha en todas las respuestas de v1
mes 21  contacto directo con los 3 grandes; migrados en 5 semanas
mes 22  corte de prueba anunciado de 1 h
        → 28 de los 49 pequeños reaccionaron esa semana
mes 23  segundo corte de prueba de 4 h
        → 17 más
mes 24  apagado; 4 clientes afectados, todos inactivos desde hacía meses
```

Los cortes de prueba movieron a 45 de 49 clientes pequeños que los correos no habían movido.

```
versiones mayores soportadas                sin política     2, escrito
medición de uso por cliente y versión          no             sí
tiempo de retirada de una versión          indefinido      6 meses
```

Problema 4: la facturación no cuadraba.

Al empezar a cobrar por consumo:

```
reclamaciones de facturación el primer mes            17 de 190
diferencia media reclamada                            +4,1 %
causa 1   los reintentos del cliente tras un 5xx contaban
causa 2   la puerta reintentaba internamente y contaba dos veces
causa 3   las peticiones limitadas contaban
```

El contador se trató como lo que era, con las técnicas de la clase 116:

registro del consumo	métrica agregada	tabla de salida
identificador de idempotencia por petición	no	sí
reintentos internos contados	sí	no
peticiones limitadas contadas	sí	no
errores 5xx contados	sí	no
conciliación diaria contador / registro	no	sí
reclamaciones de facturación	17 / mes	0-1 / mes
desviación en la conciliación	no se sabía	< 0,01 %

Y la política de agotamiento, que también generaba conflictos:

al agotar la cuota	bloqueo inmediato	avisos al 50/80/100 y ritmo reducido
clientes que se quedaron sin servicio sin saberlo	9 / trimestre	0

La adopción, medida.

documentación	escrita a mano	generada de la especificación
entorno de pruebas	vacío	con datos
credencial de prueba	por correo, 2 días	inmediata
tiempo hasta la primera llamada correcta	3,5 días	22 min
socios integrados por trimestre	6	31

Al cabo de dos años.

reglas de negocio en la puerta	29	0
incidentes por ráfaga en frontera de ventana	2	0
reintentos inmediatos tras limitación	89 %	7 %
reclamaciones por limitación injusta	11	0
versiones mayores vivas	2	2
retirada de una versión	imposible	6 meses
reclamaciones de facturación	17 / mes	0-1 / mes
tiempo hasta la primera llamada correcta	3,5 días	22 min
clientes	4	190

La lección que esta clase traslada a la parte 09: de los cuatro problemas, tres eran de información y no de tecnología. No se pudo retirar una versión durante dos años por no medir quién la usaba; no se pudo limitar con justicia por elegir la dimensión equivocada; y no se pudo facturar bien por tratar un dato contable como una métrica. El único puramente técnico —la ventana fija— se arregló cambiando un algoritmo en una tarde.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/118-api-management-cuotas-versiones-y-monetizacion/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 producto-api 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 producto-api 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
Cambiar un campo de una respuesta tarda días y bloquea a varios equipos	La puerta de entrada acumuló lógica de negocio	Devuelve a cada servicio lo que depende del significado del dato y deja en la puerta solo lo transversal.
Un cliente envía el doble del límite y el sistema satura	Ventana fija: dos ventanas consecutivas permiten el doble en la frontera	Usa cubo de credenciales, que acota la ráfaga y el ritmo medio a la vez.
Al limitar, la carga de reintentos empeora la situación	La respuesta no dice cuándo reintentar	Devuelve cuándo reintentar y el estado del límite, y documenta que reintentar antes cuenta contra el límite.
Clientes distintos comparten límite y uno solo consume mucho más	Se limita por dirección IP	Limita por clave de cliente, y añade dimensión por usuario final cuando un cliente sirva a muchos.
Una versión antigua no se puede retirar nunca	No se mide el uso por cliente, así que no se sabe a quién avisar	Mide peticiones por versión y cliente, anuncia con fecha y haz cortes de prueba anunciados.
Los clientes reclaman la factura	El contador de consumo se trató como una métrica y cuenta reintentos, errores y limitadas	Regístralo con tabla de salida e idempotencia, documenta qué cuenta y qué no, y concilia a diario.

■ 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 corresponde a la puerta de entrada y qué nunca debe acabar en ella?
2. ¿Qué diferencia hay entre límite de ritmo y cuota?
3. ¿Cuál es el modo de fallo concreto de la ventana fija y qué algoritmo lo evita?
4. ¿Por qué la dirección IP es una mala dimensión de limitación?
5. ¿Qué hace falta para poder retirar una versión, y qué paso del procedimiento es el más eficaz?

Referencias

- IETF (2024). RFC 9457: Problem Details for HTTP APIs — formato uniforme de errores. <https://www.rfc-editor.org/rfc/rfc9457.html>
- IETF (2023). RFC 8594: The Sunset HTTP header — anuncio de retirada de recursos y versiones. <https://www.rfc-editor.org/rfc/rfc8594.html>

- Cloudflare (2025). Rate limiting algorithms — ventana fija, deslizante y cubo de credenciales con sus efectos. <https://developers.cloudflare.com/waf/rate-limiting-rules/>
- Google (2025). API design guide: versioning and compatibility — cambios compatibles y versión mayor. <https://cloud.google.com/apis/design/versioning>
- OpenAPI Initiative (2025). Specification — documentación y validación generadas de la especificación. <https://spec.openapis.org/oas/latest.html>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Descargar: Parte 09 en PDF · Manual integral

Anterior	Índice	Siguiente
← 117 · Serverless: límites, cold starts y concurrencia	Parte 09 · Programa	119 · Workflows y orquestación durable →

Evaluación — 118 API management, cuotas, versiones y monetización

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 producto-api 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.

119 — Workflows y orquestación durable

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 4 Laboratorio: orchestration · Estado: EXECUTABLECORE

Propósito

Ejecutar la secuencia con compensaciones de la clase 116 en un sitio donde el estado sobrevive a la muerte del proceso, donde esperar treinta días es una instrucción normal y donde se puede preguntar en qué paso va cada instancia. La clase explica el mecanismo que lo hace posible —volver a ejecutar el código contra un historial grabado—, la restricción severa que ese mecanismo impone —el código tiene que ser determinista— y el problema operativo que nadie anuncia: cómo se despliega una versión nueva cuando hay diez mil instancias a medias.

Resultados de aprendizaje

Al finalizar podrás:

1. Reconocer los procesos que no caben en una petición ni en una función.
2. Explicar la reejecución contra historial y por qué obliga al determinismo.
3. Separar lo que va en la orquestación de lo que va en las actividades.
4. Desplegar cambios sin romper las instancias en curso.
5. Elegir entre motor durable, máquina de estados declarativa y cola con estado propio.

Conceptos centrales

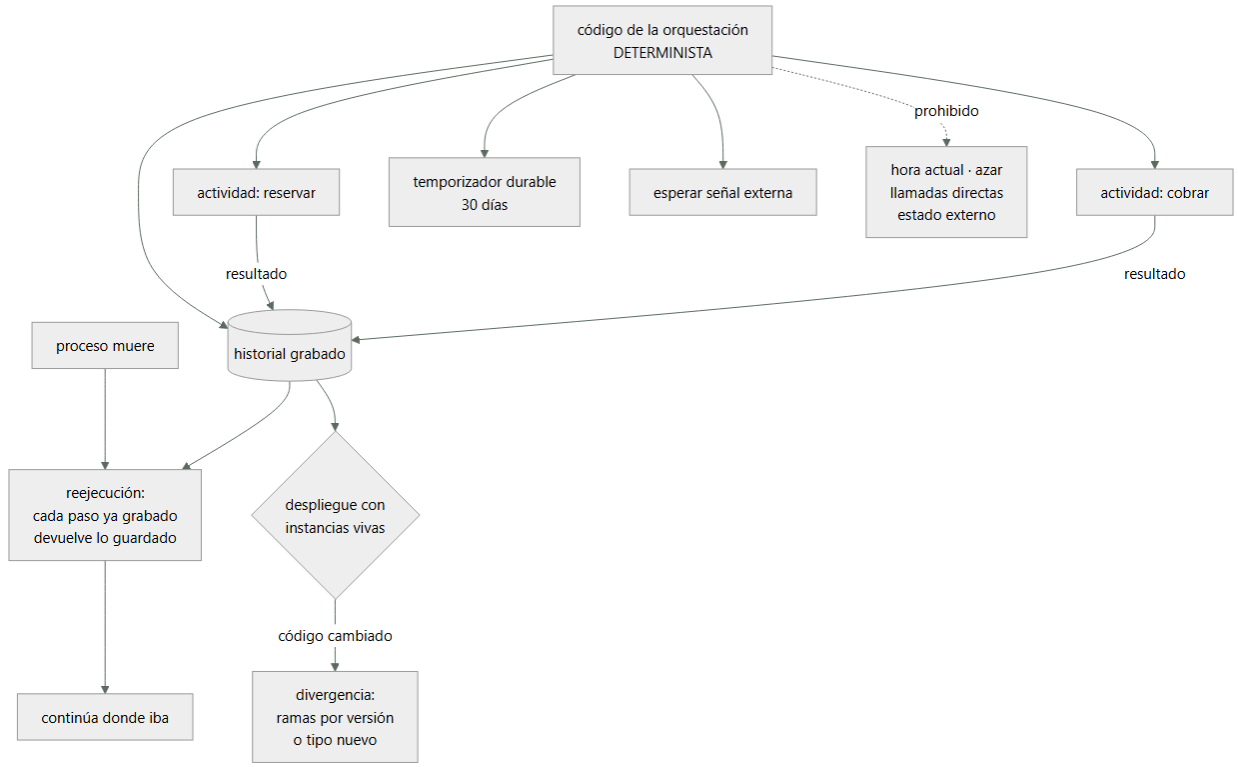
Concepto	Comprensión verificable
orquestación durable	Motor que graba el resultado de cada paso y reconstruye la ejecución tras un fallo, continuando donde iba.
reejecución contra historial	Al reanudar, el código se ejecuta de nuevo desde el principio, pero cada paso ya grabado devuelve su resultado guardado en vez de ejecutarse.
determinismo	Que el mismo código con el mismo historial tome siempre las mismas decisiones. Sin él, la reejecución diverge y la instancia se rompe.
actividad	Unidad que sí produce efectos: llama, escribe, cobra. Se reintenta, así que debe ser idempotente.
temporizador durable	Espera que sobrevive a reinicios y despliegues. Permite «espera 30 días» sin ningún proceso vivo mientras tanto.
reinicio como instancia nueva	Cerrar la instancia actual y abrir otra con el estado resumido, para que el historial no crezca sin fin.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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

■ Flujo de razonamiento



Desarrollo

1. Qué no cabe en ningún sitio de los anteriores

La clase 116 dejó una secuencia con compensaciones y plazos. Y hay tres razones por las que no cabe donde hemos ejecutado cosas hasta ahora:

```
DURA MÁS QUE UN PROCESO
  «esperar 30 días a que expire la devolución»
  → no cabe en una petición ni en una función (clase 117)

TIENE QUE SOBREVIVIR A CUALQUIER COSA
  despliegues, reinicios, caídas de la máquina
  → el estado no puede estar en memoria

HAY QUE PODER PREGUNTARLE
  «¿en qué paso va el pedido 1421 y por qué lleva ahí 3 horas?»
  → es la pregunta que la clase 115 dejó sin responder
```

Lo que se hace normalmente sin un motor —y funciona hasta cierto tamaño— es una máquina de estados propia:

```
tabla con el estado de cada instancia
un proceso que la lee y avanza los que toca
colas para cada paso
temporizadores con filas de «hacer a partir de las …»
```

Y conviene decir con claridad que eso está bien para procesos de dos o tres pasos. Lo que pasa cuando crece:

```
cada paso nuevo añade estados, transiciones y casos de fallo
las compensaciones multiplican las transiciones
nadie puede leer el proceso completo: está repartido en seis consumidores
y los plazos hay que implementarlos a mano, uno a uno
```

El motor durable resuelve eso permitiendo escribir el proceso como código secuencial:

```
def pedido(id):
    reserva = actividad(reservar, id)
    try:
        pago = actividad(cobrar, id)
    except Exception:
        actividad(liberar_reserva, reserva)
        return "cancelado_por_pago"
    actividad(crear_envio, id)
```

```
    actividad(notificar, id)
    esperar(dias=30)                    # temporizador durable
    if not senal_recibida("devolucion"):
        actividad(cerrar_pedido, id)
```

Y lo notable de ese código es lo que no tiene: no hay tabla de estado, ni reintentos, ni recuperación tras reinicio, ni gestión de temporizadores. Todo eso lo aporta el motor. La línea de los treinta días no mantiene nada vivo: el proceso se despierta cuando toca.

2. Cómo funciona, y qué prohíbe

El mecanismo es sencillo de enunciar y tiene una consecuencia grande.

```
cada vez que la orquestación llama a una actividad:
  el motor comprueba si ya está en el historial
    sí  → devuelve el resultado grabado, sin ejecutar nada
    no  → la ejecuta y graba el resultado

al reanudar tras un fallo:
  el código se ejecuta DESDE EL PRINCIPIO
  y todos los pasos ya grabados se resuelven al instante
  hasta llegar al punto donde se quedó
```

De ahí sale la restricción central:

```
el código de la orquestación tiene que ser DETERMINISTA

porque se ejecuta muchas veces y debe tomar SIEMPRE
las mismas decisiones en el mismo orden
```

Y la lista de lo prohibido dentro de la orquestación:

```
la hora actual                → usar el reloj del motor
números aleatorios            → pedirlos como actividad, o al motor
identificadores generados     → igual
leer un fichero, una base o
  llamar a una API            → eso es una actividad
variables globales o estado
  del proceso                 → no sobreviven y no se reproducen
recorrer un diccionario sin
  orden garantizado           → el orden puede cambiar entre ejecuciones
hilos y concurrencia propia   → usar los mecanismos del motor
```

El caso del recorrido desordenado es el que más sorprende y el que más cuesta diagnosticar, porque falla de forma intermitente y solo tras un reinicio.

Y la regla mental que resume todo:

```
la orquestación DECIDE; las actividades HACEN
si algo puede dar un resultado distinto dos veces, es una actividad
```

Las actividades, que es donde vive todo lo de las clases anteriores:

```
se reintentan según la política que se les fije
  → espera creciente, variación y límite: clase 113
se entregan al menos una vez
  → tienen que ser idempotentes: clase 116
tienen su propio plazo
  → y hay dos plazos distintos: el de una ejecución y el del conjunto
     de intentos
```

Y el latido para las largas: una actividad que tarda veinte minutos debe informar de que sigue viva, o el motor la dará por muerta y la reintentará —el mismo mecanismo, y el mismo error, que el tiempo de invisibilidad de la clase 113—.

Y dos límites que hay que respetar por diseño:

```
el historial crece con cada paso
  → procesos con millones de iteraciones: reiniciar como instancia nueva
los argumentos y resultados viajan y se graban
  → no pasar objetos grandes: pasar referencias al almacenamiento
```

3. Desplegar con instancias a medias

Este es el problema operativo específico de este modelo, y el que más disgustos da.

```
hay 40.000 instancias en curso, algunas empezadas hace 3 semanas
```

```
se despliega código nuevo
una instancia vieja se reanuda y se reejecuta con el CÓDIGO NUEVO
contra un historial creado por el CÓDIGO VIEJO
→ si las decisiones no coinciden, la instancia se rompe
```

Qué cambios son seguros y cuáles no:

```
SEGURO
  cambiar el interior de una actividad
  cambiar la política de reintentos
  añadir pasos DESPUÉS del punto donde están todas las instancias vivas

PELIGROSO
  añadir, quitar o reordenar llamadas a actividades
  cambiar una condición que decide qué actividad se llama
  cambiar la duración de un temporizador ya programado
  cambiar el nombre o la firma de una actividad
```

Las dos técnicas para hacerlo bien:

```
1. RAMAS POR VERSIÓN EN EL CÓDIGO
   version = obtener_version("anadir-verificacion", por_defecto=1, maxima=2)
   if version >= 2:
       actividad(verificar_fraude, id)
   → las instancias viejas siguen por la rama 1; las nuevas, por la 2
   → y las ramas viejas se limpian cuando ya no queda ninguna instancia

2. TIPO NUEVO Y VACIADO
   se publica «pedido-v2» y las instancias nuevas van ahí
   «pedido-v1» deja de recibir instancias y se apaga cuando se vacía
   → más limpio y obliga a mantener dos durante el vaciado
```

La primera es cómoda y deja restos: sin limpieza, el código acumula ramas de versiones que ya no existen. Conviene medirlo:

```
instancias vivas por versión de cada rama
→ si una rama lleva meses sin instancias, se borra
```

Y la segunda es la que conviene cuando el cambio es grande, y encaja con lo que la clase 106 llamó contrato de versiones: dos vivas, y vaciado con plazo.

Y una precaución que evita el peor caso: desplegar primero a un entorno con instancias reales de larga duración y comprobar que se reanudan. Un entorno vacío no detecta ninguno de estos errores, porque no hay historial contra el que divergir.

Y qué hacer cuando una instancia ya se rompió:

```
reiniciarla desde un punto anterior del historial
terminarla y crear una nueva con el estado que se pueda recuperar
y en ambos casos: la lista de las rotas tiene que ser visible
```

4. Operar, elegir y no pasarse

Lo que se gana en visibilidad, que es la respuesta a la pregunta que la clase 115 dejó abierta:

```
¿en qué paso está el pedido 1421?          consulta directa
¿cuánto lleva ahí?                          en el historial
¿qué actividad falló y cuántas veces?       en el historial
¿qué instancias llevan más de N horas?      consulta por estado y tiempo
¿cuántas esperan una señal que no llega?    consulta
```

Y las alertas que hacen falta, que son las de siempre con otro nombre:

```
instancias en curso por encima de su duración esperada
instancias con actividades que agotaron reintentos
instancias terminadas por error, por tipo
antigüedad de la instancia más vieja de cada tipo
cola de tareas del motor sin trabajadores que la atiendan   ← ley 13
```

La última es el modo de fallo silencioso característico: si no hay trabajadores atendiendo una cola de tareas, las instancias no fallan: se quedan quietas.

Los tres modelos, para elegir con criterio:

```
MOTOR DURABLE CON CÓDIGO
  + el proceso se lee como código; pruebas normales; lógica compleja
```

```
  – hay que operar el motor y respetar el determinismo
  encaja  procesos largos, con compensaciones y ramas

MÁQUINA DE ESTADOS DECLARATIVA (del proveedor)
  + gestionada, integrada con el resto de servicios, visual
  – la lógica se escribe en un lenguaje de expresiones limitado
  – probar en local es incómodo y las condiciones complejas se vuelven ilegibles
  encaja  secuencias de pasos con poca lógica y mucha integración

COLA CON ESTADO PROPIO
  + nada que operar de más; todo el mundo lo entiende
  – cada paso y cada compensación se implementan a mano
  encaja  procesos de 2 o 3 pasos sin espera larga
```

Y el error de diseño más común una vez se tiene el motor: meterlo todo dentro.

```
no es para          una petición síncrona de 80 ms
                    un cálculo puro
                    reemplazar toda cola de la clase 113
es para             procesos de negocio con pasos, esperas y vuelta atrás
```

Y conviene recordar que el motor no elimina ninguna de las obligaciones anteriores: las actividades siguen necesitando idempotencia, los efectos externos siguen necesitando clave, y las compensaciones siguen sin ser vueltas atrás.

Y la lista de comprobación de la clase:

- el proceso justifica un motor: dura, espera o compensa
- la orquestación no usa reloj, azar, entrada/salida ni estado del proceso
- ningún recorrido depende de un orden no garantizado
- todas las actividades son idempotentes
- las actividades largas envían latido
- no viajan objetos grandes: van referencias
- los procesos con muchas iteraciones reinician como instancia nueva
- los cambios de código usan ramas por versión o tipo nuevo con vaciado
- las ramas de versión sin instancias vivas se limpian
- el despliegue se prueba contra instancias reales de larga duración
- hay alerta por cola de tareas sin trabajadores
- las instancias rotas o atascadas son visibles y tienen procedimiento

Y el cierre que enlaza con la clase siguiente: con esto está completo el material de la parte 09. La clase 120 monta el sistema de pedidos entero sobre lo construido y, sobre todo, califica la hipótesis escrita en la clase 108: cuántos de los ocho mecanismos de la parte 08 sobrevivieron al contacto con el estado, y si la ley dominante fue la que se predijo.

Ejemplo trabajado

CloudShop lleva la secuencia de pedido de la clase 116 a un motor durable. Lo que se gana está claro desde la primera semana; lo interesante son los tres problemas propios del modelo, que ninguna clase anterior podía anticipar.

Punto de partida: la máquina de estados propia.

```
estados en la tabla                                14
transiciones implementadas                         41
consumidores que participaban                       6
líneas de código de coordinación                2.900
lugares donde estaba escrito el proceso completo    0
tiempo para añadir un paso nuevo                3 semanas
instancias atascadas detectadas por clientes    31 / mes
```

La quinta línea es la que dolía: el proceso no estaba escrito en ningún sitio; había que reconstruirlo leyendo seis consumidores.

Con el motor:

```
líneas de coordinación                    2.900          310
sitios donde se lee el proceso completo       0            1
tiempo para añadir un paso                3 semanas      2 días
temporizadores implementados a mano           4            0
reintentos implementados a mano              11            0
instancias atascadas detectadas por clientes 31 / mes       0
                                                   (las detecta la alerta)
```

Problema 1: el error de determinismo que solo aparecía tras un reinicio.

```
síntoma    tras cada despliegue, entre 20 y 60 instancias se rompían
```

```
           con «divergencia de historial»
frecuencia intermitente; no se reproducía en pruebas
semanas hasta encontrarlo                                  3
```

La causa era una línea inocente en la orquestación:

```
for sku, cantidad in lineas.items():        # orden no garantizado
    actividad(reservar, sku, cantidad)
```

Al reejecutar, el diccionario se recorría en otro orden y las actividades no coincidían con el historial.

```
recorrido                              diccionario     lista ordenada por sku
instancias rotas por despliegue             20-60            0
```

Y se añadió lo que lo habría detectado antes: una comprobación automática que reejecuta historiales reales contra el código nuevo en la canalización.

```
historiales guardados para la prueba                        500
divergencias detectadas antes de desplegar, en 6 meses        7
de ellas, que habrían roto instancias en producción           7
```

Problema 2: el despliegue que rompió 8.400 instancias.

```
cambio     añadir un paso de verificación de fraude entre reservar y cobrar
desplegado sin ramas por versión
instancias vivas en ese momento                          11.200
instancias que ya habían pasado por «reservar»            8.400
→ al reejecutarse, esperaban «cobrar» y encontraron «verificar»
instancias rotas                                          8.400
tiempo de recuperación                                    6 h 20
cómo se recuperaron       reinicio desde un punto anterior del historial
pedidos afectados que hubo que revisar a mano               212
```

La corrección y su efecto en los cambios siguientes:

```
técnica de cambio                          desplegar       ramas por versión
prueba con instancias de larga duración        no             sí
instancias rotas por despliegue en 6 meses    8.400            0
ramas de versión acumuladas                     —              9
ramas limpiadas al vaciarse                     —              6
```

La penúltima fila es el resto que deja la técnica: nueve ramas de compatibilidad en el código, de las que seis ya se pudieron borrar por no quedar instancias.

Problema 3: el proceso de treinta días y el historial que crecía.

El seguimiento de la ventana de devolución se implementó como un bucle que comprobaba cada hora:

```
iteraciones en 30 días                                     720
eventos de historial por instancia                       ~2.900
tamaño del historial                                      1,4 MB
instancias vivas simultáneas                             94.000
almacenamiento del motor                                   131 GB
tiempo de reejecución de una instancia vieja                11 s
```

Once segundos para reanudar una instancia. La corrección fue doble:

```
un temporizador durable de 30 días en vez de 720 comprobaciones
y reinicio como instancia nueva para los procesos de suscripción,
  que sí son indefinidos
```

```
eventos de historial por instancia          ~2.900             9
tamaño del historial                        1,4 MB           4 KB
almacenamiento del motor                    131 GB          0,4 GB
tiempo de reanudación                        11 s           40 ms
```

El fallo silencioso que costó cuatro horas.

```
02:10  se despliega un cambio en la configuración de los trabajadores
02:10  los trabajadores de la cola «pedidos-actividades» no arrancan
02:10  las instancias NO fallan: se quedan esperando
06:20  un cliente pregunta por un pedido de las 02:15
instancias detenidas                                      4.180
alertas que se dispararon                                     0
```

Es la ley 13 otra vez, ahora en el motor: una cola de tareas sin trabajadores no produce ningún error.

```
alerta por cola sin trabajadores                no             sí
alerta por antigüedad de la instancia más vieja no             sí
```

```
tiempo de detección de un caso igual         4 h 10          2 min
```

La visibilidad, medida contra la clase 115.

```
responder «¿dónde está el pedido 1421?»      40 s            2 s
equipos a consultar                            0              0
saber por qué lleva 3 h parado          reconstruir traza   en el historial
saber cuántos hay parados ahora          no era posible      consulta
reanudar uno concreto                    no era posible      un comando
```

A los ocho meses.

```
líneas de coordinación                        2.900           310
tiempo para añadir un paso                  3 semanas       2 días
instancias atascadas detectadas por clientes  31 / mes          0
instancias rotas por despliegue            8.400 (una vez)      0
divergencias detectadas antes de desplegar       —          7 / 6 meses
almacenamiento del motor                     131 GB         0,4 GB
tiempo de reanudación de una instancia         11 s          40 ms
detección de trabajadores caídos             4 h 10          2 min
compensaciones automáticas correctas           —          1.397 de 1.412
```

La lección que esta clase traslada a la parte 09: el motor eliminó dos mil seiscientas líneas de coordinación y respondió por construcción la pregunta que la coreografía no podía responder. Y a cambio impuso una restricción que no existe en ninguna otra parte de este programa: el código de la orquestación no puede consultar la hora, ni sortear, ni recorrer un diccionario. Los dos incidentes caros —tres semanas de diagnóstico y ocho mil cuatrocientas instancias rotas— fueron las dos formas de saltarse esa restricción sin darse cuenta.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/119-workflows-y-orquestacion-durable/lab.
py
```

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

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

Evidencia esperada

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

Reto verificable

Construye workflow-durable 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
Algunas instancias se rompen tras cada despliegue y no se reproduce en pruebas	La orquestación no es determinista: recorridos sin orden, reloj o azar dentro del código	Saca todo lo no determinista a actividades, ordena los recorridos y reejecuta historiales reales contra el código nuevo en la canalización.
Un cambio de código rompe miles de instancias en curso	Se añadió o reordenó una llamada a actividad sin proteger a las instancias vivas	Usa ramas por versión o publica un tipo nuevo y vacía el antiguo; prueba contra instancias reales de larga duración.
El almacenamiento del motor crece y reanudar tarda segundos	El historial acumula miles de eventos por instancia, normalmente por bucles de comprobación	Sustituye los bucles por temporizadores durables y reinicia como instancia nueva los procesos indefinidos.
Las instancias dejan de avanzar sin que nada falle	Ley 13: una cola de tareas sin trabajadores no produce error	Alerta por cola sin trabajadores y por antigüedad de la instancia más vieja de cada tipo.
Una actividad larga se ejecuta dos veces	El motor la dio por muerta al no recibir señal de vida, igual que el tiempo de invisibilidad de la clase 113	Envía latido desde las actividades largas y hazlas idempotentes.
El motor acaba conteniendo procesos que no lo necesitan	Se usa para todo una vez está disponible	Resérvalo para procesos con esperas, compensaciones o duración larga; lo de dos pasos sigue estando bien en una cola.

■ 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 mecanismo permite que una instancia sobreviva a la muerte del proceso?
2. ¿Por qué la orquestación debe ser determinista y qué está prohibido dentro de ella?
3. ¿Qué cambios de código son seguros con instancias en curso y cuáles no?
4. ¿Qué dos técnicas permiten desplegar sin romper instancias vivas y qué resto deja cada una?
5. ¿Cuándo basta una cola con estado propio en vez de un motor durable?

Referencias

- Temporal (2025). Workflow determinism and versioning — reejecución, restricciones y ramas por versión. <https://docs.temporal.io/workflows>
- Azure (2025). Durable Functions: orchestrator code constraints — qué no se puede hacer en el código de orquestación. <https://learn.microsoft.com/azure/azure-functions/durable/durable-functions-code-constraints>
- AWS (2025). Step Functions: standard and express workflows — máquina de estados declarativa y sus límites. <https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html>
- Google Cloud (2025). Workflows: syntax and error handling — pasos, reintentos y compensación declarativa. <https://cloud.google.com/workflows/docs/reference/syntax>
- Richardson, C. (2025). Orchestration-based saga — cuándo centralizar la secuencia y sus compensaciones. <https://microservices.io/patterns/data/saga.html>
- Documentación oficial vigente del servicio implementado; registra URL y fecha de consulta.

Evaluación — 119 Workflows y orquestación durable

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 workflow-durable 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.

120 — Proyecto: pipeline de pedidos orientado a eventos

Parte: 09 — Datos, mensajería, serverless e integración Nivel: avanzado · Horas estimadas: 8 Laboratorio: capstone · Estado: EXECUTABLECORE

Propósito

Montar el recorrido completo de un pedido con todo lo de las clases 109 a 119 funcionando a la vez, y cerrar la parte con las tres piezas de siempre: calificar las cuatro predicciones de la clase 108 con los datos de la parte, incluidas las dos que salieron mal; actualizar el recuento de leyes con la que ha aparecido cuatro veces en estas doce clases; y escribir la predicción que la parte 10 tendrá que corregir. Y con un dato incómodo que se cuenta al final: cómo se detectó cada uno de los problemas de esta parte.

Resultados de aprendizaje

Al finalizar podrás:

1. Montar el recorrido de un pedido con sus garantías explícitas en cada tramo.
2. Situar cada almacén y cada transporte por la pregunta que responde.
3. Calificar las cuatro predicciones de la clase 108 con evidencia.
4. Incorporar la ley 18 al cuestionario, con sus cuatro apariciones.
5. Escribir la predicción de la parte 10 en términos que se puedan desmentir.

Conceptos centrales

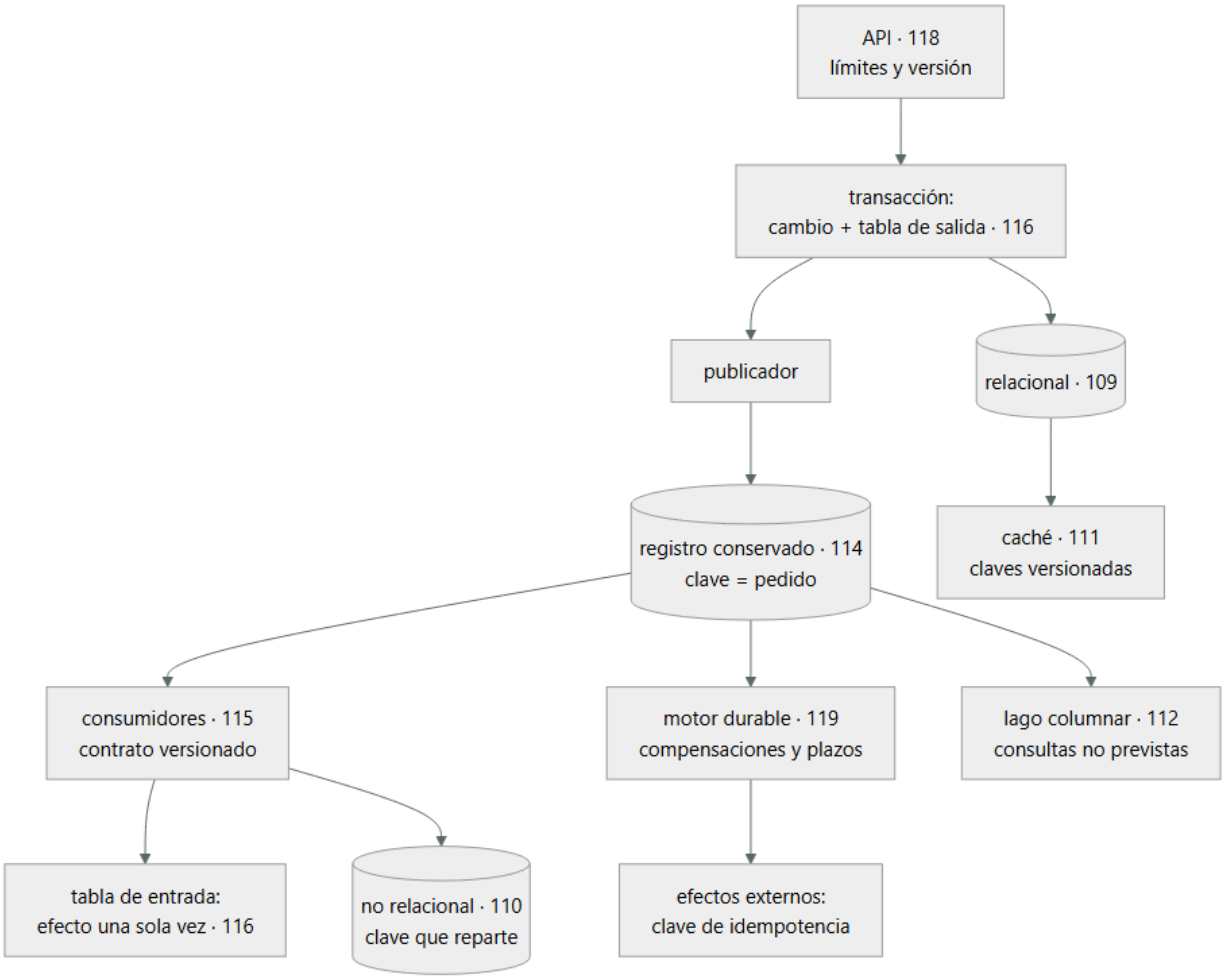
Concepto	Comprensión verificable
recorrido con garantías	Descripción de un proceso tramo a tramo indicando qué se garantiza en cada uno y qué hay que reconstruir.
punto de atomicidad	Lugar donde dos efectos ocurren o no ocurren juntos. Todo el diseño de la parte 09 consiste en colocarlo bien.
calificación de hipótesis	Comparar lo predicho con lo ocurrido publicando también lo que se predijo mal.
ley 18	Hacer algo asíncrono no elimina ninguna garantía: la traslada a la aplicación, que tiene que reconstruirla.
medio de detección	Cómo se supo de un problema: alerta, caída, auditoría, factura o cliente. Es la medida honesta del estado de la observabilidad.
hipótesis de la parte 10	Predicción escrita ahora sobre lo que ocurrirá cuando el sujeto sea saber si el sistema funciona.

Modelo mental

La base de datos correcta depende del patrón de acceso y de las garantías necesarias; distribuir datos convierte latencia, consistencia y fallos parciales en decisiones de diseño.

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

■ Flujo de razonamiento



Desarrollo

1. El recorrido, tramo a tramo

Las once clases anteriores han dado piezas. Juntas forman un recorrido en el que cada tramo tiene una garantía distinta, y lo útil es escribirlas todas:

cliente → API	síncrono; límite de ritmo y versión	118
API → base	transacción local; atómica	109
base → tabla de salida	MISMA transacción: el hecho existe si y solo si el cambio se guardó	116
tabla de salida → registro	al menos una vez; puede repetir	116
registro → consumidores	al menos una vez; orden por clave	114
consumidor → efecto	efecto una sola vez, con tabla de entrada en la misma transacción	116
consumidor → externo	idempotente solo si el proveedor acepta clave; si no, conciliar	116
motor durable	supervivencia a reinicios; plazos y compensaciones	119
registro → lago	eventualmente; consultas no previstas	112
base → caché	desfase acotado; claves versionadas	111

Y la lectura que ordena todo lo anterior: el diseño de la parte 09 consiste en colocar bien los puntos de atomicidad. Solo hay dos sitios donde dos cosas ocurren juntas —la transacción del productor y la del consumidor— y todo lo demás se construye alrededor.

Y el mapa de qué almacén responde a qué pregunta, que es la otra decisión estructural:

«dame este pedido y modifícalo»	relacional	109
«dame el valor de esta clave, millones de veces»	clave-valor	110
«dame esto rapidísimo y acepto desfase»	caché	111
«dame algo que nadie previó, sobre		

```
 millones de filas»                        lago columnar       112
«dame el histórico de hechos y déjame
 releerlo»                                 registro            114
```

Y el coste que este programa ya nombró en la clase 110 y que aquí se hace visible del todo: cinco sistemas de datos donde había uno. Cada uno con su copia, su vigilancia, su recuperación y su forma de fallar. Es la factura real de esta parte, y hay que decirla junto a los beneficios.

2. El proyecto

Montar el recorrido para el pedido, con estas entregas:

```
1. ENTRADA
   API versionada, con límite por cubo de credenciales y por clave
   respuesta que indica cuándo reintentar

2. ESCRITURA
   transacción única: estado del pedido + fila en la tabla de salida
   publicador con vigilancia de la fila más vieja sin publicar
   limpieza programada de la tabla

3. TRANSPORTE
   registro con particiones dimensionadas para el paralelismo máximo
   clave = identificador de pedido
   esquema en registro, validado al producir
   catálogo de productores y consumidores

4. CONSUMO
   tabla de entrada en la misma transacción que el efecto
   errores permanentes a fallidos sin reintentar
   alerta desde el primer mensaje fallido
   escalado por antigüedad, con techo según la dependencia

5. PROCESO LARGO
   motor durable con la secuencia, plazos por paso y compensaciones
   pasos irreversibles al final
   ramas por versión para desplegar con instancias vivas

6. LECTURA
   caché con claves versionadas y política ante origen caído
   réplicas solo para lo que no decide escrituras
   lago columnar particionado y ordenado por la columna del filtro

7. EFECTOS EXTERNOS
   clave de idempotencia a quien la acepte
   conciliación diaria con quien no
```

Y las preguntas cuya respuesta hay que escribir, porque son las que distinguen un montaje de un sistema:

```
¿qué pasa si el registro no está disponible 20 minutos?
¿qué pasa si un consumidor lleva 6 horas parado?
¿qué pasa si se procesa el mismo mensaje dos veces, en cada consumidor?
¿cuánto tarda en verse un cambio de precio, sumando todas las capas?
¿cuántas conexiones abre el sistema en su máximo, y cuál es el techo?
¿qué se pierde si hay que restaurar la base a hace una hora?
```

La última es la que casi nadie responde y la que decide el diseño de la copia: el registro y la base se restauran por separado y no quedan en el mismo instante.

Las pruebas negativas, que son la entrega más valiosa:

- matar un consumidor a mitad de proceso
- entregar el mismo mensaje dos veces a propósito
- provocar una conmutación de la base
- vaciar el caché en hora punta
- parar el publicador de la tabla de salida 30 minutos
- desplegar código nuevo con instancias del motor a medias
- enviar un mensaje con un campo renombrado
- agotar la cuota de un cliente de la API

Y la parte específica de los tres proveedores, que es más corta de lo que parece:

```
idéntico     el diseño entero: puntos de atomicidad, claves, contratos,
             idempotencia, orden, particionado
```

```
distinto     nombres de servicio, límites concretos, formato de las
             políticas y detalles de la federación de identidad
```

Y una advertencia sobre los límites, que sí varían: los máximos de tamaño de mensaje, retención, particiones y concurrencia son distintos en cada proveedor, y un diseño que funciona en uno puede chocar con un límite en otro. Conviene comprobarlos antes, no después.

3. Calificación de la hipótesis de la parte 08

La clase 108 escribió cuatro predicciones. Dos salieron bien, una salió mal y otra no se pudo probar por un motivo que también dice algo.

Predicción 1: «de los ocho mecanismos de las partes 07 y 08, tres o menos se aplicarán sin cambios a un componente con estado».

```
mecanismo                    ¿sin cambios?   evidencia
artefacto inmutable               sí         la imagen del servicio sigue igual
interruptor de funcionalidad      sí         funcionó igual; se usó en 113 y 116
puerta de canalización            sí         y MEJORÓ: la puerta de
                                             compatibilidad de esquema paró
                                             11 cambios (102) y 6 más (115)
camino asfaltado                  sí         se aplicó tal cual

reversión                         no         punto de no retorno (102)
entorno efímero                   no         una base por cambio: 27 vivas,
                                             2.400 €/mes (108)
reconciliación con poda           no         ver predicción 4
canario                           no         no detecta lo que depende del
                                             volumen de datos (109)

veredicto: EQUIVOCADA. Sobrevivieron cuatro, no tres o menos.
```

Y el error fue de pesimismo. Lo que la predicción no vio: la puerta de canalización no solo sobrevivió, sino que se volvió más valiosa, porque en un sistema con estado hay más contratos que romper —esquemas de base, de evento y de API— y una puerta los comprueba igual de bien.

Predicción 2: «la ley 14 será la ley dominante de la parte 09, con más apariciones que la ley 13».

```
LEY 14 en la parte 09        3 apariciones
  109  versión mayor, ordenación, cifrado y tipo de clave primaria
  110  clave de partición: migración de 31 h
  114  número de particiones y clave: 5 semanas de migración

LEY 13 en la parte 09        5 apariciones
  112  caducidad de instantáneas que no se ejecutaba
  113  41.216 mensajes en la cola de fallidos, 67 días
  115  un consumidor parado 19 h sin que nadie lo supiera
  116  la tabla de salida creciendo sin límite
  119  4.180 instancias detenidas por una cola sin trabajadores

veredicto: EQUIVOCADA. La ley 14 apareció y no dominó.
```

Y el dato que hace interesante el error: la ley 13 ha aparecido en todas las partes de este programa desde la 06. Es la única con ese historial, y su forma no cambia nunca: algo deja de ejecutarse y el sistema no lo distingue de que no haya nada que hacer.

Predicción 3: «el problema más difícil será la garantía en la frontera, y la respuesta recurrente será hacer la operación repetible sin efecto adicional».

```
veredicto: ACERTADA, y se quedó corta
```

Fue el contenido de la clase 116 entera y apareció en cuatro sitios más: la invisibilidad de la 113, la posición de la 114, los reintentos automáticos de la 117 y las actividades de la 119. Y lo que la predicción no anticipó es que la solución fue siempre la misma jugada: mover el punto donde ocurre la atomicidad.

```
tabla de salida        el hecho entra en la transacción del cambio
tabla de entrada       la deduplicación entra en la transacción del efecto
clave de idempotencia  el mismo truco, ejecutado por el proveedor externo
```

Predicción 4: «la poda será peligrosa de una forma nueva: un recurso con estado que se borra se recrea vacío».

```
veredicto: NO SE PUDO PROBAR, y el motivo lo confirma
```

No hubo ningún caso, porque la clase 103 ya había puesto la anotación de no-podar sobre volúmenes y bases de datos antes de que la parte 09 empezara. La defensa se construyó antes que el peligro.

Y lo que sí ocurrió tiene la misma forma con otro nombre: una regla automática y declarativa hizo daño sobre datos. No fue la poda: fue la regla de ciclo de vida de la clase 112, que archivó 890 millones de objetos pequeños y multiplicó la factura por cien, con una corrección que costó 1.290 € por la permanencia mínima.

4. La ley 18, el recuento y cómo nos enteramos

Una regularidad ha aparecido cuatro veces en esta parte, en contextos independientes:

```
LEY 18
  Hacer algo asíncrono no elimina ninguna garantía.
  La traslada a la aplicación, que tiene que reconstruirla a mano.
```

Sus cuatro apariciones:

```
clase 113   la llamada síncrona garantizaba «una vez, en orden, y el error
            se ve aquí». Con cola hay que reconstruir las tres.
clase 115   con coreografía, nadie sabe dónde está el pedido: hay que
            construir la vista por sujeto y la vigilancia de plazos.
clase 116   la transacción garantizaba atomicidad; sin ella hay que
            construir tabla de salida, tabla de entrada y conciliación.
clase 119   la pila de llamadas garantizaba el estado del proceso; sin ella
            hay que grabar un historial y reejecutarlo.
```

Y lo que añade al cuestionario de cualquier diseño:

```
¿qué garantizaba la versión síncrona de esto?
¿cuál de esas garantías sigo necesitando?
¿dónde está escrito el código que la reconstruye?
```

La tercera pregunta es la útil: si no hay código que la reconstruya, la garantía no existe, aunque todo el mundo siga contando con ella.

Recuento tras la parte 09, con las leyes de más apariciones:

```
ley 13  el bucle que no corre no da error                        15
        aparece en TODAS las partes desde la 06
ley 15  una señal con demasiados elementos deja de ser señal      12
        parte 09: 41.216 fallidos (113), 209 diferencias, 147 interruptores
ley 16  un control que estorba acaba desactivado o rodeado         9
ley 14  las decisiones de creación son irreversibles               9
        parte 09: clave de partición, número de particiones, motor
ley 11  lo que entra en un sistema de solo-añadir se queda         7
        parte 09: el registro conservado y los datos personales (115)
ley 18  lo asíncrono traslada la garantía, no la elimina           4
        NUEVA en esta parte
ley 17  la medida que se vuelve objetivo se alcanza sin mejorar     4
```

Y el dato incómodo con el que cierra la parte. De los veintiún problemas documentados en las clases 109 a 119, cómo se detectó cada uno:

```
por una caída total                              6
por una reclamación de un cliente                4
por una auditoría o un descuadre contable        4
por la factura                                   2
por una persona que miró y le pareció raro       2
por una alerta                                   3
```

Tres de veintiuno. Y los otros dieciocho ya habían ocurrido cuando alguien se enteró. Esa cifra es la que da sentido a la parte siguiente, y también la que la hipótesis usará como referencia.

5. La hipótesis de la parte 10

La parte 10 cambia el sujeto otra vez: ya no es qué construir, sino cómo se sabe si funciona. La predicción, escrita para poder desmentirla:

1. La parte 10 no introducirá fallos nuevos: hará visibles los que las partes 05 a 09 ya producían.
   → predigo que al menos la mitad de los ejemplos de la parte 10 serán problemas ya documentados en partes anteriores, vistos desde la detección
2. La cifra de referencia es 3 de 21 detectados por alerta.

```
   → predigo que el trabajo de la parte 10 la mueve por encima de 12 de 21,
     y que lo consigue MÁS con alertas sobre síntomas del usuario que con
     alertas sobre causas técnicas

3. La ley dominante será la 15 —una señal con demasiados elementos deja de
   ser señal—, porque la observabilidad consiste precisamente en producir
   señales, y producir de más es su fallo natural.
   → y predigo que el trabajo de la parte 10 REDUCIRÁ el número de alertas,
     no lo aumentará

4. El problema más difícil no será recoger datos ni almacenarlos. Será
   DEFINIR qué significa que el sistema funciona.
   → y predigo que, al revisar las alertas existentes, la mayoría resultará
     medir causas técnicas y no efectos sobre el usuario

5. Y una predicción de coste: la factura de telemetría será un problema
   real, del orden de un porcentaje de dos cifras del coste de cómputo,
   y su causa principal será guardar todo por si acaso.
```

Y lo que hay que anotar ahora para poder calificar sin trampa:

```
lo que ya sabemos     que 18 de 21 problemas se supieron tarde
lo que creemos        que el problema es de definición, no de herramienta
lo que no sabemos     si reducir alertas mejora la detección o la empeora
```

La tercera línea es la que hace que valga la pena escribir esto: es la predicción que puede salir del revés, y si sale del revés será el hallazgo más útil de la parte 10.

Ejemplo trabajado

Se monta el recorrido completo del pedido y se somete a las ocho pruebas negativas. Lo que sigue es el resultado de cada una, y después el recuento de la parte.

El recorrido, cronometrado con un pedido real.

```
POST /v2/pedidos                                    0 ms
  límite de ritmo comprobado                       +2 ms
  transacción: pedido + fila de salida             +11 ms
  respuesta al cliente                              14 ms   ← lo que ve
publicador lee la fila y publica                    +90 ms
consumidor de inventario, efecto aplicado          +140 ms
motor durable inicia la secuencia                  +210 ms
cobro (externo, con clave de idempotencia)         +1,9 s
creación de envío                                  +2,4 s
notificación al cliente                            +2,6 s
evento en el lago                                  +48 s
visible en el panel de analítica                   +3 min
```

Y la lectura importante: el cliente ve 14 ms y el proceso completo tarda 2,6 s. Es la diferencia entre desplegar y activar de la clase 105 aplicada a los datos: responder no es haber terminado.

Las ocho pruebas negativas.

```
1. matar un consumidor a mitad de proceso
   → el mensaje reaparece a los 30 s; la tabla de entrada evita el
     segundo efecto. CORRECTO

2. entregar el mismo mensaje dos veces a propósito
   → 4 consumidores de 5 correctos; el de notificaciones envió dos
     correos: no tenía tabla de entrada. CORREGIDO

3. provocar una conmutación de la base
   → 52 s de recuperación; 31 mensajes reintentados, 0 duplicados
     CORRECTO

4. vaciar el caché en hora punta
   → 90 s de degradación, sin caída; la limitación hacia el origen
     funcionó. CORRECTO

5. parar el publicador de la tabla de salida 30 minutos
   → alerta a los 5 min por antigüedad de la fila más vieja
   → al reanudar, 41.000 filas publicadas en 3 min
   → los consumidores absorbieron el lote sin caer, por el techo de
```

```
     concurrencia. CORRECTO

  6. desplegar código nuevo con instancias del motor a medias
     → 0 instancias rotas gracias a las ramas por versión
     → y la comprobación de reejecución de historiales en la canalización
       había parado un cambio la semana anterior. CORRECTO

  7. enviar un mensaje con un campo renombrado
     → rechazado al producir por el registro de esquemas. CORRECTO

  8. agotar la cuota de un cliente de la API
     → avisos al 50, 80 y 100 %; ritmo reducido en vez de bloqueo
       CORRECTO
```

Siete de ocho a la primera. La que falló es la más instructiva: el consumidor de notificaciones no tenía tabla de entrada porque «enviar un correo no es un efecto de datos». Enviar dos correos sí es un efecto.

Lo que quedó sin resolver, y se documenta como tal.

```
restaurar base y registro al mismo instante
  la base se restaura a un punto; el registro tiene su propia posición
  → tras una restauración, hay que reprocesar desde la posición
    correspondiente y aceptar duplicados, que la idempotencia absorbe
  → probado una vez: 11 min de reproceso, 0 efectos duplicados

el transportista que no acepta clave de idempotencia
  → conciliación diaria; 0-3 descuadres al mes que resuelve una persona

borrar los datos de un cliente concreto del registro conservado
  → se resolvió no publicándolos: el evento lleva identificadores
  → el dato personal vive en la base, donde sí se puede borrar
```

La tercera es la decisión de diseño de la que más se benefició el sistema, y se tomó por un motivo legal, no técnico.

El recuento de la parte 09, con el sistema completo.

```
latencia de la petición de pedido            2,8 s              14 ms
pedidos duplicados                       214 / 3 semanas          0
eventos perdidos                         579 / 6 meses            0
mensajes en cola de fallidos               41.216               0-3
pedidos inconsistentes                     31 / mes              0
conexiones máximas contra la base           1.410                24
coste mensual de consultas analíticas      6.300 €              11 €
tiempo de una consulta histórica           47 min               6 s
desfase peor caso de precios               16 min               ~2 s
instancias de proceso atascadas            31 / mes              0
tiempo para responder «¿dónde está?»       2 h 40               2 s
sistemas de datos en producción                1                5
```

La última fila es el precio, y conviene mirarla junto a las otras doce.

Y el recuento que abre la parte 10.

```
problemas documentados en las clases 109-119               21
detectados por una alerta                                   3
detectados por una caída total                              6
por una reclamación de un cliente                           4
por una auditoría o un descuadre                            4
por la factura                                              2
porque a alguien le pareció raro                            2
```

La conclusión que cierra la parte 09: el sistema funciona, tiene garantías escritas tramo a tramo y pasa siete de ocho pruebas negativas. Y aun así, de los veintiún problemas que hubo que resolver para llegar aquí, dieciocho se supieron cuando ya habían hecho daño. Ninguna de las doce clases de esta parte tenía nada que decir sobre eso, y esa es exactamente la materia de la parte 10.

Laboratorio guiado

Ejecuta desde la raíz:

```
python classes/part-09-data-messaging-serverless-integration/120-proyecto-pipeline-de-pedidos-
orientado-a-eventos/lab.py
```

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

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

Evidencia esperada

El artefacto principal es un incremento integrado, demostrable y documentado. Además, la entrega debe incluir el comando ejecutado, la salida estructurada y una conclusión que no exceda lo observado.

Reto verificable

Construye plataforma-eventos 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 recorrido funciona y nadie sabe qué garantiza cada tramo	Las garantías están en la cabeza de quien lo montó, no escritas	Escribe el recorrido tramo a tramo con lo que se garantiza en cada uno y dónde está el código que lo reconstruye.
Un consumidor duplica efectos porque «lo suyo no son datos»	Se aplicó la idempotencia solo a los efectos sobre la base	Enviar un correo, llamar a un tercero o crear un envío son efectos; todos necesitan tabla de entrada o clave.
Tras restaurar la base, el sistema queda inconsistente con el registro	Se restauran por separado y no coinciden en el mismo instante	Define el procedimiento: restaurar, situar la posición correspondiente y reprocesar aceptando duplicados que la idempotencia absorbe.
Un diseño que funciona en un proveedor choca con un límite en otro	Los máximos de tamaño, retención, particiones y concurrencia son distintos	Comprueba los límites concretos de los tres antes de fijar el diseño, no después.
La hipótesis de la parte anterior se declara acertada sin revisarla	Calificar solo lo que salió bien convierte el aprendizaje en opinión	Publica el veredicto de cada predicción con su evidencia, incluidas las dos que fallaron y la que no se pudo probar.
Se celebra que el sistema es fiable sin mirar cómo se detectan sus fallos	Se mide el resultado y no el medio de detección	Cuenta cómo te enteraste de cada problema: alerta, caída, auditoría, factura o 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. ¿Cuáles son los dos únicos puntos de atomicidad del recorrido y qué se construye alrededor?
2. ¿Qué cuatro mecanismos de las partes 07 y 08 sobrevivieron sin cambios, y cuál mejoró?
3. ¿Por qué la predicción sobre la ley dominante falló y qué ley volvió a dominar?
4. ¿Qué dice la ley 18 y cuál es su pregunta útil?
5. ¿Qué predice la hipótesis de la parte 10 sobre el número de alertas y sobre lo que miden las existentes?

Referencias

• Kleppmann, M. (2017). Designing Data-Intensive Applications, caps. 11 y 12 — procesamiento de flujos y sistemas derivados. <https://dataintensive.net/>

• Richardson, C. (2025). Microservice patterns: data management — tabla de salida, secuencias con compensación y consultas. <https://microservices.io/patterns/data/>

• Helland, P. (2016). Life beyond distributed transactions — diseñar sin transacciones que abarquen varios servicios. <https://queue.acm.org/detail.cfm?id=3025012>

• AWS (2025). Well-Architected: reliability pillar — garantías, pruebas de fallo y recuperación. <https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/welcome.html>

• Google Cloud (2025). Architecture framework: data and event-driven design — elección de almacén por patrón de acceso. <https://cloud.google.com/architecture/framework>

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

Descargar: Parte 09 en PDF · Manual integral

Anterior	Índice	Siguiente
← 119 · Workflows y orquestación durable	Parte 09 · Programa	121 · Logs, métricas, trazas y eventos como señales →

Evaluación — 120 Proyecto: pipeline de pedidos orientado a eventos

Preguntas

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

Reto verificable

Entrega plataforma-eventos 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.