BE SOFTWARE ARCHITECTURE WISE · CAPÍTULO 6

Estilos arquitectónicos II: sistemas distribuidos

Este capítulo lleva la arquitectura a través de la red. Explica cuándo merece la pena pagar el costo de distribuir un sistema, las falacias que lo hacen más difícil de lo que parece y los principales estilos distribuidos: la arquitectura basada en servicios, la arquitectura orientada a servicios, los microservicios, la arquitectura dirigida por eventos, serverless y la arquitectura basada en espacios. Después trata los mecanismos de los que dependen todos ellos: la comunicación síncrona y asíncrona y el diseño de API, los modelos de consistencia y el teorema CAP, y las transacciones distribuidas con sagas y el patrón outbox. Está escrito para desarrolladores y arquitectos que diseñan o revisan sistemas formados por muchos servicios, y es preciso en cuanto a los fallos, las garantías de entrega y la consistencia.

bewiseapps.com/es/software-architecture-wise/distributed-systems/ · Versión 2026-11-03. Solo información educativa, no asesoramiento profesional.

DE UN VISTAZO

Aprenderás a

- Explicar las razones legítimas para distribuir un sistema y los costos que conlleva cada una.
- Describir las ocho falacias de la computación distribuida y el fallo parcial, y diseñar respuestas a ellos.
- Calcular la disponibilidad y la latencia de una cadena de llamadas síncronas y proponer cambios que las mejoren.
- Comparar la arquitectura basada en servicios, la SOA clásica y los microservicios, y elegir entre ellos para una organización concreta.
- Decidir la granularidad de los servicios sopesando los desintegradores frente a los integradores, y reconocer el monolito distribuido.
- Distinguir eventos, comandos y mensajes, las topologías de intermediario (broker) y de mediador, y los cuatro patrones de eventos de Fowler.
- Explicar las garantías de entrega, por qué "exactamente una vez" exige una definición cuidadosa y cómo funcionan los consumidores idempotentes.
- Elegir estilos de comunicación, niveles de consistencia y patrones de transacción (sagas, outbox) para flujos de trabajo distribuidos.

Antes de empezar

Capítulos 0 a 5, en especial la disponibilidad en serie y en paralelo (Capítulo 2, Sección 2.5), el acoplamiento y la ley de Conway (Capítulo 3), y el cuanto de arquitectura, el monolito modular y los contextos delimitados (Capítulo 5).

Tiempo

Unas 3½ horas de lectura; 45 minutos para la prueba.

Términos clave

sistema distribuido · fallo parcial · falacias de la computación distribuida · acoplamiento temporal · arquitectura basada en servicios · bus de servicios empresarial · microservicios · desplegabilidad independiente · base de datos por servicio · arquitectura dirigida por eventos · entrega al menos una vez (at-least-once) · consumidor idempotente · Teorema CAP · saga

Por qué importa este capítulo

El Capítulo 5 describió estilos en los que todo el sistema se despliega como una sola unidad. Este capítulo lleva las mismas preguntas a través de una red. Hoy casi todo sistema de cierto tamaño está distribuido de algún modo: una app para dispositivos móviles se comunica con servicios de backend, que llaman a proveedores de pago, servicios de mapas e intermediarios de mensajes (message brokers), y los propios servicios se ejecutan en muchas máquinas. Dividir un sistema puede aportar escalado independiente, aislamiento de fallos, lanzamientos independientes y equipos autónomos. También trae una clase de problemas que no pueden

darse dentro de un solo proceso: mensajes perdidos o duplicados, servicios que están lentos en lugar de caídos, datos que no coinciden entre copias y fallos en máquinas de las que no sabías que dependías.

La idea más importante que debes llevarte es que **distribuir es un intercambio, no una mejora**. Cada beneficio se paga con latencia, consistencia, observabilidad y esfuerzo operativo, y un buen arquitecto sabe decir qué beneficio se está comprando, a qué precio y por qué no basta con una opción más barata dentro de un monolito. Las falacias de la computación distribuida y la aritmética de las cadenas de llamadas síncronas de la primera sección son las herramientas para poner precio a ese intercambio antes de comprometerse con él.

Cómo encajan las secciones

El capítulo va del porqué al cómo. La Sección 6.1 expone las razones para distribuir, las ocho falacias, el fallo parcial y los costos, y muestra con un ejemplo práctico cómo cinco servicios que cumplen cada uno una disponibilidad del 99.9% dan una ruta de solicitud de solo un 99.5% aproximadamente. Termina volviendo sobre el cuanto de arquitectura, que explica por qué las llamadas síncronas unen entre sí servicios desplegados por separado.

Las tres secciones siguientes son los principales estilos distribuidos, cada uno tratado como los estilos del Capítulo 5: definición y origen, un diagrama, cómo funciona, un ejemplo, una tabla de características, cuándo usarlo y los antipatrones habituales. La Sección 6.2 trata la arquitectura basada en servicios, un término medio pragmático de unos pocos servicios de dominio de grano grueso que a menudo comparten una base de datos, y la clásica arquitectura orientada a servicios de los buses de servicios empresariales y los modelos canónicos, incluido por qué cayó en desuso. La Sección 6.3 trata los microservicios: servicios alineados con contextos delimitados, desplegabilidad independiente, base de datos por servicio, patrones de apoyo como los API gateways y los service meshes, cómo decidir la granularidad, y el monolito distribuido y otros antipatrones. La Sección 6.4 trata la arquitectura dirigida por eventos: eventos y comandos, las topologías de intermediario (broker) y de mediador, las particiones y los grupos de consumidores de Kafka, las garantías de entrega y los consumidores idempotentes.

La segunda mitad pasa a los mecanismos de los que depende todo estilo distribuido. La Sección 6.5 compara la comunicación síncrona y la asíncrona y el acoplamiento que crea cada una, y trata REST, gRPC, GraphQL y la mensajería, las prácticas de diseño de API, el versionado y las pruebas de contrato. La Sección 6.6 trata lo que pueden esperar quienes leen datos replicados: los modelos de consistencia, desde la linealizabilidad hasta la consistencia eventual, el teorema CAP y lo que dice realmente, PACELC y los quórums. La Sección 6.7 explica por qué se evita la confirmación en dos fases entre servicios y cómo las sagas, las transacciones compensatorias y la bandeja de salida transaccional (transactional outbox)

mantienen correctos los procesos de negocio sin ella. La Sección 6.8 cierra con dos estilos para perfiles de carga particulares: las funciones serverless, que escalan hasta cero y se facturan por invocación, y la arquitectura basada en espacios, que mantiene los datos en memoria para sobrevivir a picos extremos como la puesta a la venta de entradas.

Ejemplos recurrentes y cómo leer el capítulo

Hay dos ejemplos que se repiten. La plataforma de reparto de comida de los capítulos anteriores (clientes, restaurantes, repartidores, pagos) muestra el diseño distribuido corriente: aparece como cinco servicios de dominio, como un conjunto de microservicios, como un flujo del evento `OrderPlaced` y como una saga de pedido. Un sistema de venta de entradas para una gira por estadios muestra lo que cambia con una carga extrema y en picos, cuando decenas de miles de personas quieren los mismos pocos asientos en el mismo segundo.

El capítulo tiene un nivel de intermedio a avanzado. Presupone los Capítulos 0 a 5, en particular la disponibilidad en serie y en paralelo (Capítulo 2), el acoplamiento y la ley de Conway (Capítulo 3), y el cuanto de arquitectura, el monolito modular y los contextos delimitados (Capítulo 5). Quienes empiezan pueden leer primero el resumen inicial de cada sección y los recuadros "Si eres nuevo en esto". Los ingenieros con experiencia encontrarán precisión donde importa: lo que la entrega al menos una vez (at-least-once) implica para cada consumidor, lo que "exactamente una vez" significa y lo que no, y por qué la consistencia eventual debe diseñarse en lugar de tolerarse.

Qué llevarte

Tres hábitos recorren todas las secciones. Primero, cuenta las dependencias síncronas, no los servicios: el número de cosas que deben funcionar todas en el mismo momento para que la solicitud de un usuario tenga éxito decide la disponibilidad y la latencia. Segundo, da por hecho que los mensajes se pierden, se retrasan, se duplican y llegan desordenados, y diseña los receptores para que sean idempotentes. Tercero, decide quién es el propietario de cada dato, porque la propiedad decide dónde son posibles las transacciones y dónde hay que gestionar la consistencia entre servicios. El Capítulo 7 desarrolla ese tercer hábito en un tratamiento completo de la arquitectura de datos.

SECCIÓN 6.1 DE 6.8 · INTERMEDIO · 26 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Por qué distribuir y las falacias de la computación distribuida

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.4 Comunicación en red
- 2.5 Disponibilidad y fiabilidad
- 5.1 Estilos y patrones

Términos anteriores que utiliza: Cuanto de arquitectura (architecture quantum), Latencia, Disponibilidad, Ley de Conway, Monolito modular

LA IDEA EN BREVE

Imagina un pequeño restaurante en el que un solo cocinero lo hace todo en una sola cocina. Es sencillo: si el cocinero se pone enfermo, el restaurante cierra, pero nada se pierde entre personas. Ahora imagina una cadena con una panadería central, una cocina de salsas y muchas sucursales unidas por furgonetas de reparto. Puede atender a muchos más clientes, y una cocina puede cerrar sin que cierren las demás. Pero las furgonetas se quedan atascadas en el tráfico, los pedidos se pierden y, cuando una entrega no llega, nadie sabe si la panadería nunca recibió el pedido o si se averió la furgoneta. El software distribuido funciona como la cadena.

LO QUE NECESITAS SABER

- Un sistema distribuido es un conjunto de programas en máquinas distintas que cooperan enviándose mensajes a través de una red para ofrecer lo que los usuarios ven como un único servicio.
- Las razones legítimas para distribuir son la escala, el aislamiento de fallos, la desplegabilidad independiente, la autonomía de los equipos, la geografía y la residencia de datos, las distintas necesidades tecnológicas y el aislamiento de seguridad; cada una tiene un sustituto parcial más barato dentro de un monolito que debe considerarse primero.
- El fallo parcial es el problema definitorio: tras un timeout, quien llama no puede saber si la solicitud se perdió, si falló el destinatario o si solo se perdió la respuesta, por lo que las operaciones que se reintentan deben ser idempotentes.
- Las ocho falacias de la computación distribuida (Deutsch y sus colegas de Sun, con la octava de Gosling) son suposiciones falsas sobre la fiabilidad, la latencia, el ancho de banda, la seguridad, la topología, la administración, el costo del transporte y la

homogeneidad.

- Distribuir cuesta latencia, observabilidad, consistencia y esfuerzo operativo, y esos costos crecen aproximadamente con el número de unidades desplegables.
- En una cadena síncrona las disponibilidades se multiplican y las latencias se suman: cinco servicios al 99.9% dan alrededor del 99.5%, unas 3.6 horas de inactividad al mes.
- Las llamadas síncronas unen los servicios en tiempo de ejecución (acoplamiento temporal), por lo que el número de saltos síncronos en los recorridos clave importa más que el número de servicios.

Qué hace que un sistema sea distribuido

Un sistema distribuido es un conjunto de programas que se ejecutan en computadoras distintas y cooperan, enviándose mensajes a través de una red, para ofrecer lo que los usuarios ven como un único servicio. Tu app de reparto de comida es uno: el teléfono, los servicios de pedidos y de pagos de la empresa, la tableta del restaurante, el teléfono del repartidor y un procesador de tarjetas participan todos en un mismo pedido. El Capítulo 5 describió estilos que se despliegan como una sola unidad (monolitos); este capítulo trata de estilos cuyas partes se despliegan por separado y se comunican a través de la red.

Leslie Lamport, que hizo más que nadie por dar al campo una base matemática, resumió su dificultad esencial en un correo electrónico interno de mayo de 1987: un sistema distribuido es aquel en el que “el fallo de una computadora cuya existencia ni siquiera conocías” puede dejar inutilizable tu propia computadora. La broma funciona porque es verdad. En cuanto un sistema depende de máquinas que no controla y no puede ver, aparece toda una clase de fallos que simplemente no pueden ocurrir dentro de un solo proceso.

Por eso distribuir es una decisión con un precio, no una señal de madurez. El resto de esta sección expone las razones que justifican el precio, las suposiciones falsas que llevan a subestimarlo y una forma de calcular parte de él antes de pagarlo.

Por qué distribuir

Hay unas pocas razones legítimas para repartir un sistema entre procesos y máquinas. Cada una es un atributo de calidad del Capítulo 2 que a una sola unidad de despliegue le cuesta ofrecer a partir de cierto tamaño. Cada una tiene además un sustituto parcial más barato dentro de un monolito, y un buen arquitecto comprueba primero el sustituto.

Razones para distribuir y lo que ya puede hacer un monolito bien construido

Razón	Lo que ofrece la distribución	Lo que ya puede hacer un único despliegue
Escala	Las partes con cargas muy distintas pueden escalarse por separado: el flujo de ubicaciones de los repartidores recibe miles de actualizaciones por segundo; las pantallas de alta de restaurantes, unas pocas por hora.	Ejecutar muchas réplicas idénticas detrás de un balanceador de carga y añadir réplicas de lectura y cachés. Esto basta para la mayoría de los sistemas y es mucho más barato.
Aislamiento de fallos	Una fuga de memoria en las recomendaciones no puede tumbar el proceso de pago, porque se ejecutan en procesos distintos en máquinas distintas.	Los pools de hilos separados y los bulkheads dentro de un mismo proceso dan un aislamiento parcial; un fallo grave sigue tumbándolo todo.
Desplegabilidad independiente	Un equipo puede lanzar su parte veinte veces al día sin coordinar un lanzamiento con nadie más.	Un monolito modular con buena automatización también puede lanzar muchas veces al día, pero cada lanzamiento lo entrega todo.
Autonomía de los equipos	Los equipos son propietarios de un servicio de principio a fin, eligen su ritmo y están de guardia para él; la ley de Conway juega a tu favor en lugar de en tu contra.	La propiedad por módulos funciona bien hasta unos pocos equipos; más allá, una compilación y un lanzamiento compartidos se convierten en una cola.
Geografía y residencia de datos	Ejecutar partes cerca de los usuarios para reducir la latencia, o mantener los datos personales en una jurisdicción porque lo exige una ley o un contrato.	Desplegar una copia completa por región (una célula). Eso ya es en sí una forma de distribución, pero sencilla.
Distintas necesidades tecnológicas	Un modelo de recomendación necesita GPU, la búsqueda necesita un motor de búsqueda y una integración heredada necesita un entorno de ejecución concreto.	Llamar al componente especializado y mantener todo lo demás junto.
Aislamiento de seguridad	Que los datos de tarjetas los gestione un servicio pequeño, reforzado y auditado por separado reduce el alcance de una evaluación PCI DSS.	En realidad no: cualquier cosa dentro del proceso puede alcanzar cualquier otra en memoria.

De la tabla se desprenden dos observaciones. Primera, las razones de más peso suelen ser organizativas: la desplegabilidad independiente y la autonomía de los equipos importan sobre todo cuando muchos equipos trabajan en un mismo producto, y por eso la ley de Conway

(Capítulo 3) reaparece una y otra vez en este capítulo. Segunda, la distribución rara vez es todo o nada. Muchos sistemas de éxito son un monolito modular más dos o tres servicios separados, cada uno desgajado por una razón concreta y explícita.

Fallo parcial: el problema definitorio

Dentro de un proceso, una llamada a una función o bien devuelve un resultado o bien todo el proceso muere con ella. A través de una red hay un tercer resultado, y lo cambia todo: el fallo parcial, en el que algunas partes del sistema funcionan mientras otras han fallado, están lentas o no se pueden alcanzar. Peor aún, a menudo quien llama no puede saber cuál es el caso.

Piensa en el servicio de pedidos pidiendo al servicio de pagos que autorice £24.50. Si no llega respuesta antes del timeout, el servicio de pedidos no sabe si la solicitud se perdió por el camino, si el servicio de pagos se cayó o se bloqueó antes o después de cargar la tarjeta, o si la tarjeta recibió el cargo y solo se perdió la respuesta. Las tres situaciones parecen idénticas desde el lado de quien llama.

Tres fallos que parecen el mismo



Para quien llama, las tres son idénticas: sin respuesta antes del timeout.

Tras un timeout, quien llama no puede saber si el trabajo se hizo; solo los reintentos idempotentes o una consulta posterior del estado pueden resolverlo.

Las consecuencias prácticas determinan todo diseño distribuido. Un reintento podría cobrar dos veces al cliente, así que las operaciones que pueden reintentarse deben hacerse seguras de repetir: esta propiedad es la idempotencia, y normalmente se consigue dando a cada solicitud una clave única que el receptor recuerda. Una dependencia lenta suele ser peor que una caída, porque quienes la llaman siguen esperando y acaparan hilos y conexiones; cuando un servicio se ralentiza, sus llamadores se quedan sin recursos y se ralentizan a su vez, lo que

es un fallo en cascada. Y como una red puede dividirse de modo que dos grupos de máquinas no puedan alcanzarse entre sí mientras ambos siguen funcionando, los datos guardados en los dos sitios pueden divergir: el tema de la sección sobre CAP, más adelante en este capítulo.

Tampoco hay un reloj global. El reloj de cada máquina se desvía y se corrige por separado, de modo que las marcas de tiempo de dos servicios no pueden compararse con total confianza, y "¿qué pasó primero?" puede ser una pregunta difícil. El artículo de Lamport de 1978 sobre el tiempo y los relojes en los sistemas distribuidos mostró cómo ordenar eventos sin depender de relojes físicos; para los arquitectos, la lección es simplemente no basar la corrección en comparar horas de reloj entre máquinas.

SI ESTO ES NUEVO PARA TI

Paso a paso: cuando una respuesta nunca llega

La idea que requiere más atención aquí es el fallo parcial. Dentro de un programa, una llamada a otra parte del programa o bien termina o bien se detiene todo el programa. A través de una red hay una tercera posibilidad: preguntas y no vuelve ninguna respuesta.

- El servicio de pedidos pide al servicio de pagos que cobre £24.50 de la tarjeta de un cliente, y espera la respuesta.
- No puede esperar eternamente, así que se rinde tras un límite de tiempo fijado, llamado timeout: digamos dos segundos. Pasan los dos segundos sin respuesta.
- Pueden haber ocurrido tres cosas distintas, y desde el lado del servicio de pedidos parecen exactamente iguales: la solicitud se perdió por el camino y no se cobró nada; el servicio de pagos se cayó o se bloqueó, y nadie sabe si cargó la tarjeta; o la tarjeta recibió el cargo y solo se perdió la respuesta.
- Si el servicio de pedidos simplemente lo vuelve a intentar, podría cobrar dos veces al cliente. Si se rinde, el cliente puede haber pagado un pedido que nunca se realizó.
- Las salidas seguras son hacer que la solicitud sea idempotente, es decir, que enviarla dos veces tenga el mismo efecto que enviarla una, o preguntar después al servicio de pagos qué pasó realmente. La idempotencia suele funcionar enviando un número de solicitud único con el cobro: el servicio de pagos recuerda los números que ya ha tramitado y nunca cobra dos veces el mismo.
- **La lección:** en una red, el silencio no es una respuesta. Toda llamada remota necesita un límite de tiempo y un plan para cuando no se sabe qué pasó, y por eso este capítulo vuelve una y otra vez a los reintentos y los duplicados.

Las ocho falacias de la computación distribuida

Las falacias de la computación distribuida son ocho suposiciones que hacen quienes se inician en los sistemas distribuidos, todas falsas y todas causantes de problemas tarde o temprano. La lista surgió en Sun Microsystems: L Peter Deutsch redactó siete en 1994, a partir de cuatro que ya habían señalado colegas como Bill Joy, y James Gosling, el creador de Java, añadió la octava hacia 1997. Tres décadas después, cada punto sigue describiendo una caída real en algún lugar esta misma semana.

Las ocho falacias, con una consecuencia y una respuesta de diseño para cada una

Falacia	Qué sale mal si te la crees	Qué hacer en su lugar
1. La red es fiable	Una llamada de pago se queda colgada; el hilo que llama espera eternamente; con carga, pronto todos los hilos están esperando y el servicio de pedidos deja de responder a nadie.	Poner un timeout en toda llamada remota, reintentar solo operaciones idempotentes y poner colas donde el trabajo pueda esperar (patrones de resiliencia).
2. La latencia es cero	Una página del carrito que hacía 40 llamadas dentro del proceso ahora hace 40 llamadas de red de 2 ms cada una; aparecen 80 ms de la nada, y muchos más en la cola.	Diseñar interfaces de grano grueso, agrupar solicitudes en lotes, llamar en paralelo a los servicios independientes y cachear los datos estables.
3. El ancho de banda es infinito	Cada evento `MenuUpdated` lleva todas las fotografías de los platos; una noche de mucho trabajo satura el enlace y retrasa las actualizaciones de ubicación que lo comparten.	Enviar identificadores y los campos que necesitan los consumidores, paginar, comprimir y mantener los datos masivos junto al código que los procesa.
4. La red es segura	Los servicios dentro del centro de datos aceptan cualquier solicitud sin autenticación; un solo contenedor comprometido puede llamar a la API de reembolsos.	Cifrar y autenticar el tráfico entre servicios (por ejemplo, con TLS mutuo) y autorizar cada llamada: zero trust.
5. La topología no cambia	Una dirección IP escrita a fuego en la configuración deja de funcionar cuando el contenedor se reprograma en otra máquina.	Usar descubrimiento de servicios y nombres DNS, y contar con que las instancias aparezcan y desaparezcan a cada minuto.
6. Hay un solo administrador	Un proveedor de nube, el equipo de firewall de un socio u otro departamento cambia una regla, rota un certificado o baja un límite de tasa, y tus llamadas empiezan a fallar.	Saber quién controla cada salto, monitorizar las dependencias que no son tuyas y acordar contratos y avisos de cambio.
7. El costo del transporte es cero	La serialización consume CPU, y los datos que cruzan zonas de disponibilidad, regiones o la internet pública se facturan por gigabyte; la factura de red crece más rápido que el tráfico.	Medir el costo de los flujos de datos, mantener local el tráfico muy conversacional y elegir formatos compactos donde el volumen es alto.
8. La red es homogénea	Un servicio supone que todos los clientes hablan la misma versión del protocolo, la misma codificación de caracteres y el mismo reloj; una red móvil, un sistema antiguo de un socio o un proxy rompe esa suposición.	Usar formatos y protocolos estándar con versiones explícitas, y probar la interoperabilidad con los clientes reales.

DE DÓNDE VIENE

La lista sigue creciendo

En una entrevista de 2021, Deutsch respaldó una novena falacia propuesta, que describió como una extensión de la cuarta: que la parte con la que te comunicas es de confianza. Mark Richards y Neal Ford, en la segunda edición de *Fundamentals of Software Architecture* (2025), propusieron tres más extraídas de proyectos de microservicios: el versionado es fácil, las actualizaciones compensatorias siempre funcionan y la observabilidad es opcional. Las tres vuelven a aparecer más adelante en este capítulo: el versionado en comunicación y API, la compensación en sagas y la observabilidad más abajo.

Lo que cuesta distribuir

Toda razón para distribuir se paga con la misma moneda. La **latencia** es el primer costo. Una llamada a una función dentro de un proceso tarda nanosegundos; un viaje de ida y vuelta por la red entre máquinas del mismo centro de datos suele tardar desde unos cientos de microsegundos hasta un milisegundo o más, antes de cualquier serialización o trabajo. Entre continentes, la física pone el límite inferior: la luz en la fibra óptica viaja a unos 200 000 km por segundo, así que un viaje de ida y vuelta entre Londres y Nueva York (unos 5600 km en cada sentido) no puede tardar menos de unos 56 ms. La aritmética de la latencia de cola del Capítulo 2 se aplica a cada salto.

La **observabilidad** es el segundo. En un monolito, una traza de pila (stack trace) cuenta toda la historia de una solicitud fallida. En un sistema distribuido, la historia está repartida entre los logs de muchos servicios en muchas máquinas. La observabilidad, la capacidad de entender lo que está haciendo el sistema a partir de sus salidas, tiene que diseñarse desde el principio: identificadores de correlación transmitidos en cada llamada, logs centralizados, métricas por servicio y trazado distribuido (Capítulo 9).

La **consistencia** es el tercero. Una sola transacción de base de datos puede actualizar un pedido y su pago de forma atómica. Cuando el pedido y el pago viven en servicios distintos con bases de datos distintas, no hay una transacción compartida; el sistema debe aceptar un periodo en el que los dos no coinciden y debe asegurarse de que converjan. Las secciones posteriores tratan las herramientas: los modelos de consistencia y las sagas y el patrón outbox.

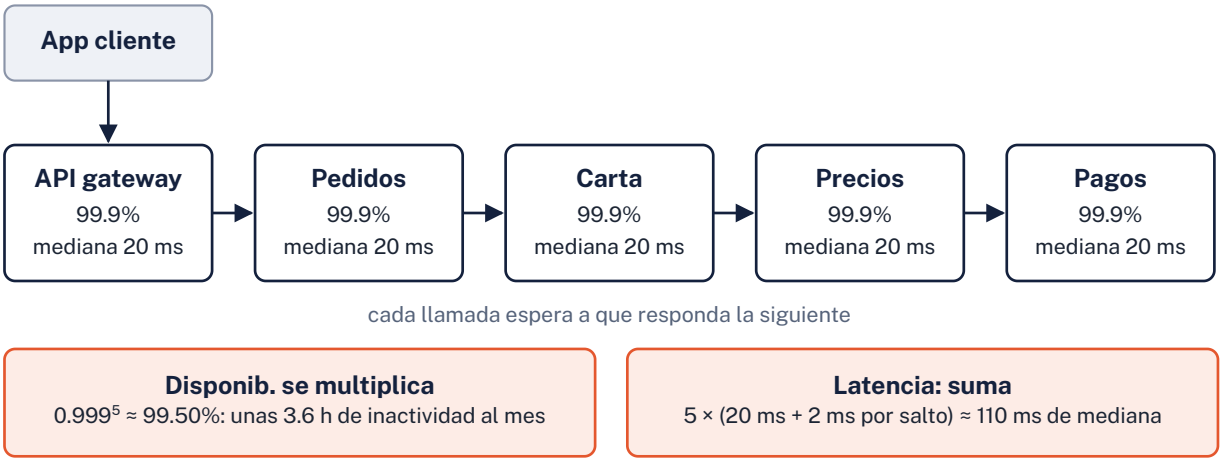
La **carga operativa** es el cuarto, y el que más a menudo se subestima. Cada servicio necesita un pipeline de compilación, un despliegue, configuración, secretos, monitorización, alertas, planificación de capacidad, parches de seguridad y alguien de guardia. Las pruebas también cambian: un cambio solo es seguro si todos los consumidores de su interfaz siguen

funcionando, lo que exige pruebas de contrato y cambios compatibles hacia atrás. Estos costos son aproximadamente proporcionales al número de unidades desplegables, y por eso importa el número de servicios.

Hacer las cuentas: las cadenas síncronas

El momento más barato para descubrir que un diseño distribuido es demasiado frágil o demasiado lento es sobre el papel. La herramienta es la regla de los componentes en serie del Capítulo 2: si una solicitud necesita que funcionen todos los componentes de una cadena, la disponibilidad de la cadena es el producto de las disponibilidades de los componentes, y su latencia es aproximadamente la suma de las suyas.

Una cadena síncrona de cinco servicios



Cinco servicios que cumplen cada uno el 99.9% producen una ruta de solicitud que solo cumple alrededor del 99.5%, y con cinco veces la latencia de uno.

EJEMPLO PRÁCTICO

Ejemplo práctico: el presupuesto de disponibilidad y de latencia de una cadena de llamadas

Una plataforma de reparto de comida gestiona **Hacer pedido** con una cadena síncrona: el API gateway llama al servicio de pedidos, que llama al servicio de carta, que llama al de precios, que llama al de pagos. Cada servicio por sí solo cumple una disponibilidad del 99.9% (medida en tiempo, sobre un mes de 30 días de 43 200 minutos), responde con una mediana de 20 ms de trabajo propio y tiene un p99 de 250 ms. Cada salto de red añade unos 2 ms. Supón, con generosidad, que los fallos son independientes y que cualquier fallo hace fallar toda la solicitud.

Un servicio	Disponibilidad 0.999 = 99.9%; inactividad permitida 0.001 × 43 200 = 43.2 minutos al mes.
Dos en serie	0.999 × 0.999 = 0.998 (99.8%); unos 86 minutos al mes.
Cinco en serie	$0.999^5 \approx 0.99501$ (99.50%); no disponible unos 0.005 × 43 200 ≈ 216 minutos, o 3.6 horas, al mes. Cinco servicios respetables por separado han producido un sistema de dos nueves y medio.
Diez en serie	$0.999^{10} \approx 0.99004$ (99.0%); unos 430 minutos, más de 7 horas, al mes.
Latencia mediana, cinco en serie	Las latencias se suman: aproximadamente 5 × (20 ms + 2 ms) ≈ 110 ms, frente a 20 ms para el mismo trabajo hecho en un solo proceso. (En sentido estricto, las medianas no se suman exactamente; la suma es una buena primera estimación.)
Latencia de cola, cinco en serie	La probabilidad de que una solicitud se encuentre al menos un salto en su 1% más lento es $1 - 0.99^5 \approx 4.9\%$, así que alrededor de 1 solicitud de cada 20 se ve ralentizada por al menos un salto de 250 ms. Con diez saltos es $1 - 0.99^{10} \approx 9.6\%$.
Lo que hace falta para mantener el 99.9% de extremo a extremo	Cada uno de los cinco servicios necesitaría alrededor del 99.98% ($0.9998^5 \approx 0.9990$): unos 8.6 minutos de inactividad al mes cada uno, lo que en la práctica significa recuperación automatizada.
Respuestas de diseño	Llamar a carta y a precios en paralelo (la latencia pasa a ser la del más lento de los dos, no la suma); cachear los datos de la carta en el servicio de pedidos; hacer asíncrona la confirmación del pago para que el pedido se acepte y se pague poco después; eliminar los saltos que solo reenvían solicitudes.

El Capítulo 2 presentó la regla de que las disponibilidades se multiplican en serie; la distribución es donde esa regla empieza a doler, porque cada llamada de red añade un componente a la serie.

La suposición de independencia embellece el resultado. Los servicios que comparten una base de datos, un clúster, un sistema de configuración o un tren de lanzamientos fallan juntos, y un destinatario lento puede agotar los hilos de quien lo llama y propagar la caída hacia arriba.

La solución rara vez es hacer cada servicio heroicamente fiable. Consiste en acortar las cadenas síncronas y dejar que las partes que no necesitan responder de inmediato trabajen a partir de una cola.

El cuanto de arquitectura, de nuevo

El Capítulo 5 presentó el cuanto de arquitectura: una parte desplegable de forma independiente, con alta cohesión funcional e incluidos sus datos, a la que se aplica un único conjunto de características de arquitectura (Estilos, patrones y tácticas arquitectónicos). En *Software Architecture: The Hard Parts* (2021), Ford, Richards, Sadalage y Dehghani afinan la idea con dos tipos de acoplamiento. El acoplamiento estático es lo que debe estar presente para que una parte se despliegue y se ejecute: su código, sus bibliotecas, su base de datos y su infraestructura. El acoplamiento dinámico es cómo dependen las partes unas de otras durante la ejecución: si se llaman de forma síncrona o asíncrona, y cómo se coordinan y se mantienen consistentes.

El ejemplo práctico muestra por qué importa el lado dinámico. Mientras dura una llamada síncrona, quien llama no puede completar su trabajo a menos que el destinatario esté disponible y responda rápido; los dos quedan unidos operativamente, y su disponibilidad y su latencia se combinan como se calculó más arriba. Cinco servicios desplegados por separado en una cadena síncrona son cinco unidades de despliegue, pero para la disponibilidad de **Hacer pedido** se comportan como una sola unidad frágil. Esta dependencia de que la otra parte esté disponible en el mismo momento es el acoplamiento temporal. La mensajería asíncrona elimina gran parte de él: si el servicio de pedidos deja un mensaje en una cola duradera, el servicio de pagos puede estar caído un minuto y ponerse al día después, y el cliente sigue recibiendo un número de pedido. El costo es que el cliente no recibe de inmediato una respuesta definitiva, lo que es tanto una decisión de producto como técnica.

Esta es la lección más profunda de la sección. Trazar los límites de los servicios es la mitad del diseño; decidir qué interacciones deben ser síncronas, y mantener pequeño ese conjunto, es la otra mitad.

ATENCIÓN

Errores comunes al decidir distribuir

- **Distribuir para arreglar un problema de calidad del código.** Un monolito enmarañado dividido en servicios se convierte en un sistema distribuido enmarañado, con fallos de red añadidos. Arregla primero los límites; el monolito modular del Capítulo 5 es el camino habitual.
- **Tratar las llamadas remotas como si fueran locales.** Las bibliotecas que hacen que una llamada remota parezca una llamada a un método ocultan la latencia y los fallos; la "primera ley del diseño de objetos distribuidos" de Martin Fowler era sencillamente no distribuir tus objetos.
- **Contar servicios en lugar de saltos síncronos.** Diez servicios que sobre todo intercambian eventos pueden ser más robustos que tres que se llaman entre sí en cadena.
- **Suponer que los fallos son independientes.** Los servicios que comparten una base de datos, un clúster de Kubernetes o un almacén de configuración fallan juntos; la fórmula en serie resulta entonces optimista, y la fórmula en paralelo, ficción.
- **Ignorar el costo en personas.** Diez servicios necesitan diez juegos de pipelines, paneles y turnos de guardia. Si un solo equipo de seis personas es propietario de todos ellos, el principal beneficio de la distribución, la autonomía de los equipos, no existe, mientras que todos los costos siguen ahí.

Lista de comprobación: antes de distribuir

Usa estas preguntas antes de repartir un sistema, o una parte de él, a través de la red.

- [ ] **Nombra el impulsor**
  ¿Qué razón de la tabla anterior se aplica, a qué parte y con qué evidencia? "Los microservicios son modernos" no es un impulsor.
- [ ] **Prueba primero la solución más barata**
  ¿Podrían cubrir la necesidad unas réplicas, una caché, una réplica de lectura, un proceso en segundo plano o un límite de módulo más claro dentro del monolito?
- [ ] **Dibuja el grafo de llamadas**
  Cuenta los saltos síncronos de cada recorrido clave del usuario y calcula el presupuesto de disponibilidad y de latencia, como en el ejemplo práctico.
- [ ] **Decide quién es el propietario de qué datos**
  Cada dato necesita un único servicio propietario; todo lo que deba cambiar de forma atómica con él probablemente pertenezca al mismo servicio.
- [ ] **Diseña para el fallo parcial**
  Timeouts, reintentos con idempotencia, alternativas (fallbacks) y lo que ve el usuario cuando una dependencia está caída, decididos antes de la primera caída.
- [ ] **Comprueba los requisitos previos**
  El aprovisionamiento y el despliegue automatizados, la monitorización, los logs centralizados y el trazado distribuido deben existir, o estar planificados y financiados.
- [ ] **Ajusta los equipos a los límites**
  Cada servicio debe tener un único equipo propietario que pueda cambiarlo, desplegarlo y darle soporte sin pedir permiso a otros.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ Un servicio llama a un servicio de inventario sin timeout. El servicio de inventario se vuelve muy lento en lugar de fallar. ¿Por qué puede ser esto peor que si se hubiera caído?

Un servicio caído falla rápido y quienes lo llaman pueden reaccionar; uno lento los mantiene esperando, de modo que sus hilos y conexiones se llenan hasta que también dejan de responder, y la caída se propaga como un fallo en cascada.

▼ **Tres servicios tienen cada uno una disponibilidad del 99.95% y se llaman en serie en cada solicitud. ¿Cuál es la disponibilidad aproximada de la ruta de solicitud?**

$0.9995^3 \approx 0.9985$, alrededor del 99.85%, suponiendo fallos independientes.

▼ **¿Por qué importa la falacia "el costo del transporte es cero" aunque la latencia y el ancho de banda estén bien?**

Mover datos cuesta CPU para la serialización y dinero por el tráfico entre zonas, regiones o proveedores, así que un diseño muy conversacional o voluminoso puede salir caro aunque sea lo bastante rápido.

▼ **¿Por qué cinco servicios desplegados por separado pueden comportarse aún como una sola unidad a efectos de disponibilidad?**

Si una solicitud los llama de forma síncrona en cadena, los cinco deben estar disponibles en el mismo momento; sus disponibilidades se combinan, de modo que están acoplados temporalmente pese a desplegarse por separado.

▼ **Inténtalo: Hacer pedido llama de forma síncrona a cuatro servicios, cada uno con un 99.9% de disponibilidad. Calcula la disponibilidad y el tiempo de inactividad mensual (43 200 minutos). Si la notificación y el despacho pasan a ser asíncronos y quedan dos síncronos, ¿qué cambia? ¿Qué debes añadir si un reintento sigue a un timeout?**

Cuatro: 0.999^4 = 0.99601, alrededor del 99.6%, o 173 minutos al mes. Dos: 0.999^2 = 0.99800, unos 86 minutos, aproximadamente la mitad, suponiendo fallos independientes. Tras un timeout, el llamador no puede saber si el trabajo se hizo, de modo que las llamadas reintentadas deben ser idempotentes, por ejemplo con una clave de idempotencia, o a un cliente se le podría cobrar dos veces. Los pasos asíncronos necesitan colas monitorizadas, pero ya no detienen el pedido.

FUENTES

- Leslie Lamport: the 1987 email defining a distributed system (lamport.org): https://lamport.org/pubs/distributed-system.txt
- The Eight Fallacies of Distributed Computing (James Gosling's page): https://nighthacks.com/jag/res/Fallacies.html
- Software Engineering Radio episode 470: L Peter Deutsch on the fallacies of distributed computing (2021): https://se-radio.net/2021/07/episode-470-l-peter-deutsch-on-the-fallacies-of-distributed-computing/
- Martin Fowler: Microservices and the First Law of Distributed Objects: https://martinfowler.com/articles/distributed-objects-microservices.html

- Google SRE book: Addressing Cascading Failures: https://sre.google/sre-book/addressing-cascading-failures/

SECCIÓN 6.2 DE 6.8 · INTERMEDIO · 22 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Arquitectura basada en servicios y arquitectura orientada a servicios

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.5 Almacenar datos
- 5.3 Monolito modular
- 6.1 Por qué distribuir

Términos anteriores que utiliza: ACID, Esquema, Cuanto de arquitectura (architecture quantum), Particionamiento por dominio, Contexto delimitado (bounded context)

LA IDEA EN BREVE

Piensa en un hospital que se queda pequeño para un solo edificio. Podría partirlo todo en cientos de clínicas diminutas, cada una con su propia recepción y su propio archivo de historiales, o podría trasladarse a cinco departamentos en alas separadas: maternidad, cirugía, consultas externas, farmacia y urgencias. Cada ala tiene su propio personal y puede reformarse sin cerrar las demás, pero todas siguen compartiendo un archivo central de historiales. Eso es la arquitectura basada en servicios: unas pocas partes grandes y separadas, cada una encargada de toda un área del negocio, que a menudo siguen compartiendo una base de datos.

LO QUE NECESITAS SABER

- La arquitectura basada en servicios (bautizada así por Richards y Ford) divide un sistema en unos pocos servicios de dominio de grano grueso desplegados por separado, normalmente de 4 a 12, que suelen compartir una base de datos y a menudo están detrás de una capa de API.
- Como la mayoría de los flujos de trabajo se quedan dentro de un servicio de dominio, pueden usar transacciones ACID locales, y los servicios rara vez se llaman entre sí, lo que mantiene cortas las cadenas síncronas.
- Una base de datos compartida acopla los cambios de esquema y une los servicios en un solo cuanto; los esquemas por servicio, las pequeñas bibliotecas de entidades por dominio y las bases de datos separadas cuando está justificado reducen ese acoplamiento.

- La SOA clásica dirigida por la orquestación organizaba los servicios en capas técnicas en torno a un bus de servicios empresarial, con un modelo de datos canónico y los estándares WS-*, con el objetivo de la reutilización en toda la empresa.
- La SOA decepcionó porque la reutilización creó acoplamiento, el bus se convirtió en un cuello de botella propiedad de un solo equipo, los modelos canónicos no le servían a nadie y los estándares eran pesados.
- Los microservicios mantienen contratos de servicio explícitos, pero ponen la lógica en los servicios (smart endpoints, dumb pipes: extremos inteligentes, tuberías tontas), se organizan por capacidad de negocio y dan a cada servicio sus propios datos.
- La arquitectura basada en servicios es un término medio pragmático para unos pocos dominios y equipos, y una forma de bajo riesgo de salir de un monolito.

Entre el monolito y los microservicios

Gran parte de la conversación del sector plantea una elección entre un monolito y muchos microservicios. En la práctica, una gran parte de los sistemas reales está en un punto intermedio: un puñado de servicios desplegados por separado, cada uno de los cuales cubre toda un área de negocio, que a menudo siguen compartiendo una base de datos. Mark Richards y Neal Ford llamaron a este estilo arquitectura basada en servicios en *Fundamentals of Software Architecture* y sostienen que es uno de los estilos distribuidos más pragmáticos que existen. Esta sección lo describe, lo contrasta con la arquitectura orientada a servicios (SOA), más antigua y con la que a menudo se confunde, y muestra en qué se diferencian los microservicios de ambas.

Arquitectura basada en servicios: la topología

Un sistema basado en servicios tiene tres partes. Una o varias interfaces de usuario llaman a un conjunto de servicios de dominio desplegados por separado, cada uno de los cuales cubre un área de negocio grande y coherente, como los pedidos o los pagos. Richards y Ford observan que estos sistemas suelen tener entre 4 y 12 servicios de dominio, unos 7 de media. Los servicios suelen compartir una base de datos, y una capa de API opcional (un proxy inverso o un gateway) puede situarse entre las interfaces de usuario y los servicios para encargarse del enrutamiento, la autenticación y la limitación de tasa.

Cada servicio de dominio es de grano grueso: contiene todo lo necesario para su dominio, desde la API hasta la lógica de negocio y el acceso a los datos, y por dentro está estructurado como un pequeño monolito, a menudo en capas o en módulos (Capítulo 5). Un servicio se despliega normalmente como unas pocas instancias detrás de un balanceador de carga.

De ello se derivan dos consecuencias. Primera, la mayoría de los flujos de trabajo de negocio se ejecutan por completo dentro de un servicio, de modo que pueden usar una transacción local corriente de base de datos con garantías ACID completas (Capítulo 0). Segunda, los servicios de dominio rara vez necesitan llamarse entre sí: una solicitud de la interfaz de usuario va al único servicio que es propietario del dominio. Esto mantiene cortas las cadenas síncronas, lo que importa por las razones calculadas en la Sección 6.1.

SI ESTO ES NUEVO PARA TI

Paso a paso: cómo una base de datos compartida une los servicios

El punto en el que más gente tropieza aquí es que unos servicios desplegados por separado pueden seguir unidos por la base de datos que comparten. Un pequeño ejemplo muestra cómo.

- Supón que los servicios de la plataforma de reparto de comida leen libremente las tablas de los demás. Pedidos, Restaurante y Reparto leen todos la tabla de restaurantes de la única base de datos compartida.
- El equipo de Restaurante quiere dividir la columna de dirección de la tabla en columnas separadas de calle, localidad y código postal, para facilitar la búsqueda por código postal. En su propio código es un cambio de un día.
- Pero Pedidos y Reparto también leen la columna de dirección. En el momento en que desaparece, se rompen, aunque nadie haya tocado su código.
- Así que los tres servicios deben cambiarse, probarse y lanzarse juntos. Los servicios unidos por tablas compartidas no pueden desplegarse de forma independiente, y por eso la sección dice que forman un solo cuanto de arquitectura.
- El remedio principal de esta sección es dar a cada servicio su propio conjunto de tablas, su propio esquema, que ningún otro servicio puede tocar. Pedidos y Reparto leen entonces las direcciones a través de una vista de solo lectura que publica Restaurante, y Restaurante puede reorganizar libremente sus propias tablas siempre que esa vista mantenga la misma forma.
- **La lección:** compartir una base de datos es lo que hace que este estilo sea barato y sencillo, pero cada tabla que leen varios servicios es una promesa que varios equipos deben cumplir juntos. Da a cada tabla un único servicio propietario y comparte solo unas pocas vistas publicadas.

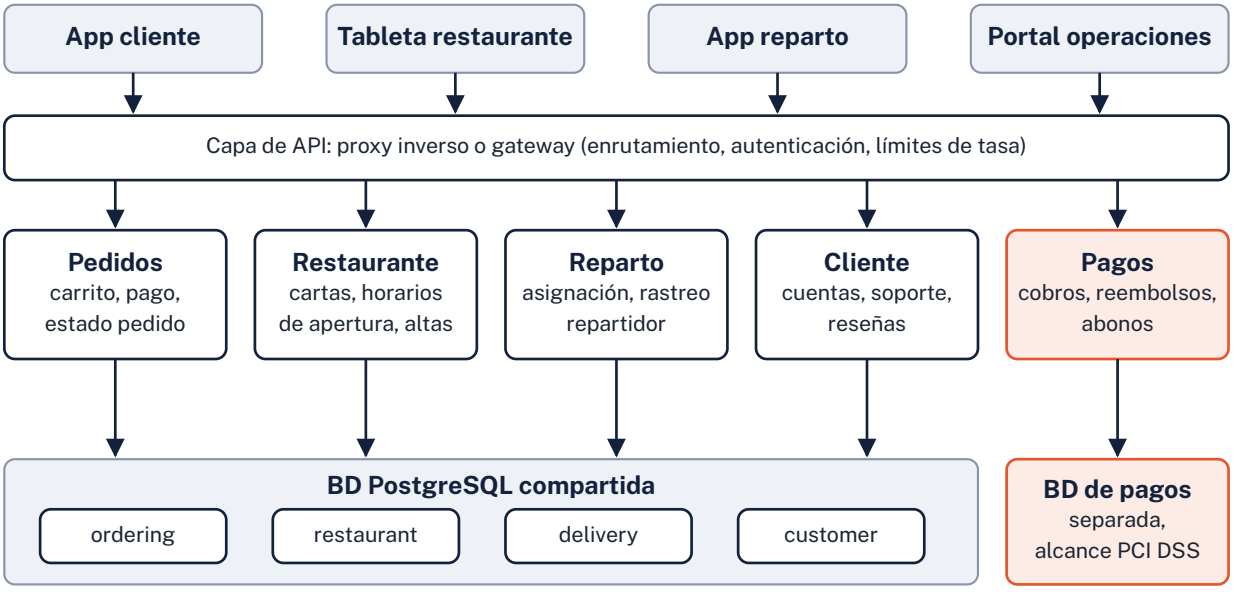
La base de datos compartida y cómo domarla

Compartir una base de datos es lo que hace que la arquitectura basada en servicios sea barata y sencilla: no hay sincronización de datos entre servicios, no hay consistencia eventual y los informes se hacen con SQL corriente. También es su principal debilidad. Si todos los servicios

leen y escriben las mismas tablas, un cambio en las columnas de una tabla puede romper varios servicios, que entonces deben probarse y volver a desplegarse juntos. Se aplica la definición del Capítulo 5: los servicios que comparten una base de datos quedan unidos en un solo cuanto de arquitectura, de modo que comparten un único nivel de disponibilidad y un único ritmo de lanzamiento del esquema.

Los profesionales usan tres técnicas para reducir el acoplamiento. La primera es el particionado lógico: dar a cada servicio su propio esquema (o conjunto de tablas) y prohibirle tocar el de los demás, algo que se hace cumplir con permisos de la base de datos además de por convención. La segunda es mantener las clases de entidad de la base de datos compartida en pequeñas bibliotecas por dominio en lugar de en una gran biblioteca compartida, de modo que un cambio en las tablas de restaurantes solo afecte a los servicios que las usan. La tercera es dar a un servicio su propia base de datos cuando hay una razón de peso, como el alcance de seguridad, un patrón de carga muy distinto o un tipo distinto de almacén de datos. Los equipos que las aplican con constancia descubren que un paso posterior a bases de datos separadas, si alguna vez llega a ser necesario, es mucho más fácil.

La plataforma de reparto de comida como cinco servicios de dominio



Cada servicio tiene su propio esquema; solo Pagos tiene una base de datos aparte

Cinco servicios de grano grueso, cada uno propiedad de un equipo, detrás de una capa de API; cuatro comparten una base de datos con un esquema cada uno, y Pagos mantiene la suya propia.

EJEMPLO PRÁCTICO

Ejemplo práctico: la plataforma de reparto de comida como cinco servicios de dominio

Una empresa de reparto de comida ha pasado de un equipo a cinco, unos 35 ingenieros. Su monolito en Django está bien modularizado, pero los lanzamientos se han convertido en una cola y un fallo en el seguimiento de repartidores tumbó dos veces el proceso de pago el trimestre pasado. El arquitecto propone una arquitectura basada en servicios con cinco servicios de dominio, uno por cada módulo y equipo existentes.

Pedidos	Carrito, proceso de pago, ciclo de vida y estado del pedido. Es propietario del esquema `ordering`. Mucho trabajo a las horas de comer; ejecuta de 6 a 20 instancias.
Restaurante	Cartas, precios, horarios de apertura, alta de restaurantes. Predominan las lecturas; las cartas se cachean en la capa de API. Ejecuta 3 instancias.
Entrega	Despacho de repartidores, estimaciones de ruta, seguimiento en vivo. Recibe actualizaciones de ubicación cada pocos segundos de cada repartidor activo, así que se escala por separado de todo lo demás.
Pagos	Autorización de tarjetas a través del proveedor de pagos, reembolsos, pagos a restaurantes y repartidores. Tiene su propia base de datos para que el alcance de la evaluación PCI DSS sea un servicio pequeño en lugar de toda la plataforma.
Cliente	Cuentas, direcciones, incidencias de soporte, reseñas. Poca carga, pero contiene la mayor parte de los datos personales, así que las reglas de acceso y de conservación se concentran aquí.
Cómo se ejecuta un flujo de trabajo	**Hacer pedido** se ejecuta dentro de Pedidos en una transacción local (crear el pedido, reservar la franja del restaurante) y después hace una llamada a Pagos. Restaurante y Reparto se enteran del pedido por un mensaje, no por una cadena de llamadas.
Qué se comparte	Un servidor PostgreSQL con un esquema por servicio. Cada servicio solo puede leer y escribir su propio esquema; se publican unas pocas vistas de solo lectura (los nombres de los restaurantes, por ejemplo) para los demás.
Qué aporta	Cinco pipelines de despliegue en lugar de uno; un fallo en el seguimiento ya no detiene el proceso de pago; los pagos pueden auditarse por separado; cada equipo lanza cuando está listo.
Qué cuesta	Cinco desplegables y paneles en lugar de uno; llamadas de red entre Pedidos y Pagos que necesitan timeouts e idempotencia; los cambios de esquema siguen coordinándose a través del servidor de base de datos compartido.

El paso es modesto a propósito. Pasa de un cuanto a unos dos (Pagos con sus propios datos, y los cuatro servicios que comparten la base de datos), y da a cada equipo su propio lanzamiento, que era el principal problema.

Si más adelante el flujo de ubicaciones de Reparto se queda grande para la base de datos compartida, se puede dar a ese único servicio su propio almacén y dividirlo más; el resto puede quedarse como está.

El arquitecto deja por escrito las reglas de los límites como pruebas de arquitectura: ningún servicio puede consultar el esquema de otro, y los servicios solo pueden llamarse entre sí a través de API publicadas.

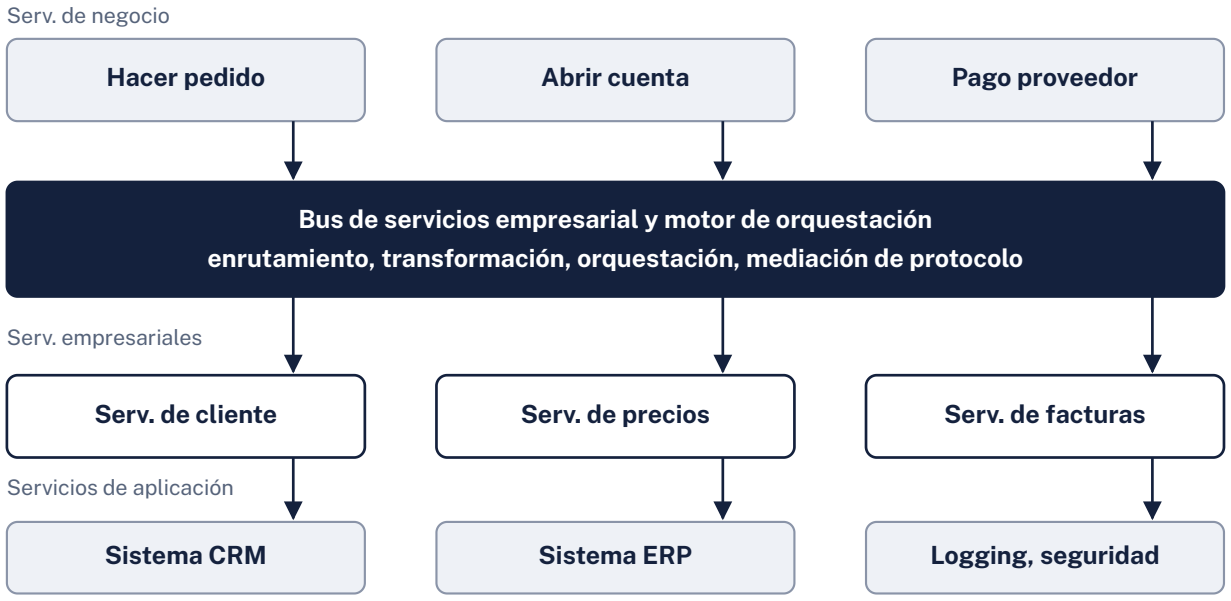
La arquitectura orientada a servicios clásica

La arquitectura orientada a servicios es más antigua y más amplia. El término entró en el sector a través de un informe de Gartner de 1996 de Roy Schulte y Yefim Natis, y desde principios de la década de 2000 se convirtió en el modelo dominante de integración empresarial. Su idea central era sensata: exponer las capacidades de negocio como servicios con contratos explícitos y neutrales respecto a la tecnología, de modo que las aplicaciones de una gran organización puedan montarse a partir de ellos en lugar de reconstruirse o integrarse punto a punto.

Sin embargo, lo que la mayoría de las empresas construyó en realidad fue lo que Richards y Ford llaman SOA dirigida por la orquestación. Los servicios se organizaban en capas técnicas: servicios de negocio que describían procesos (hacer un pedido, abrir una cuenta), servicios empresariales compartidos que ofrecían capacidades reutilizables (cliente, precios, factura), y servicios de aplicación y de infraestructura que envolvían los sistemas existentes y las inquietudes técnicas. En el centro estaba un bus de servicios empresarial (ESB): un middleware que enrutaba los mensajes, los transformaba entre formatos, mediaba entre protocolos y, mediante un motor de orquestación, ejecutaba los propios procesos de negocio.

Dos ideas más completaban el cuadro. Un modelo de datos canónico definía una única representación de cada entidad de negocio para toda la empresa (un Customer, un Order), y cada mensaje se traducía a ella y desde ella en el bus. Y una familia de estándares de servicios web, normalmente llamada WS-*, definía cómo los servicios se describían y se llamaban entre sí: SOAP para los mensajes, WSDL para las descripciones de interfaces, UDDI para los registros, WS-Security, WS-ReliableMessaging y WS-AtomicTransaction para las cualidades, y WS-BPEL (versión 2.0 aprobada por OASIS en 2007) para las definiciones de procesos ejecutables.

SOA dirigida por la orquestación



En la SOA clásica, el bus contenía la lógica de enrutamiento, de transformación y de procesos, y todos los servicios compartían un único modelo canónico del negocio.

Por qué la SOA clásica cayó en desuso

A finales de la década de 2000, muchos programas de SOA habían decepcionado. Anne Thomas Manes, de Burton Group, captó el ambiente en una entrada muy leída de enero de 2009 titulada *SOA is Dead; Long Live Services*: la etiqueta se había vuelto tóxica, pero la idea de los servicios sobrevivió. Merece la pena conocer las razones, porque cada una es una trampa en la que las arquitecturas modernas pueden volver a caer.

La reutilización creó acoplamiento. Un servicio empresarial compartido de Customer usado por cuarenta aplicaciones no podía cambiar sin el acuerdo de cuarenta consumidores; la reutilización, el objetivo principal, convirtió cada cambio en una negociación. **El bus se convirtió en el cuello de botella.** Poner la lógica de enrutamiento, de transformación y de procesos en un ESB central, normalmente propiedad de un único equipo de integración, significaba que casi cualquier cambio de negocio necesitaba a ese equipo, y el bus se convirtió en un único punto de fallo y de retraso. **El modelo canónico era demasiado grande.** Una única definición de Customer que satisfaga a ventas, facturación, soporte y logística o es enorme o es errónea para la mayoría de ellos; los contextos delimitados del Capítulo 5 explican por qué un término significa legítimamente cosas distintas en distintas partes de un negocio. **Los estándares eran pesados.** Las especificaciones WS-* eran complejas, las herramientas eran caras y las transacciones distribuidas entre servicios resultaron frágiles. **El particionamiento técnico** significaba que una funcionalidad sencilla atravesaba varias capas de servicios propiedad de grupos distintos.

Las lecciones duraderas son aquellas sobre las que se construyeron los microservicios: contratos de servicio explícitos, servicios cuyos propietarios son las personas que los construyen, la lógica en los servicios y no en el middleware, y los datos propiedad de un único servicio en lugar de compartidos por toda la empresa.

En qué se diferencian los microservicios

Los microservicios, que se tratan a fondo en la sección siguiente, conservan la idea de la SOA de servicios con contratos explícitos e invierten la mayoría de sus decisiones de implementación. James Lewis y Martin Fowler resumieron una de esas inversiones como *smart endpoints and dumb pipes* (extremos inteligentes y tuberías tontas): la red solo transporta mensajes, y toda la lógica vive en los servicios. Los servicios se organizan en torno a capacidades de negocio en lugar de capas técnicas, la reutilización queda deliberadamente por detrás de la independencia, cada servicio es propietario de sus datos y la gobernanza está descentralizada.

Comparados con la arquitectura basada en servicios, los microservicios son de grano más fino, mucho más numerosos e insisten en tener datos separados. Eso compra más independencia, un escalado más selectivo y más aislamiento de fallos, y cuesta mucho más en operación, consistencia y coordinación. La arquitectura basada en servicios puede verse como el punto del espectro en el que llega la mayor parte de los beneficios de despliegue y de equipos por una fracción del costo.

Comparación de la arquitectura basada en servicios, la SOA clásica y los microservicios

Aspecto	Basada en servicios	SOA clásica (dirigida por la orquestación)	Microservicios
Objetivo principal	Modularidad pragmática y despliegue independiente de unas pocas áreas de dominio	Reutilización de servicios compartidos en toda la empresa	Cambio y despliegue independientes por parte de muchos equipos autónomos
Particionamiento	Por dominio: un servicio por cada área de negocio grande	Técnico: capas de negocio, empresarial, de aplicación y de infraestructura	Por dominio: un servicio por contexto delimitado, o más pequeño
Número típico	Unos 4 a 12 por sistema	De decenas a cientos en toda una empresa	De decenas a cientos por sistema
Granularidad	Grueso: todo un dominio, por ejemplo todo lo relativo a pedidos	Mixto; los servicios empresariales compartidos suelen ser de grano fino	De fino a medio: una capacidad cohesiva
Datos	A menudo una base de datos compartida, idealmente particionada por esquema	Sistemas de registro compartidos detrás de los servicios empresariales	Cada servicio es propietario de sus datos; no hay tablas compartidas
Comunicación	Sobre todo de la interfaz de usuario a un servicio; pocas llamadas entre servicios	A través del bus de servicios empresarial, normalmente SOAP o mensajería XML	HTTP ligero, gRPC o mensajería; extremos inteligentes, tuberías tontas
Dónde vive la lógica	Dentro de los servicios de dominio	Gran parte en el bus: enrutamiento, transformación, orquestación	Dentro de los servicios; la red solo transporta
Transacciones	Transacciones ACID locales dentro de un servicio de dominio	Se intentaban transacciones distribuidas (WS-AtomicTransaction, XA)	Sin transacciones distribuidas; sagas y consistencia eventual
Gobernanza	Ligera; basta con un equipo de arquitectura	Central y pesada: modelo canónico, registro de servicios, comité de estándares	Descentralizada; plataforma compartida y caminos pavimentados

Aspecto	Basada en servicios	SOA clásica (dirigida por la orquestación)	Microservicios
Costo operativo	Moderado	Alto, más licencias caras de middleware	Alto: muchos desplegables, pipelines y turnos de guardia

Características de la arquitectura basada en servicios

Característica	Valoración	Razón
Desplegabilidad	Media a alta	Cada servicio de dominio se despliega por su cuenta, así que un cambio en las cartas no vuelve a desplegar los pagos; un cambio en un esquema compartido sigue necesitando coordinación.
Tolerancia a fallos	Media a alta	Una caída del servicio de reparto deja funcionando los pedidos y los pagos; una base de datos compartida sigue siendo un punto común de fallo.
Escalabilidad	Media	Los servicios escalan por separado, pero cada uno es de grano grueso, y la base de datos compartida limita hasta dónde pueden escalar las escrituras.
Elasticidad	Baja a media	Los servicios grandes arrancan despacio y escalan a saltos grandes, así que los picos repentinos se absorben con menos soltura que con servicios pequeños.
Consistencia de los datos	Alto	La mayoría de los flujos de trabajo de negocio se quedan dentro de un servicio, donde una transacción corriente de base de datos mantiene la consistencia de los datos.
Simplicidad	Media	Muchas menos piezas móviles que los microservicios; sigue siendo un sistema distribuido, con llamadas de red, pipelines de despliegue y monitorización por servicio.
Costo global	Medio (moderado)	Un puñado de servicios y una base de datos son baratos de operar en comparación con decenas de servicios y bases de datos.
Capacidad de prueba (testabilidad)	Media a alta	Cada servicio tiene un alcance menor que el monolito y puede probarse por su cuenta contra su esquema.
Capacidad de evolución	Media	El particionamiento por dominio mantiene los cambios típicos dentro de un servicio, pero la base de datos compartida acopla la evolución del esquema.

Cuándo la arquitectura basada en servicios es el término medio pragmático

Elígela cuando el sistema tenga unos pocos dominios de negocio claros, cuando entre dos y quizá ocho equipos necesiten lanzar de forma independiente, cuando la mayoría de los flujos de trabajo se queden de forma natural dentro de un dominio y cuando la consistencia fuerte dentro de un dominio sea valiosa. También es un paso de migración excelente para salir de un monolito: dividir un monolito bien modularizado en un puñado de servicios de dominio es mucho menos arriesgado que saltar a cincuenta microservicios, y a menudo el resultado es lo bastante bueno como para quedarse ahí.

Evítala, o planifica ir más allá, cuando partes de un dominio necesiten una escalabilidad o una disponibilidad muy distintas (pertenecen a cuantos separados), cuando decenas de equipos necesiten autonomía, cuando la base de datos compartida se haya convertido en el cuello de botella de las escrituras, o cuando los servicios de dominio se llamen constantemente entre sí para completar solicitudes corrientes, lo que es señal de que los límites están mal.

ATENCIÓN

Antipatrones

- **La base de datos compartida en la que todo vale.** Servicios que leen directamente las tablas de los demás. En pocos meses, el esquema ya no puede cambiarse sin un lanzamiento coordinado de todo.
- **Servicios de dominio demasiado conversacionales.** Pedidos llama a Restaurante, que llama a Cliente, que llama a Pedidos. Si los flujos de trabajo corrientes necesitan varios servicios, fusiónalos o traslada el flujo de trabajo.
- **La única biblioteca gigante de entidades.** Una sola biblioteca compartida que contiene las clases de todas las tablas, de modo que cualquier cambio de esquema obliga a todos los servicios a recompilarse y volver a desplegarse.
- **SOA con otro nombre.** Una capa central de integración que poco a poco va acumulando reglas de enrutamiento, transformaciones y lógica de negocio: un ESB reinventado, con el mismo cuello de botella.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué la mayoría de los flujos de trabajo de una arquitectura basada en servicios pueden usar una transacción corriente de base de datos?

Porque cada servicio de dominio cubre toda un área de negocio y el flujo de trabajo suele quedarse dentro de él, tocando solo datos del propio esquema de ese servicio en una única base de datos.

▼ ¿Qué hizo que el bus de servicios empresarial se convirtiera en un cuello de botella en muchos programas de SOA?

La lógica de enrutamiento, de transformación y de procesos vivía en el bus, normalmente propiedad de un único equipo central, así que la mayoría de los cambios de negocio necesitaban a ese equipo y el bus era un punto de fallo compartido.

▼ Nombra dos técnicas que reducen el acoplamiento causado por una base de datos compartida.

Dar a cada servicio su propio esquema y prohibir el acceso a los de los demás (con permisos que lo hagan cumplir), y dividir las clases de entidad en pequeñas bibliotecas por dominio; también dar a un servicio su propia base de datos cuando esté justificado.

▼ ¿Qué invierte "smart endpoints and dumb pipes" (extremos inteligentes y tuberías tontas) respecto a la SOA clásica?

Saca la lógica del middleware (el ESB) y la lleva a los servicios, dejando que la red solo transporte mensajes.

▼ Inténtalo: Una empresa de venta de entradas con 12 ingenieros y un monolito ordenado sufre colas de lanzamiento, y un defecto en los informes dejó fuera de servicio el proceso de pago dos veces. ¿Recomendarías arquitectura basada en servicios o microservicios de grano fino? Da tu razonamiento y un riesgo de tu elección.

Basada en servicios: unos cuatro a seis servicios de dominio de grano grueso, como Reservas, Pagos, Recintos e Informes. La mayoría de los flujos de trabajo se quedan dentro de un servicio, de modo que las transacciones locales sobreviven y las cadenas síncronas siguen siendo cortas. Aísla los fallos de los informes y alivia las colas de lanzamiento sin el costo operativo de decenas de servicios, que 12 ingenieros no podrían asumir. El riesgo es la base de datos compartida, que acopla los cambios de esquema y los mantiene en un solo cuanto. Usa esquemas por servicio y separa los datos de Informes.

FUENTES

- James Lewis and Martin Fowler: Microservices (2014), on smart endpoints and dumb pipes: https://martinfowler.com/articles/microservices.html
- OASIS: Reference Model for Service Oriented Architecture 1.0: https://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf
- W3C: Web Services Architecture (Working Group Note): https://www.w3.org/TR/ws-arch/
- OASIS: Web Services Business Process Execution Language Version 2.0: https://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
- Enterprise Integration Patterns: messaging patterns catalogue: https://www.enterpriseintegrationpatterns.com/patterns/messaging/

Microservicios: principios, beneficios y costos

ANTES DE LEER ESTO

Esta sección se basa en:

- 5.7 Diseño guiado por el dominio
- 6.1 Por qué distribuir
- 6.2 Basada en servicios y SOA

Términos anteriores que utiliza: Contexto delimitado (bounded context), Cuanto de arquitectura (architecture quantum), Consistencia eventual, Bus de servicios empresarial (ESB), Ley de Conway

LA IDEA EN BREVE

Imagina un gran mercado gastronómico en lugar de un único restaurante grande. Cada puesto tiene su propio equipo pequeño, su propia cocina, su propia nevera y su propia carta, y puede cambiar sus recetas o cerrar por reformas sin preguntar a los demás puestos. Los clientes piden en un mostrador común que envía cada pedido al puesto que corresponde. El mercado puede crecer añadiendo puestos, y si se quema la cocina de un puesto, los demás no cierran. Pero alguien tiene que gestionar el mostrador común, las entregas, la limpieza y los controles de seguridad de todos los puestos, y una comida de tres puestos exige una coordinación cuidadosa. Los microservicios son el mercado gastronómico.

LO QUE NECESITAS SABER

- Los microservicios (Lewis y Fowler, 2014) construyen una aplicación como servicios pequeños, cada uno en su propio proceso, construido en torno a una capacidad de negocio, desplegable de forma independiente y que se comunica mediante mecanismos ligeros.
- Los límites entre servicios funcionan mejor cuando se alinean con los contextos delimitados y con los equipos; los límites trazados en torno a capas técnicas o a tablas de la base de datos producen servicios conversadores y fuertemente acoplados.
- La desplegabilidad independiente es la propiedad que los define; depende de contratos compatibles hacia atrás, de pruebas de contrato y del versionado, no solo de tener contenedores separados.
- Una base de datos por servicio convierte cada servicio en su propio cuanto, pero sustituye los joins y las transacciones por llamadas a API, eventos, sagas y consistencia eventual.

- Entre los patrones de apoyo están los API gateways, los backends for frontends, el descubrimiento de servicios y las service meshes; cada uno resuelve un problema real y añade algo más que operar.
- La granularidad se decide sopesando los desintegradores (alcance, volatilidad, escalabilidad, tolerancia a fallos, seguridad, extensibilidad) frente a los integradores (transacciones, flujo de trabajo, código compartido, relaciones entre datos).
- Los fallos típicos son el monolito distribuido, los servicios de entidad, los servicios conversadores y las bibliotecas compartidas que acoplan los despliegues; el estilo necesita automatización, monitorización y equipos que operen lo que construyen.

Definición y origen

Los microservicios son un estilo arquitectónico en el que una aplicación se construye como un conjunto de servicios pequeños, cada uno ejecutándose en su propio proceso, cada uno construido en torno a una capacidad de negocio, cada uno desplegable de forma independiente mediante maquinaria automatizada, que se comunican mediante mecanismos ligeros como API HTTP o mensajería, con un mínimo de gestión centralizada. Ese resumen sigue el artículo que definió el término para la mayor parte del sector: *Microservices*, de James Lewis y Martin Fowler, publicado en martinfowler.com en marzo de 2014. El artículo señala que el nombre se discutió en un taller de arquitectos de software cerca de Venecia en mayo de 2011 y que el grupo lo adoptó en mayo de 2012; empresas como Amazon y Netflix ya trabajaban así.

Lewis y Fowler no dieron una definición formal. Describieron nueve características comunes, y los profesionales todavía las usan como lista de comprobación: componentización mediante servicios; organización en torno a capacidades de negocio; productos en lugar de proyectos; extremos inteligentes y tuberías tontas; gobernanza descentralizada; gestión de datos descentralizada; automatización de la infraestructura; diseño para el fallo; y diseño evolutivo. Los apartados siguientes tratan una a una las más significativas desde el punto de vista arquitectónico.

SI ESTO ES NUEVO PARA TI

Si esto es nuevo para ti

Aquí, un *servicio* es simplemente un programa que se ejecuta por sí solo, normalmente en varios servidores a la vez, y ofrece una API a otros programas (Capítulo 0). En un monolito, el código de las cartas, los pedidos y los pagos se compila en un solo programa y se despliega junto. En los microservicios, cada una de esas partes se convierte en un programa independiente, propiedad de su propio equipo pequeño, con su propia base de datos, y se lanza cuando ese equipo está listo. El resto de esta sección trata de dónde trazar las líneas entre esos programas y de lo que cuesta operar tantos.

Servicios alineados con contextos delimitados

La pregunta más difícil en los microservicios es dónde trazar las líneas. La respuesta más fiable viene del diseño estratégico guiado por el dominio (Capítulo 5): un contexto delimitado es una parte del negocio dentro de la cual un modelo y su lenguaje son coherentes. En el reparto de comida, 'pedido' significa un carrito que se está pagando en Pedidos, una comanda que cocinar en el contexto de la cocina del restaurante y una recogida y una entrega en Despacho. Cada contexto es un candidato natural a un servicio, o a un pequeño número de servicios propiedad de un solo equipo.

Alinear los servicios con los contextos delimitados les da una cohesión alta (lo que cambia junto vive junto) y un acoplamiento bajo (los contextos se comunican a través de unos pocos contratos explícitos, traducidos en los bordes). También los alinea con los equipos, porque un contexto delimitado suele tener el tamaño de algo que un solo equipo puede entender y poseer. Los servicios trazados en torno a capas técnicas o a tablas de la base de datos carecen de ambas propiedades, y los antipatrones que se ven más adelante en esta sección son en su mayoría el resultado.

Desplegabilidad independiente: la propiedad que los define

Si una propiedad define los microservicios, es la desplegabilidad independiente: un equipo puede cambiar su servicio y lanzarlo a producción sin cambiar, probar ni lanzar ningún otro servicio al mismo tiempo. Sam Newman, en *Building Microservices* (segunda edición, 2021), la trata como la propiedad que hay que proteger por encima de todas las demás, porque casi todos los otros beneficios dependen de ella. La entrega más rápida, un radio de impacto menor, el escalado selectivo y la autonomía de los equipos desaparecen si los lanzamientos deben coordinarse.

La desplegabilidad independiente es una propiedad de los contratos, no de los contenedores. Dos servicios que se despliegan por separado pero que siempre deben lanzarse juntos, porque un cambio en uno rompe el otro, no son desplegables de forma independiente. Protegerla exige cambios de interfaz compatibles hacia atrás (añadir campos, nunca eliminarlos ni renombrarlos sin un periodo de obsolescencia), pruebas de contrato dirigidas por el consumidor y API versionadas, todo lo cual se trata en comunicación y API.

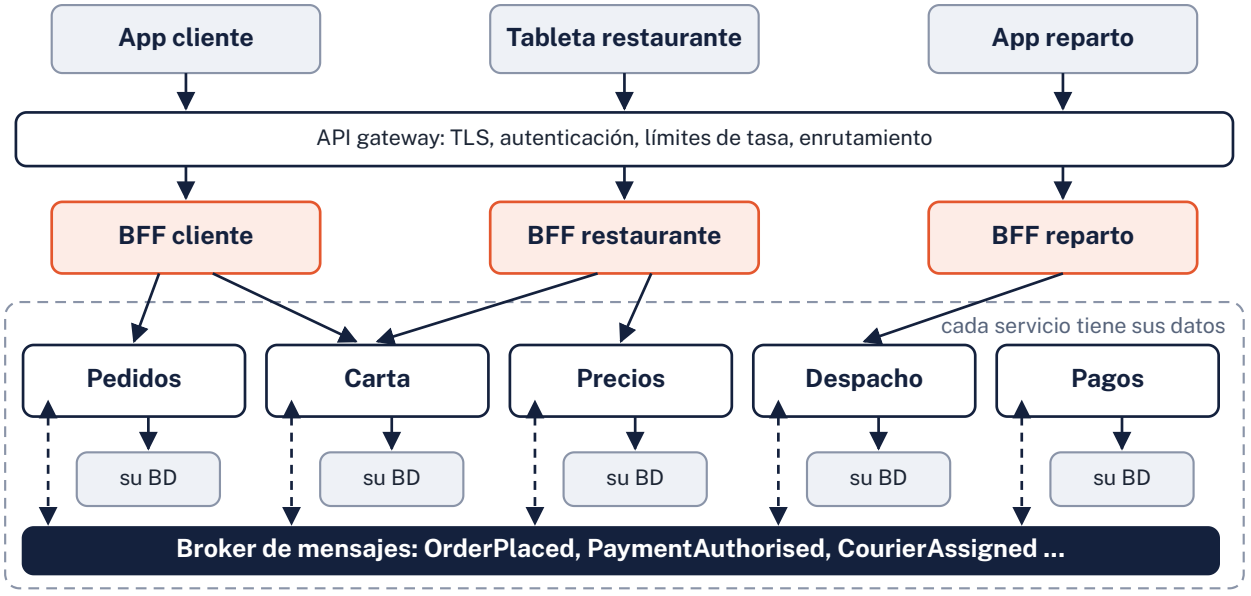
Una base de datos por servicio, y extremos inteligentes con tuberías tontas

La gestión de datos descentralizada de Lewis y Fowler se convierte, en la práctica, en la regla de una base de datos por servicio: cada servicio es propietario de sus datos, y ningún otro servicio los lee ni los escribe directamente. Los demás servicios obtienen los datos a través de la API del propietario o de los eventos que el propietario publica. La regla es lo que convierte un servicio en un cuanto independiente: su esquema puede cambiar, y puede escalarse, respaldarse y restaurarse, sin pedir permiso a nadie. Cada servicio puede además elegir el almacén que le conviene (una base de datos relacional para los pedidos, un almacén clave-valor para las sesiones, un motor de búsqueda para las cartas), lo que se llama persistencia políglota.

El precio es alto y hay que pagarlo con conocimiento de causa. Las consultas que antes eran un join SQL entre tablas ahora atraviesan servicios. Los cambios que antes eran una sola transacción ahora necesitan una saga y pasan a ser eventualmente consistentes. Los informes necesitan un pipeline aparte. El Capítulo 7 trata en profundidad la propiedad de los datos.

La otra característica con mayores consecuencias arquitectónicas son los *extremos inteligentes y tuberías tontas* (smart endpoints and dumb pipes). La lógica de negocio pertenece a los servicios; la red y el intermediario de mensajes (message broker) solo transportan mensajes. Es una reacción directa contra el bus de servicios empresarial de la SOA clásica, donde las reglas de enrutamiento, las transformaciones y los procesos se acumulaban en un middleware central.

Una topología de microservicios para el reparto de comida



Flechas continuas: llamadas síncronas. Discontinuas: eventos publicados en el broker y leídos de él.

Los clientes llegan a los servicios a través de un gateway y un backend for frontend cada uno; cada servicio es propietario de sus datos, y los eventos fluyen a través de un broker.

Patrones de apoyo

En torno al estilo ha surgido un conjunto de patrones de apoyo. Cada uno resuelve un problema real y cada uno añade un componente que hay que operar, así que adóptalos cuando aparezca el problema y no todos a la vez.

Patrones de apoyo habituales

Patrón	Problema que resuelve	Qué es	Atención a
API gateway (puerta de enlace de API)	Los clientes no deberían conocer las direcciones de decenas de servicios ni repetir la autenticación ante cada uno.	Un único punto de entrada que termina TLS, autentica, aplica límites de tasa y enruta las solicitudes a los servicios; puede combinar varias llamadas en una sola respuesta.	Lógica de negocio que se cuela en el gateway y recrea un ESB; un punto único de fallo que necesita alta disponibilidad.
Backend for frontend (BFF)	Una app móvil, una app web y una API para socios necesitan datos con formas distintas; una API general no sirve bien a ninguna.	Un backend independiente y delgado por cada tipo de cliente, propiedad del equipo de ese cliente, que llama a los servicios y da forma a las respuestas. El nombre es de Phil Calçado, de SoundCloud; Sam Newman popularizó el patrón en 2015.	Lógica duplicada entre los BFF; un BFF que se convierte en un minimonolito.
Descubrimiento de servicios	Las instancias de los servicios arrancan, se detienen y se mueven constantemente; las direcciones no pueden configurarse a mano.	Un registro de las instancias sanas en cada momento, que consultan los clientes o un balanceador de carga. Kubernetes lo proporciona mediante nombres DNS para los Services.	Entradas del registro obsoletas; clientes que guardan en caché las direcciones demasiado tiempo.
Service mesh (malla de servicios)	Todos los servicios necesitan TLS mutuo, reintentos, timeouts, desvío de tráfico y telemetría; implementarlos en cada lenguaje es un desperdicio.	Una capa de infraestructura de proxies (tradicionalmente un sidecar junto a cada instancia del servicio) controlada de forma centralizada, como Istio o Linkerd.	Una complejidad operativa considerable; los reintentos configurados a la vez en la malla y en el código multiplican la carga.

Los requisitos previos de Fowler y los costos organizativos

En agosto de 2014, Martin Fowler enumeró tres cosas que una organización debería tener antes siquiera de construir microservicios. **Aprovisionamiento rápido**: la capacidad de crear un servidor o un entorno nuevo en horas, lo que implica automatización. **Monitorización básica**: tanto medidas técnicas (errores, latencia, disponibilidad) como medidas de negocio (pedidos por minuto) para cada servicio, de modo que los problemas se detecten rápido. **Despliegue rápido de aplicaciones**: un pipeline automatizado que lleva un servicio a pruebas y a producción rápidamente, con un rollback sencillo. Sin esto, muchos servicios pequeños

multiplican el trabajo manual. Su idea complementaria, la *prima de los microservicios* (microservice premium, 2015), es que el estilo conlleva un costo fijo en complejidad y productividad que solo compensa en sistemas lo bastante complejos como para necesitarlo.

Los costos organizativos son tan reales como los técnicos. El estilo presupone equipos estables que son propietarios de sus servicios en producción, el principio que Werner Vogels, CTO de Amazon, describió en 2006 como “You build it, you run it” (quien lo construye, lo opera). Eso significa que cada equipo necesita habilidades operativas y un turno de guardias. Necesita una plataforma: pipelines de despliegue, observabilidad, gestión de secretos y plantillas de servicio, normalmente proporcionada por un equipo de plataforma (Capítulo 10). Necesita acuerdos sobre estándares transversales (formato de logging, identificadores de correlación, convenciones de API) sin aprobación central de cada cambio. Y necesita que la ley de Conway se use deliberadamente: los límites de los equipos y los de los servicios deben coincidir; de lo contrario, cada cambio exige que se coordinen varios equipos.

El lado del cliente: renderizado, micro-frontends, dispositivos móviles, modo sin conexión y el BFF

Los clientes se encuentran con un sistema a través de sus aplicaciones cliente, y estas tienen su propia arquitectura. La primera decisión es dónde se renderiza una página. En una **aplicación de una sola página** (SPA), el navegador construye cada pantalla con JavaScript, lo que encaja con sesiones largas e interactivas, como la pantalla de pedidos de un restaurante, pero el código tiende a crecer con la aplicación y puede ralentizar la respuesta a los toques. Con el **renderizado del lado del servidor** (SSR), el servidor envía HTML terminado, de modo que el contenido aparece rápido y se ejecuta menos código en el dispositivo, a costa de trabajo del servidor en cada solicitud. El **renderizado estático** produce HTML en tiempo de compilación para una CDN, lo que encaja con las páginas de ayuda, pero no con las personalizadas. Muchos frameworks renderizan en el servidor y luego *hidratan*, es decir, añaden el comportamiento en el navegador; web.dev de Google advierte de que una página así puede parecer lista antes de responder. La elección puede variar de una página a otra.

Los **micro-frontends** llevan el despliegue independiente a la interfaz: "aplicaciones frontend que se pueden entregar de forma independiente se componen en un todo mayor" (Cam Jackson, 2019). Los equipos de la carta y del proceso de pago podrían lanzar cada uno su parte del sitio web, compuesta en tiempo de ejecución por una herramienta como Module Federation de webpack. Los costos son bibliotecas duplicadas en cada descarga, o versiones compartidas que devuelven el acoplamiento, además de más pipelines y más esfuerzo para mantener una apariencia coherente. El estilo responde al problema de muchos equipos en un mismo frontend; un equipo pequeño gana poco.

Los dispositivos móviles cambian las reglas de los lanzamientos. Una actualización llega a los usuarios de forma gradual, mediante el lanzamiento por fases de siete días de Apple o el lanzamiento escalonado de Google Play, y detener un lanzamiento en Play deja la actualización a quienes ya la tienen. Las versiones antiguas siguen en uso, así que las API deben seguir siendo compatibles durante mucho tiempo (consulta la Sección 6.5).

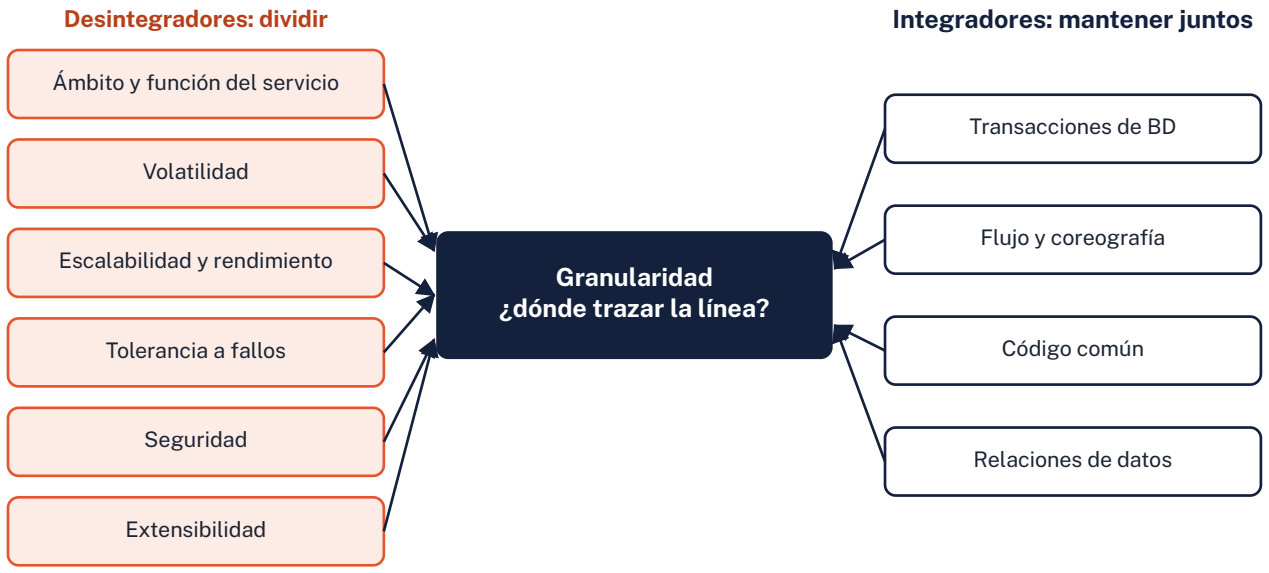
Un cliente **offline-first** (primero sin conexión) trata un almacén local como su fuente de verdad: las pantallas leen datos locales y un proceso de sincronización los concilia con el servidor, que conserva la autoridad final. La guía de Android nombra tres estrategias de escritura: solo en línea, adecuada para los pagos; en cola; y diferida, que escribe primero en local y debe resolver los conflictos. Un repartidor en un estacionamiento subterráneo puede marcar un pedido como entregado y dejar que la app lo envíe más tarde, con una clave de idempotencia. En los navegadores, los service workers pueden servir páginas en caché, pero la Background Sync API no funciona en todos los navegadores principales, así que comprueba la compatibilidad actual.

El **backend for frontend** lo une todo. La pauta de Sam Newman es una experiencia, un BFF, mantenido por el equipo que construye esa interfaz, de modo que el BFF de los repartidores puede devolver cargas útiles pequeñas para redes deficientes, mientras que el BFF web da forma a los datos página por página. También puede mantener los tokens fuera del navegador (consulta la Sección 8.3). Las reglas de negocio permanecen en los servicios: Newman sugiere trasladar la lógica compartida por los BFF a un servicio de dominio. Su advertencia sobre la alternativa, una API general para todos los clientes, es que se convierte en un cuello de botella.

Granularidad: ¿qué tamaño debe tener un servicio?

No hay un tamaño correcto en líneas de código ni en personas. Lewis y Fowler encontraron equipos de alrededor de una docena de personas propietarios de un servicio y equipos de unas seis personas propietarios de seis servicios, y lamentaron que la palabra 'micro' centre la atención en el tamaño. Ford, Richards, Sadalage y Dehghani ofrecen una herramienta mejor en *Software Architecture: The Hard Parts* (2021): sopesar las fuerzas que empujan a dividir un servicio, que llaman **desintegradores de granularidad**, frente a las que empujan a mantenerlo unido, los **integradores**.

Fuerzas que actúan sobre la granularidad de los servicios



Fuerzas de Ford, Richards, Sadalage y Dehghani, Software Architecture: The Hard Parts (2021)

Seis desintegradores abogan por servicios más pequeños; cuatro integradores, por servicios más grandes. La granularidad es un equilibrio que se decide caso por caso.

Desintegradores e integradores de granularidad, con ejemplos del reparto de comida

Fuerza	Dirección	Pregunta	Ejemplo
Ámbito y función del servicio	Dividir	¿El servicio hace una sola cosa cohesionada o varias cosas sin relación?	Un servicio de Notificaciones que además calcula los puntos de fidelidad está haciendo dos trabajos.
Volatilidad	Dividir	¿Hay una parte que cambia mucho más a menudo que el resto?	Las reglas de Promociones cambian cada semana; el resto de Precios cambia cada trimestre.
Escalabilidad y rendimiento	Dividir	¿Las partes necesitan capacidades muy diferentes?	Las actualizaciones de ubicación de los repartidores llegan miles de veces por segundo; las altas de repartidores, unas pocas veces al día.
Tolerancia a fallos	Dividir	¿Un fallo en una parte haría caer otra sin necesidad?	Una caída del generador de recibos en PDF no debería impedir que se hagan pedidos.
Seguridad	Dividir	¿Las partes necesitan controles de acceso o auditorías diferentes?	Tokens de tarjeta almacenados frente a datos públicos de la carta.
Extensibilidad	Dividir	¿Se añadirán variantes nuevas con frecuencia?	Nuevos métodos de pago cada pocos meses, cada uno como una unidad desplegable independiente.
Transacciones de BD	Mantener juntos	¿Estos cambios deben tener éxito o fallar de forma atómica?	Crear un pedido y reservar su franja de entrega.
Flujo y coreografía	Mantener juntos	¿Dividir crearía largas cadenas de llamadas para las solicitudes corrientes?	El carrito, el proceso de pago y el estado del pedido se llaman entre sí en cada solicitud.
Código común	Mantener juntos	¿Las partes comparten una lógica de negocio importante que cambia a la vez?	Carta y Precios comparten reglas complejas sobre las opciones de los platos.
Relaciones de datos	Mantener juntos	¿Los datos están estrechamente relacionados, con joins y claves foráneas?	Los pedidos y sus líneas; separarlos no aporta nada.

EJEMPLO PRÁCTICO

Ejemplo práctico: ¿conviene dividir Precios?

La plataforma de reparto de comida tiene un servicio de Precios que calcula los totales del carrito: precios de los platos, recargos por opciones, tarifas de envío (que dependen de la distancia y de la demanda) y promociones (cupones, descuentos en el primer pedido, ofertas de restaurantes). El equipo de promociones lanza cambios varias veces por semana, y cada lanzamiento de Precios necesita una ejecución completa de regresión del cálculo de tarifas. Alguien propone dividir Precios en los servicios Tarifas y Promociones. El arquitecto sopesa las fuerzas.

Volatilidad del código (dividir)	Fuerte: las promociones cambian cada semana; las reglas de tarifas, cada trimestre. Separarlas permitiría lanzar las promociones sin volver a probar las tarifas.
Escalabilidad (dividir)	Débil: ambas se llaman una vez por cada visualización del carrito, al mismo ritmo.
Tolerancia a fallos (dividir)	Moderada: una regla de promoción defectuosa ha hecho caer Precios dos veces. Si Promociones fuera independiente, un fallo podría degradarse a 'no se muestra el descuento' mientras las tarifas siguen funcionando.
Seguridad (dividir)	Débil: la misma sensibilidad de los datos.
Transacciones de base de datos (mantener)	Débil: ninguna escritura necesita abarcar ambas; un cupón se canjea cuando se hace el pedido, en Pedidos.
Flujo de trabajo (mantener)	Moderada: cada total del carrito necesita ambas. Tras la división, Precios (o un BFF) debe llamar a Promociones de forma síncrona en cada visualización del carrito, lo que añade un salto y sus modos de fallo.
Código compartido (mantener)	Moderada: ambas necesitan el modelo del carrito y las reglas de redondeo de divisas, que cambian rara vez.
Relaciones entre datos (mantener)	Débil: las promociones se refieren a los restaurantes y los platos solo por su identificador.
Decisión	Dividir, con cuidado: extraer Promociones; Precios lo llama con un timeout de 150 ms y, si falla, calcula el precio del carrito sin descuentos y lo marca para recalcularlo en el proceso de pago. Compartir solo una biblioteca pequeña y versionada del modelo del carrito.

La decisión se apoya en dos desintegradores fuertes (la volatilidad y el aislamiento de fallos) que pesan más que unos integradores moderados. Si las promociones hubieran tenido que actualizar las mismas filas que el cálculo de tarifas en una sola transacción, la respuesta probablemente habría sido otra.

Poner las fuerzas por escrito convierte una cuestión de gusto en un argumento que otros pueden comprobar, y ese registro corresponde a un registro de decisiones de arquitectura.

Antipatrones

La mayoría de los programas de microservicios que fracasan lo hacen de formas reconocibles. La más grave es el monolito distribuido: un sistema con todos los costos de la distribución y nada de la independencia, porque sus servicios comparten una base de datos, se llaman entre sí en largas cadenas síncronas o dependen de las mismas bibliotecas desplegables.

Antipatrones de los microservicios

Antipatrón	Síntoma	Causa y remedio
Monolito distribuido	Los servicios deben lanzarse juntos en un orden fijo; la caída de un servicio los detiene a todos.	Base de datos compartida, cadenas síncronas y bibliotecas desplegables compartidas. Remedio: volver a trazar los límites en torno a contextos delimitados, dar a cada servicio sus datos, pasar las llamadas a eventos o volver a fusionarlos en un monolito modular.
Servicios de entidad	Servicios con nombres de sustantivos (Cliente, Pedido, Restaurante) con operaciones de creación, lectura, actualización y borrado; cada proceso de negocio llama a varios de ellos.	Límites trazados en torno a los datos en lugar del comportamiento. Michael Nygard lo llama el antipatrón del servicio de entidad. Remedio: organizar en torno a las capacidades y los procesos de negocio.
Servicios conversadores	Una sola solicitud de un usuario provoca decenas de llamadas entre servicios; la latencia y las tasas de fallo aumentan con cada funcionalidad.	Los límites atraviesan un flujo de trabajo cohesionado, o las interfaces son demasiado finas. Remedio: API más gruesas, copias locales de los datos de referencia, fusionar los servicios que siempre cambian juntos.
Bibliotecas compartidas que acoplan los despliegues	Un cambio en una biblioteca común obliga a todos los servicios a recompilarse y volver a desplegarse; las versiones se van separando.	Lógica de negocio o modelos de dominio situados en código compartido. Remedio: compartir solo bibliotecas técnicas y estables; duplicar pequeñas cantidades de código de negocio; versionar todo lo que se comparta y permitir que haya varias versiones en uso.
Envidia de microservicios	Un equipo pequeño con un producto nuevo empieza con treinta servicios y dedica la mayor parte de su tiempo a la infraestructura.	Elegir el estilo por moda o por su valor en el currículum en lugar de por los impulsores. Remedio: empezar con un monolito modular y extraer servicios cuando aparezca un impulsor.
Nanoservicios	Servicios tan pequeños que la mayor parte de su código es fontanería; una funcionalidad necesita cambios en cinco de ellos.	Aplicar 'pequeño' en exceso. Remedio: usar los integradores de granularidad; dimensionar los servicios por su cohesión, no por las líneas de código.

Características de los microservicios

Característica	Valoración	Razón
Desplegabilidad	Alto	Cada servicio se lanza por sí solo, muchas veces al día, con un radio de impacto pequeño.
Escalabilidad y elasticidad	Alto	Cada servicio escala de forma independiente; los servicios pequeños arrancan rápido y escalan en pasos finos.
Tolerancia a fallos	Alto (si se diseña para ello)	Los fallos pueden aislarse por servicio, pero solo con timeouts, bulkheads, alternativas de respaldo (fallbacks) y comunicación asíncrona; de lo contrario, los defectos se propagan en cascada.
Capacidad de evolución	Alto	Los servicios pequeños y bien delimitados pueden reescribirse o sustituirse uno a uno.
Capacidad de prueba de un servicio	Alto	Cada servicio es pequeño; las pruebas de contrato comprueban sus interacciones.
Probar el conjunto	Bajo	El comportamiento de extremo a extremo surge de muchos servicios y es difícil de reproducir en un entorno de pruebas.
Rendimiento	Baja a media	Los saltos de red, la serialización y los controles de seguridad añaden latencia a cada interacción.
Consistencia de los datos	Bajo	No hay transacciones compartidas; las invariantes de negocio que abarcan varios servicios dependen de la consistencia eventual y de las sagas.
Simplicidad	Bajo	Muchas piezas móviles, muchos modos de fallo, mucha infraestructura.
Costo global	Bajo (costoso)	Plataforma, herramientas, observabilidad y personas para operar decenas de servicios.

Cuándo encajan los microservicios y cuándo no

Los microservicios encajan en organizaciones con muchos equipos que trabajan en un mismo producto y necesitan lanzar de forma independiente, en sistemas cuyas partes tienen necesidades realmente distintas de escalabilidad, disponibilidad o seguridad, y en dominios que se entienden lo bastante bien como para trazar límites estables. Necesitan inversión en una plataforma y en habilidades operativas antes de separar el primer servicio.

No encajan en un equipo pequeño, en un producto nuevo cuyo dominio todavía se está descubriendo (los límites trazados demasiado pronto son costosos de mover a través de la red), en un sistema cuyos flujos de trabajo principales necesitan una consistencia transaccional

fuerte en la mayor parte de sus datos, ni en una organización sin despliegue y monitorización automatizados. El consejo *MonolithFirst* de Fowler (Capítulo 5) y las marchas atrás que algunas empresas han documentado, como la consolidación que hizo Segment en 2018 de más de 140 servicios de destino de nuevo en uno solo, muestran que el estilo es una herramienta con un costo, no un destino.

> **NOTA**
>
> ### Los profesionales discrepan, y eso es útil
>
> Algunos arquitectos con experiencia sostienen que la mayoría de las organizaciones nunca deberían ir más allá de un monolito modular o de una arquitectura basada en servicios; otros sostienen que los servicios pequeños, con una buena plataforma, son la forma más productiva de trabajar para muchos equipos. Ambos bandos coinciden en las condiciones: límites de dominio claros, desplegabilidad independiente, propiedad de los datos y automatización. Donde faltan esas condiciones, el número de servicios no es el problema que hay que resolver.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ Dos servicios se despliegan en contenedores separados, pero siempre deben lanzarse juntos. ¿Son desplegables de forma independiente?

No. La desplegabilidad independiente consiste en poder cambiar y lanzar uno sin el otro; un contrato acoplado o una base de datos compartida los convierte en una sola unidad a efectos de lanzamiento.

▼ ¿Qué diferencia hay entre un API gateway y un backend for frontend?

Un gateway es un punto de entrada general que se encarga de inquietudes transversales como TLS, la autenticación, la limitación de tasa y el enrutamiento; un BFF es un backend específico de un cliente que da forma a los datos para un tipo de frontend y es propiedad del equipo de ese frontend.

▼ ¿Qué integrador de granularidad argumenta con más fuerza en contra de dividir dos partes que deben actualizar datos de forma atómica?

Las transacciones de base de datos: dividirlas sustituiría una transacción ACID por un flujo de trabajo distribuido, como una saga.

▼ **¿Por qué es una señal de alarma tener servicios llamados Cliente, Pedido y Restaurante con solo operaciones de creación, lectura, actualización y borrado?**

Son servicios de entidad: límites en torno a los datos en lugar del comportamiento, así que cada proceso de negocio tiene que llamar a varios servicios, lo que crea interacciones conversadoras y fuertemente acopladas.

▼ **Inténtalo: Un equipo propone CustomerService, OrderService y OrderLineService, cada uno envolviendo una tabla. Hacer un pedido exige escribir juntos un pedido y cinco líneas. Usando desintegradores e integradores, ¿es OrderLineService una buena división? ¿Qué harías?**

No. Es un servicio de entidad en torno a una tabla, no una capacidad de negocio. Los pedidos y las líneas comparten una transacción, un flujo de trabajo y relaciones de datos, todos ellos integradores fuertes, y no se aplica ningún desintegrador como un escalado, una volatilidad o una seguridad diferentes. Dividido, sería muy conversador, con una saga necesaria para lo que ahora hace una transacción local. Une las líneas en un servicio de Pedidos alineado con su contexto delimitado. Divide solo cuando aparezcan fuerzas reales.

FUENTES

- James Lewis and Martin Fowler: Microservices (2014): https://martinfowler.com/articles/microservices.html
- Martin Fowler: Microservice Prerequisites (2014): https://martinfowler.com/bliki/MicroservicePrerequisites.html
- Sam Newman: Pattern: Backends For Frontends: https://samnewman.io/patterns/architectural/bff/
- Michael Nygard: The Entity Service Antipattern: https://michaelnygard.com/blog/2017/12/the-entity-service-antipattern/
- Thoughtworks: Software Architecture: The Hard Parts, chapter 7 on service granularity (sample PDF): https://www.thoughtworks.com/content/dam/thoughtworks/documents/books/bk_software_architecture_hard_parts_ch7_en.pdf
- Cam Jackson: Micro Frontends (martinfowler.com, 2019): https://martinfowler.com/articles/micro-frontends.html
- web.dev: Rendering on the Web: https://web.dev/articles/rendering-on-the-web
- webpack: Module Federation: https://webpack.js.org/concepts/module-federation/
- Apple: Release a version update in phases: https://developer.apple.com/help/app-store-connect/update-your-app/release-a-version-update-in-phases/

- Google Play Console Help: staged rollouts: https://support.google.com/googleplay/android-developer/answer/6346149
- Android Developers: Build an offline-first app: https://developer.android.com/topic/architecture/data-layer/offline-first
- MDN: Offline and background operation: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Offline_and_background_operation
- MDN: Background Synchronization API: https://developer.mozilla.org/en-US/docs/Web/API/Background_Synchronization_API

SECCIÓN 6.4 DE 6.8 · AVANZADO · 28 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Arquitectura dirigida por eventos: brokers, mediadores y flujos de eventos

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.5 Almacenar datos
- 0.6 API e integración
- 6.1 Por qué distribuir

Términos anteriores que utiliza: Mensajería asíncrona, Idempotente, Transacción, Contexto delimitado (bounded context), Acoplamiento

LA IDEA EN BREVE

Piensa en la cocina de un restaurante con mucho trabajo. Cuando un camarero cuelga una comanda en el riel, a nadie se le dice qué hacer: el cocinero de la parrilla ve que hay un filete que cocinar, el puesto de ensaladas ve una ensalada y el pastelero ve un postre para más tarde. Cada puesto reacciona a la comanda a su propio ritmo. Si el pastelero está en su descanso, la comanda espera en el riel hasta que vuelva. Añadir un puesto nuevo no cambia la forma en que los camareros escriben las comandas. Un sistema dirigido por eventos funciona igual: unas partes anuncian lo que ha ocurrido y otras reaccionan.

LO QUE NECESITAS SABER

- Un evento es un hecho que ha ocurrido (en tiempo pasado, con muchos receptores posibles); un comando es una solicitud a un único manejador que puede rechazarse; un mensaje es el sobre que transporta cualquiera de los dos.
- La topología de broker reparte un flujo de trabajo entre reacciones independientes (coreografía); la topología de mediador pone la secuencia en un único coordinador (orquestación). La coreografía es la que menos acopla; la orquestación hace visibles los flujos de trabajo y facilita el manejo de los fallos.
- Fowler distingue la notificación de eventos, la transferencia de estado transportada por eventos, el event sourcing y CQRS; las dos primeras son formas de comunicarse entre servicios; las dos últimas, formas de que un servicio almacene sus datos.
- En Kafka, un topic se divide en particiones; los eventos con la misma clave van a la misma partición, el orden solo está garantizado dentro de una partición, y cada grupo de consumidores reparte las particiones entre sus miembros y lleva el control de sus propios

offsets.

- La entrega puede ser como máximo una vez (at-most-once), al menos una vez (at-least-once) o efectivamente una vez (effectively-once); la entrega exactamente una vez no es alcanzable en general, así que da por hecho que es al menos una vez y haz que cada consumidor sea idempotente.
- Los eventos publicados son contratos de larga duración: dales identificadores, versiones y marcas de tiempo, hazlos evolucionar de forma compatible y verifícalos con un registro de esquemas.
- Reintenta los fallos transitorios con backoff, mueve los mensajes envenenados a una cola de mensajes fallidos monitorizada y no dejes nunca que un mensaje defectuoso bloquee una partición.

Definición: reaccionar a lo que ocurre

En la arquitectura dirigida por eventos (EDA), los componentes se comunican produciendo eventos y reaccionando a ellos: registros de hechos que han ocurrido en el negocio, como *se hizo un pedido* o *se asignó un repartidor*. El componente que publica un evento no llama a los componentes que reaccionan a él y normalmente ni siquiera sabe que existen. Un intermediario de mensajes (message broker) lleva los eventos de los productores a los consumidores, los retiene mientras los consumidores están ocupados o caídos, y permite añadir consumidores nuevos sin cambiar el productor.

Los diseños dirigidos por eventos no son nuevos: las interfaces gráficas de usuario, los sistemas operativos y los sistemas de negociación bursátil llevan décadas funcionando así, y los productos de mensajería empresarial son anteriores a la web. Lo que cambió en la década de 2010 fue la escala y la durabilidad. Los brokers basados en logs, como Apache Kafka, creado en LinkedIn y publicado como código abierto en 2011, hicieron viable conservar cada evento de negocio durante días o años y volver a reproducirlos. La EDA puede ser el estilo principal de un sistema o, con más frecuencia, la forma en que se comunican entre sí los servicios de un sistema de microservicios o basado en servicios.

SI ESTO ES NUEVO PARA TI

Si esto es nuevo para ti

Un *intermediario de mensajes* (message broker) es un programa que se sitúa entre otros programas y les pasa mensajes, un poco como una oficina de correos. El emisor entrega un mensaje al broker y sigue con su trabajo; el broker lo guarda de forma segura hasta que cada receptor está listo para recogerlo. Como el emisor no espera, no importa si un receptor está ocupado o desconectado un momento. El Capítulo 0 presentó esta idea como mensajería asíncrona a través de una cola; esta sección muestra cómo se construyen arquitecturas enteras en torno a ella.

Eventos, comandos y mensajes

Hay tres palabras que a menudo se usan sin precisión, y la diferencia importa. Un mensaje es el sobre: cualquier dato enviado de un programa a otro a través de un canal. Lo que transporta el mensaje puede ser un evento (un hecho, nombrado en tiempo pasado), un comando (una solicitud dirigida a un único manejador, nombrada como una instrucción) o una consulta y su respuesta. *Enterprise Integration Patterns* (2003), de Gregor Hohpe y Bobby Woolf, catalogó estos tipos de mensaje y los canales que los transportan, y su vocabulario sigue siendo el estándar.

Evento, comando y mensaje

	Evento	Comando	Mensaje (el sobre)
Significado	Un hecho: algo que ha ocurrido	Una solicitud: por favor, haz algo	Cualquier dato enviado de un programa a otro a través de un canal
Nombres	En pasado: `OrderPlaced`, `PaymentAuthorised`	En imperativo: `AuthorisePayment`, `AssignCourier`	Transporta un evento, un comando, una consulta o una respuesta
Receptores	Cero, uno o muchos; el publicador no los conoce	Exactamente un manejador previsto	Depende del canal: una cola (uno) o un topic (muchos)
¿Puede rechazarse?	No; ya ha ocurrido. Los receptores deciden cómo reaccionar	Sí; el manejador puede rechazarlo o fallar al procesarlo	No procede
Acoplamiento	El publicador no depende de los consumidores	El emisor depende de la capacidad y del contrato del receptor	Depende de lo que transporte

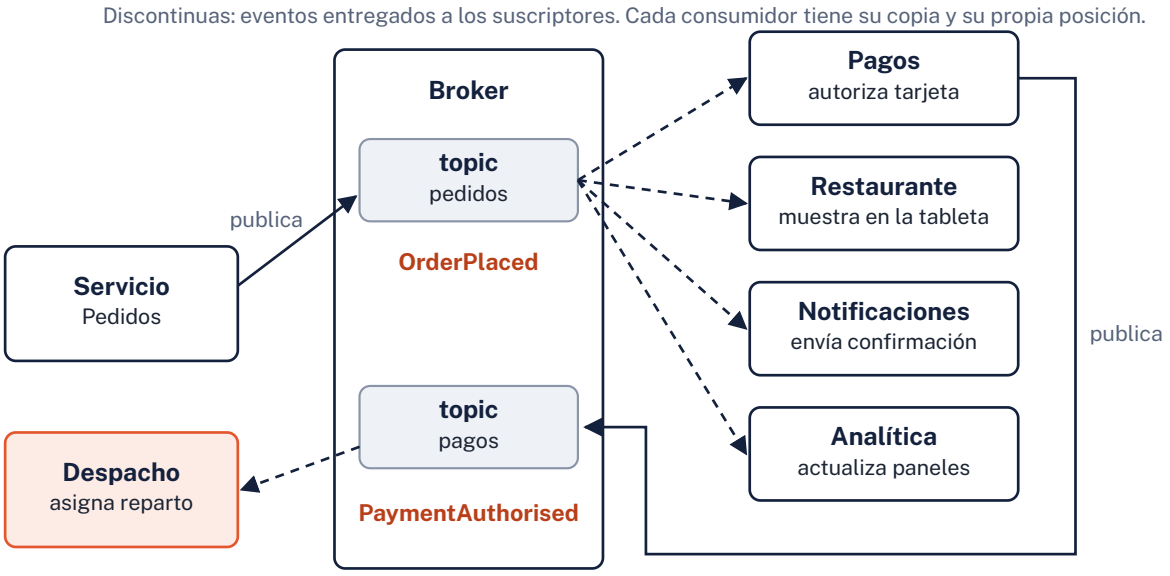
Los mensajes viajan por dos tipos de canal. Una cola entrega cada mensaje a uno de sus consumidores, lo que encaja con los comandos y el reparto de trabajo. Un topic entrega una copia de cada mensaje a cada suscriptor, lo que es publicación-suscripción (pub-sub) y encaja con los eventos. Un diseño que envía `OrderPlaced` a un topic permite que Pagos, Restaurante, Notificaciones y Analítica reaccionen, cada uno a su propio ritmo; añadir el año que viene un servicio de detección de fraude supone una suscripción nueva y ningún cambio en Pedidos.

Topología de broker y topología de mediador

Richards y Ford describen dos topologías para los sistemas dirigidos por eventos. En la topología de intermediario (broker) no hay un coordinador central. Se publica un evento inicial; cada procesador interesado reacciona y puede publicar más eventos, a los que otros reaccionan a su vez. El flujo de trabajo solo existe como la suma de esas reacciones. Esto es coreografía: como bailarines que conocen cada uno sus propios pasos, nadie dirige el conjunto.

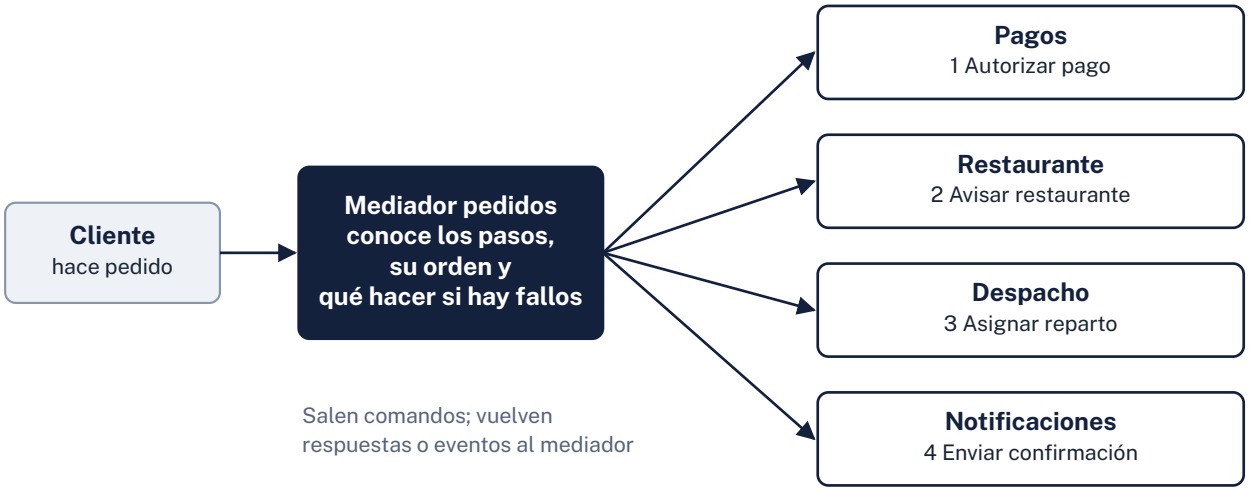
En la topología de mediador, un mediador de eventos recibe el evento inicial y dirige el flujo de trabajo, enviando comandos a los procesadores en un orden definido y actuando según sus respuestas. Esto es orquestación: un director de orquesta conoce la partitura. Los mediadores van desde un servicio sencillo con una máquina de estados hasta un motor de flujos de trabajo como Temporal, AWS Step Functions o un motor BPMN.

Topología de broker: el flujo de OrderPlaced



En la topología de broker, Pedidos solo anuncia el hecho; cada consumidor reacciona de forma independiente, y el propio evento de Pagos impulsa el paso siguiente.

Topología de mediador: el mismo flujo, orquestado



En la topología de mediador, un componente es dueño de la secuencia, así que el flujo de trabajo es visible y los fallos se gestionan en un solo lugar.

¿Coreografía u orquestación?

Pregunta	Coreografía (broker)	Orquestación (mediador)
Acoplamiento	El más bajo: los publicadores no conocen a los consumidores	El mediador depende del contrato de cada participante
Ver el flujo de trabajo completo	Difícil: está repartido entre servicios; hace falta trazado para reconstruirlo	Fácil: está escrito en un solo lugar
Manejo de errores	Cada participante debe detectar los fallos y reaccionar a ellos, a menudo publicando eventos compensatorios	El mediador decide de forma centralizada los reintentos, los timeouts y las compensaciones
Añadir un paso	Añadir un suscriptor; no cambia nada más	Cambiar el mediador
Riesgos	Cadenas cíclicas de eventos; nadie se da cuenta cuando un paso deja de funcionar en silencio	El mediador se convierte en un cuello de botella o acumula lógica de negocio
Encaja bien con	Flujos sencillos y estables; muchas reacciones independientes a un mismo hecho	Flujos de trabajo largos y condicionales con compensación, como la saga del pedido

La mayoría de los sistemas reales combinan ambas: coreografía entre contextos delimitados (Pedidos anuncia hechos y no le importa quién escucha) y orquestación dentro de un contexto cuando un proceso de varios pasos necesita un propietario claro.

Cuatro formas de usar los eventos

En un artículo de 2017, Martin Fowler señaló que 'dirigido por eventos' se usa para al menos cuatro patrones bastante diferentes, y que las conversaciones se tuercen cuando cada persona se refiere a uno distinto. Distinguirlos es uno de los hábitos más útiles que puede tener un arquitecto.

Los cuatro patrones de eventos de Fowler (2017)

Patrón	Qué transporta el evento y por qué	Beneficio	Costo
Notificación de eventos	Una pequeña señal de que algo ha cambiado, a menudo solo un identificador: `OrderPlaced {order_id}`. Los receptores vuelven a llamar para obtener los detalles si los necesitan.	Acoplamiento muy bajo; el modelo del publicador sigue siendo privado.	Las llamadas de vuelta añaden carga y una dependencia en tiempo de ejecución del publicador; el flujo a lo largo de muchos eventos es difícil de ver.
Transferencia de estado transportada por eventos	Datos suficientes para que los receptores mantengan su propia copia: el pedido completo con sus artículos y totales.	Los receptores funcionan sin llamar al publicador, incluso cuando está caído; menor latencia.	Los datos se copian ampliamente y son eventualmente consistentes; eventos más grandes; los esquemas se convierten en contratos.
Event sourcing	Cada cambio en el estado de una entidad se almacena como un evento; el log de eventos es la fuente de verdad y el estado actual se obtiene reproduciéndolo.	Pista de auditoría completa; reconstruir estados pasados; vistas nuevas a partir de la historia antigua.	Complejo; la reproducción es difícil cuando los resultados dependían de sistemas externos; los esquemas de eventos deben evolucionar para siempre.
CQRS	Modelos separados (a menudo almacenes separados) para escribir y para leer, con frecuencia alimentados por eventos.	Cada lado está optimizado para su tarea; encaja con sistemas con muchas lecturas y consultas complejas.	Dos modelos que mantener sincronizados; a menudo se aplica donde bastaría un solo modelo.

La notificación de eventos y la transferencia de estado transportada por eventos tratan de la comunicación entre servicios y pertenecen a este capítulo. El event sourcing y CQRS tratan de cómo almacena sus datos un servicio; a menudo se combinan con la EDA, pero son decisiones independientes, que se tratan en el Capítulo 7.

Flujos de eventos y logs: cómo funciona Kafka

Las colas de mensajes tradicionales eliminan un mensaje en cuanto un consumidor confirma su recepción. Un broker basado en logs, como Apache Kafka, en cambio, almacena los eventos en un log de solo anexado y los conserva durante un periodo de retención configurado (o para

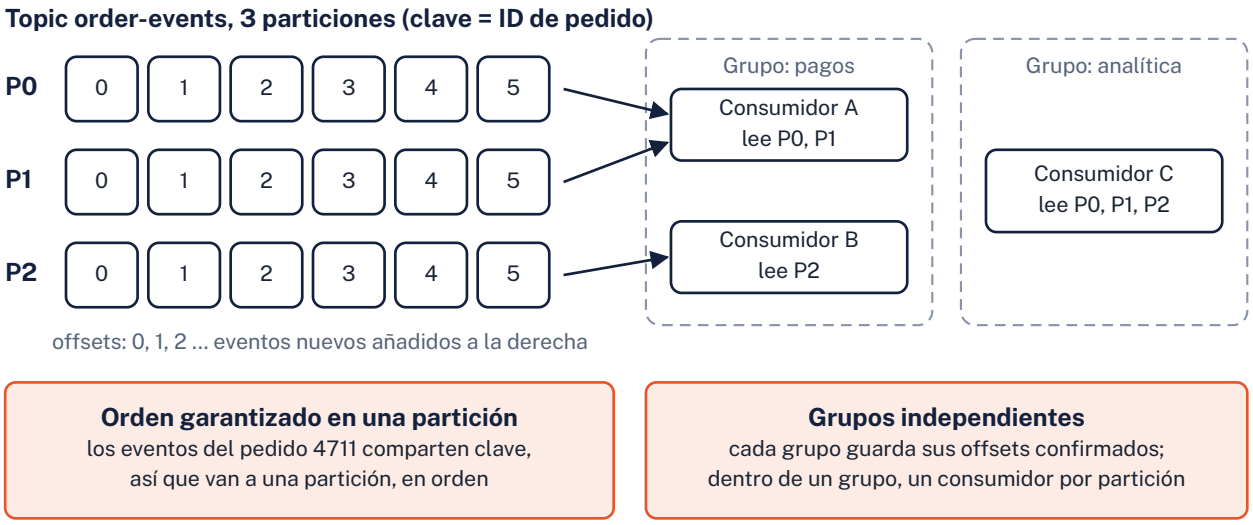
siempre), se hayan leído o no. De ese diseño se derivan varias cosas, que explican tanto los puntos fuertes de Kafka como sus trampas.

Un topic se divide en particiones, cada una de ellas un log ordenado que se almacena y se replica entre brokers. El productor elige la partición, normalmente aplicando un hash a la clave del evento, de modo que todos los eventos con la misma clave (por ejemplo, el mismo ID de pedido) van a la misma partición. Cada evento de una partición tiene una posición secuencial llamada su offset (desplazamiento). Los consumidores extraen los eventos y registran el offset al que han llegado; también pueden rebobinar y volver a reproducirlos.

Los consumidores se unen a un grupo de consumidores. Dentro de un grupo, cada partición la lee exactamente un consumidor a la vez, así que el grupo se reparte el trabajo; los distintos grupos leen de forma independiente, cada uno con sus propios offsets, y así se consigue el pub-sub. Por tanto, el número de particiones limita el paralelismo de un grupo: un topic con 12 particiones puede mantener ocupados como máximo a 12 consumidores de un grupo. (Los *share groups* de Kafka, en vista previa en versiones 4.x anteriores y listos para producción desde Kafka 4.2 en 2026, son una alternativa similar a una cola: varios consumidores pueden compartir una partición y confirmar los mensajes uno a uno, pero entonces no se conserva el orden dentro de la partición. Esta sección describe los grupos de consumidores clásicos).

Kafka garantiza el orden solo dentro de una partición. Los eventos de un pedido llegan en el orden en que se escribieron si comparten clave; los eventos de pedidos distintos, en particiones distintas, no tienen un orden definido entre sí. Los diseños que necesitan un orden global deben usar una sola partición (renunciando al paralelismo) o, mejor, rediseñarse para que solo importe el orden por entidad.

Particiones, offsets y grupos de consumidores



Cada partición es un log ordenado; un grupo de consumidores reparte las particiones entre sus miembros, y cada grupo lleva el control de sus propios offsets.

Garantías de entrega, y lo que significa realmente 'exactamente una vez'

Como las redes y los procesos fallan (Sección 6.1), un broker y sus clientes deben elegir qué ocurre cuando una caída o un timeout deja sin aclarar si un mensaje se procesó. Hay tres garantías posibles, y la elección depende de cuándo se registran las confirmaciones y los offsets.

Garantías de entrega

Garantía	Qué promete	Cómo se produce	Consecuencia típica
Como máximo una vez (at-most-once)	Cada mensaje se procesa una vez o ninguna; nunca dos veces.	El consumidor registra su posición (confirma el offset o envía la confirmación) antes de procesar, o el productor nunca reintenta.	Pueden perderse mensajes tras una caída. Aceptable para muestras de métricas o avisos de ubicación de los repartidores, donde la siguiente actualización sustituye a la anterior.
Al menos una vez (at-least-once)	Cada mensaje se procesa una o más veces; nunca se pierde.	El productor reintenta hasta recibir la confirmación, y el consumidor confirma su posición solo después de procesar.	Duplicados tras caídas, reintentos y rebalanceos. Es la opción por defecto de la mayoría de los brokers y la elección habitual para los eventos de negocio.
Efectivamente una vez (effectively-once)	El efecto de cada mensaje se aplica exactamente una vez, aunque el mensaje pueda entregarse más de una vez.	Entrega al menos una vez más un consumidor idempotente, o una transacción que almacena juntos el resultado y la posición del consumidor.	El significado práctico de 'exactamente una vez' en los sistemas de negocio; exige un diseño deliberado en cada consumidor.

'Exactamente una vez' necesita una definición cuidadosa. La *entrega* exactamente una vez a través de una red no fiable no es alcanzable en general: un emisor que no recibe confirmación no puede saber si debe reenviar. Lo que los sistemas pueden ofrecer es un *efecto* exactamente una vez dentro de un límite definido. Kafka, por ejemplo, ofrece un productor idempotente y transacciones desde la versión 0.11 (2017), de modo que un ciclo de lectura, procesamiento y escritura de topics de Kafka a topics de Kafka confirma su salida y los offsets de su consumidor de forma atómica. Esa garantía se detiene en el borde de Kafka. Si el consumidor cobra una tarjeta, envía un correo electrónico o escribe en otra base de datos, una caída entre esa acción y la confirmación del offset hará que la acción se repita, a menos que el propio consumidor sea idempotente o almacene sus offsets en la misma transacción que sus resultados.

Por eso, la regla práctica para los arquitectos es sencilla: da por hecho que la entrega es al menos una vez y haz que cada consumidor sea un consumidor idempotente. Da a cada evento un identificador único; en el consumidor, registra los identificadores procesados en la misma transacción local que el cambio de estado y omite cualquier evento ya registrado. Siempre que sea posible, haz también que la operación sea idempotente por naturaleza, por ejemplo 'poner el estado en PAID si está en AWAITING_PAYMENT' en lugar de 'añadir un pago'.

```
# Idempotent consumer: processing the same event twice has the effect of once
def handle_payment_authorised(event, db):
    with db.transaction():                        # one local ACID transaction
        inserted = db.execute(
            "INSERT INTO processed_events (event_id) VALUES (%s) "
            "ON CONFLICT (event_id) DO NOTHING",
            [event["event_id"]],
        ).rowcount
        if inserted == 0:
            return                                # seen before: do nothing
        db.execute(
            "UPDATE orders SET status = 'PAID' WHERE order_id = %s "
            "AND status = 'AWAITING_PAYMENT'",
            [event["order_id"]],
        )
    # commit the broker offset only after the transaction has committed
```

Un consumidor idempotente: el registro de eventos procesados y el cambio de estado se confirman juntos, así que un evento reentregado no cambia nada.

Esquemas y versionado de los eventos

Una vez que otros equipos consumen un evento, su estructura es un contrato publicado, igual que una API, y dura más: con un broker basado en logs, los consumidores pueden leer eventos escritos hace meses. Un buen evento tiene un identificador único, un tipo, una versión del esquema, el momento en que ocurrió, los identificadores de las entidades implicadas y los datos de negocio que necesitan los consumidores. El dinero se almacena en unidades menores con una divisa, nunca como un número de coma flotante (Capítulo 0).

```
{
  "event_id": "9f1c2e7a-4b0d-4e55-9a51-0c3d8f2b6a10",
  "type": "OrderPlaced",
  "schema_version": 2,
  "occurred_at": "2026-10-22T18:42:07Z",
  "order_id": "ord-4711",
  "customer_id": "cus-208",
  "restaurant_id": "res-31",
  "items": [{"dish_id": "d-5", "qty": 2, "price_minor": 1150}],
  "total_minor": 2450,
  "currency": "GBP"
}
```

Un evento OrderPlaced que usa transferencia de estado transportada por eventos: identificadores, versión, momento y los datos que necesitan los consumidores.

Los esquemas deben evolucionar de forma compatible hacia atrás: añadir campos opcionales, nunca eliminar ni renombrar campos ni cambiar su significado sin una versión nueva, y nunca reutilizar el nombre de un campo para algo distinto. Muchos equipos lo imponen con un formato de esquema como Avro, Protocol Buffers o JSON Schema y un registro de esquemas que rechaza los cambios incompatibles en el momento de publicar. La especificación CloudEvents, un proyecto graduado de la Cloud Native Computing Foundation, estandariza los atributos comunes del sobre, como id, source, type y time. Los modos de compatibilidad y las técnicas de migración se tratan en evolución de esquemas.

Manejo de errores: reintentos, mensajes envenenados y colas de mensajes fallidos

Los consumidores fallan por dos motivos muy diferentes, y necesitan un tratamiento distinto. Los **fallos transitorios**, como una base de datos que no está disponible durante un momento o un timeout al llamar a otro servicio, suelen tener éxito al reintentar; reintenta unas pocas veces con backoff exponencial y jitter (esperas crecientes y aleatorias) para no saturar una dependencia que ya tiene problemas. Los **fallos permanentes**, como un evento mal formado o uno que infringe una regla de negocio, fallarán siempre. Un 'mensaje envenenado' (poison message) así no debe bloquear al consumidor para siempre, lo que en un log particionado detendría todos los eventos posteriores de esa partición.

La respuesta estándar es una cola de mensajes fallidos (dead-letter queue, DLQ): tras un número fijado de intentos, el mensaje se mueve a un canal aparte, con el error registrado, y el consumidor sigue adelante. Una DLQ solo es útil si alguien la vigila: genera alertas según su tamaño, asigna su propiedad a un equipo concreto, ofrece herramientas para inspeccionar,

corregir y volver a reproducir los mensajes, y decide de antemano qué garantías de orden se pierden cuando un mensaje se reproduce tarde. Algunos equipos añaden uno o más topics de reintento intermedios con esperas más largas antes de la DLQ.

Características de la arquitectura dirigida por eventos

Característica	Valoración	Razón
Escalabilidad y elasticidad	Alto	Los consumidores escalan de forma independiente y absorben las ráfagas gracias al búfer del broker; las particiones permiten el paralelismo.
Capacidad de respuesta	Alto	El publicador responde en cuanto el evento queda registrado de forma duradera; el trabajo lento se hace después.
Tolerancia a fallos	Alto	Un consumidor puede estar caído durante minutos y ponerse al día desde el broker; los fallos no se propagan en cascada hacia el publicador.
Extensibilidad	Alto	Los consumidores nuevos se suscriben a los eventos existentes sin ningún cambio en el publicador.
Consistencia de los datos	Bajo	El estado se propaga entre los servicios con retraso; los lectores ven datos desactualizados durante un tiempo.
Capacidad de prueba y de depuración	Bajo	El comportamiento surge de reacciones asíncronas; reproducir una secuencia de eventos entre servicios es difícil sin un buen trazado.
Visibilidad del flujo de trabajo	De baja (coreografía) a media (mediador)	En la coreografía, nadie es dueño del flujo de extremo a extremo; un mediador lo hace explícito.
Simplicidad	Bajo	Hay que diseñar los brokers, los esquemas, los reintentos, el manejo de mensajes fallidos, el orden y los duplicados.

Cuándo usarla y errores habituales

La arquitectura dirigida por eventos encaja cuando muchas partes de un negocio deben reaccionar a los mismos hechos, cuando el trabajo puede hacerse después de que responda la solicitud del usuario, cuando la carga llega a ráfagas y hay que suavizarla, y cuando se esperan consumidores nuevos con el tiempo. Encaja peor cuando un usuario necesita una respuesta inmediata y definitiva que depende de varios servicios (aunque una llamada síncrona para la respuesta más eventos para el seguimiento es un híbrido habitual), o cuando se exige consistencia fuerte entre servicios.

ATENCIÓN

Errores comunes

- **Comandos disfrazados de eventos.** `SendConfirmationEmail` publicado en un topic es un comando con el acoplamiento oculto; publica `OrderPlaced` y deja que Notificaciones decida.
- **Suponer que la entrega es exactamente una vez.** Un consumidor que cobra una tarjeta o envía un correo electrónico sin una comprobación de idempotencia acabará haciéndolo dos veces.
- **Suponer un orden global.** Los eventos de particiones distintas, o de productores distintos, pueden llegar en cualquier orden; diseña el orden por entidad con claves y números de versión.
- **Escribir en la base de datos y publicar por separado.** Si el servicio cae entre ambas operaciones, el evento se pierde o anuncia algo que nunca ocurrió. El patrón de bandeja de salida transaccional (transactional outbox) lo resuelve.
- **Espagueti de eventos.** Decenas de servicios que reaccionan a los eventos de los demás sin ningún mapa de los flujos. Mantén un catálogo de eventos, traza con identificadores de correlación y usa un mediador donde un flujo de trabajo necesite un propietario.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿`ReserveCourier` es un evento o un comando, y cómo lo sabrías?**

Un comando: es una instrucción dirigida a un único manejador (Despacho), que puede rechazarla. Un evento se nombraría en tiempo pasado, como `CourierReserved`.

▼ **Un topic de Kafka tiene 6 particiones y un grupo de consumidores tiene 10 consumidores. ¿Cuántos consumidores estarán ocupados?**

Como máximo 6. Dentro de un grupo, cada partición se asigna a un solo consumidor, así que 4 consumidores quedan inactivos.

▼ **¿Por qué el soporte de Kafka para 'exactamente una vez' no impide que un consumidor envíe dos veces un correo electrónico de confirmación?**

Sus transacciones abarcan la lectura y la escritura en topics de Kafka y la confirmación de los offsets; un correo electrónico enviado a un sistema externo queda fuera de ese límite, así que una caída antes de confirmar el offset provoca un reprocesamiento, a menos que el consumidor sea idempotente.

▼ **¿Qué diferencia hay entre la notificación de eventos y la transferencia de estado transportada por eventos?**

La notificación envía una pequeña señal, a menudo solo un identificador, así que los receptores vuelven a llamar para obtener los detalles; la transferencia de estado incluye los datos que necesitan los receptores, de modo que pueden mantener su propia copia y evitar llamar al publicador.

▼ **Inténtalo: Un consumidor a veces ve OrderCancelled antes que OrderPaid para un mismo pedido. El topic tiene 12 particiones y los eventos tienen claves aleatorias. También procesa ocasionalmente un evento dos veces. Diagnostica ambos problemas y da soluciones.**

El orden solo se mantiene dentro de una partición, y las claves aleatorias dispersan los eventos de un pedido entre particiones. Usa el ID de pedido como clave de los eventos, para que todos los eventos de un pedido lleguen a la misma partición y en orden. Los duplicados son normales con la entrega al menos una vez, así que haz que el consumidor sea idempotente, por ejemplo registrando los ID de los eventos procesados en la misma transacción que su actualización. Incluye un número de versión y envía los mensajes defectuosos a una cola de mensajes fallidos monitorizada en lugar de bloquear la partición.

FUENTES

- Martin Fowler: What do you mean by "Event-Driven"? (2017): https://martinfowler.com/articles/201701-event-driven.html
- Apache Kafka: Introduction (topics, partitions and ordering): https://kafka.apache.org/intro
- Confluent documentation: Kafka message delivery guarantees: https://docs.confluent.io/kafka/design/delivery-semantics.html
- Enterprise Integration Patterns: Idempotent Receiver: https://www.enterpriseintegrationpatterns.com/patterns/messaging/IdempotentReceiver.html
- CloudEvents specification: https://cloudevents.io/

SECCIÓN 6.5 DE 6.8 · INTERMEDIO · 37 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Comunicación síncrona y asíncrona: REST, gRPC, GraphQL y mensajería

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.6 API e integración
- 6.1 Por qué distribuir
- 6.4 Dirigida por eventos

Términos anteriores que utiliza: HTTP, Código de estado HTTP, Fallo parcial, Cambio incompatible (breaking change), Publicación-suscripción (publish-subscribe)

LA IDEA EN BREVE

Los servicios se comunican de dos maneras. Una llamada telefónica es síncrona: esperas en la línea hasta obtener una respuesta y, si la otra persona no está, no obtienes nada. Una carta es asíncrona: la envías y sigues con tu día, y la respuesta llega más tarde. Las llamadas telefónicas son sencillas, pero atan a ambas personas al mismo momento; las cartas son más flexibles, pero no sabes enseguida si tu solicitud ha funcionado. Además, las dos partes necesitan un acuerdo compartido y por escrito sobre lo que se puede pedir y lo que se recibe, y una forma de cambiar ese acuerdo sin perjudicar a quienes dependen de él.

LO QUE NECESITAS SABER

- Las llamadas síncronas crean acoplamiento temporal: quien llama y quien recibe la llamada deben estar activos a la vez, de modo que a lo largo de una cadena las latencias se suman y las disponibilidades se multiplican; la mensajería asíncrona elimina ese acoplamiento, pero no el acoplamiento de formato ni el semántico.
- REST es el conjunto de restricciones de Fielding (cliente-servidor, sin estado, almacenable en caché, interfaz uniforme, sistema en capas y, opcionalmente, código bajo demanda); la mayoría de las API reales alcanzan el nivel 2 del modelo de madurez de Richardson, no el nivel 3 de hipermedia.
- El RFC 9110 define GET, HEAD, OPTIONS, PUT y DELETE como idempotentes, y POST como ni seguro ni idempotente; los clientes solo deberían reintentar automáticamente las solicitudes idempotentes tras un timeout, un 502, un 503 o un 504, por lo que POST necesita una clave de idempotencia para poder reintentarse con seguridad.

- gRPC usa Protocol Buffers sobre HTTP/2, con código generado y streaming, y es adecuado para llamadas internas; GraphQL permite que clientes muy distintos elijan los campos, pero necesita agrupación de consultas (batching), límites al costo de las consultas y autorización a nivel de campo.
- Las colas entregan cada mensaje a un solo consumidor de entre varios que compiten; los topics entregan una copia a cada suscriptor.
- Las API duraderas modelan recursos del dominio, paginan con cursores, devuelven problem details del RFC 9457, cambian solo de forma compatible hacia atrás y se versionan o usan expandir y contraer cuando tienen que romper la compatibilidad.
- OpenAPI y AsyncAPI describen contratos HTTP y de mensajes; las pruebas de contrato dirigidas por el consumidor, como Pact, permiten a los proveedores saber qué cambios romperían a qué consumidores.

Dos formas de comunicarse y el acoplamiento que crea cada una

Una vez que un sistema se divide en partes que se despliegan por separado (consulta por qué distribuir), esas partes deben comunicarse a través de una red, y la forma en que se comunican da forma al sistema tanto como la forma en que se dividen. Hay dos modos básicos. En la comunicación síncrona, quien llama envía una solicitud y espera la respuesta antes de continuar, como en una llamada telefónica. En la comunicación asíncrona, el emisor entrega un mensaje a un intermediario, normalmente un intermediario de mensajes (message broker), y sigue adelante; el receptor recoge el mensaje cuando está listo, como una carta en un buzón. La Sección 0.6 presentó la diferencia; aquí se examina como una decisión arquitectónica.

Las llamadas síncronas crean acoplamiento temporal: ambas partes deben estar en ejecución y accesibles en el mismo momento. Por tanto, quien llama es, como mucho, tan disponible como el producto de las disponibilidades de sus dependencias, y su tiempo de respuesta, incluida su latencia de cola, contiene el de ellas: es la aritmética de la Sección 6.1. Un servicio de pagos lento hace lento al servicio de pedidos y, si todos los hilos del servicio de pedidos están esperándolo, el servicio de pedidos deja de responder a nadie. Los timeouts, los reintentos y los circuit breakers limitan el daño y se tratan en patrones de resiliencia.

La mensajería asíncrona elimina el acoplamiento temporal: el intermediario guarda el mensaje, así que el receptor puede estar lento, ocupado o caído brevemente sin que el emisor lo note. No elimina todo tipo de acoplamiento. Ambas partes siguen dependiendo del formato y del significado del mensaje (acoplamiento de formato y semántico), y del propio intermediario, que se convierte en infraestructura crítica. El emisor también pierde el conocimiento inmediato del resultado, por lo que el diseño debe hacer frente a trabajo que falla más tarde, a mensajes que llegan dos veces y a mensajes que llegan en otro orden. La asincronía traslada la complejidad en lugar de eliminarla.

Una regla práctica útil: usa una llamada síncrona cuando quien llama realmente no puede continuar sin la respuesta ("¿está autorizada esta tarjeta?"), y mensajería asíncrona cuando quien llama solo necesita saber que el trabajo se hará ("envía el recibo", "avisa al restaurante"). Muchos buenos diseños mantienen corto y síncrono el camino que ve el usuario y ponen todo lo demás detrás de una cola.

Acoplamiento temporal, con y sin intermediario



En una cadena síncrona, quien llama queda bloqueado y depende de todos los servicios posteriores; con un intermediario en medio, el emisor y el receptor ya no necesitan estar disponibles en el mismo momento.

SI ESTO ES NUEVO PARA TI

Paso a paso: ¿una llamada telefónica o una carta?

La idea más difícil de esta sección es de qué te libera, y de qué no, enviar un mensaje en lugar de esperar una respuesta. El Capítulo 0 presentó los dos modos; sigue un pedido de comida a través de ambos.

- Un cliente toca Hacer pedido. El servicio de pedidos debe saber si la tarjeta está autorizada antes de poder decir que sí, así que llama al servicio de pagos y espera: una llamada síncrona, como una llamada telefónica.
- Si el servicio de pagos está caído, no se puede hacer el pedido: ambos deben estar en ejecución y accesibles en el mismo momento. Eso es acoplamiento temporal.
- A continuación, el cliente necesita un recibo por correo electrónico. El servicio de pedidos no necesita esperar a que se envíe, así que deja un mensaje ReceiptNeeded en un intermediario de mensajes, una especie de buzón, y responde al cliente de inmediato: asíncrono, como una carta.
- Supón que el servicio de correo electrónico está caído durante diez minutos. El intermediario guarda el mensaje y el recibo simplemente llega tarde. Si el servicio de pedidos hubiera llamado al servicio de correo electrónico y hubiera esperado, todos los pedidos de esos diez minutos habrían fallado o se habrían quedado colgados.
- La carta no libera del todo a las dos partes. Si el servicio de pedidos cambia el nombre de un campo del mensaje, por ejemplo de email a emailAddress, el servicio de correo electrónico ya no puede leerlo: ambas partes siguen dependiendo de la misma estructura y del mismo significado acordados para lo que se envía. Y el servicio de pedidos ya no sabe enseguida si el correo electrónico salió.
- **La lección:** llama y espera solo cuando no puedas continuar sin la respuesta; envía un mensaje cuando solo necesites saber que el trabajo se hará. En ambos casos, el formato de lo que se envía es una promesa que debe cambiarse con cuidado, como explica el final de esta sección.

REST tal como lo definió Fielding

REST (Representational State Transfer) fue definido por Roy Fielding en su tesis doctoral de 2000 como un estilo arquitectónico para grandes sistemas hipermedia en red: la propia web. Fielding lo derivó como un conjunto de restricciones, cada una de las cuales aporta una cualidad. **Cliente-servidor** separa la interfaz de usuario del almacenamiento de datos para que ambos evolucionen de forma independiente. La interacción **sin estado** significa que cada solicitud lleva todo lo necesario para entenderla, de modo que cualquier instancia del servidor puede atenderla, lo que favorece la escalabilidad y la recuperación. Las respuestas

almacenables en caché indican si pueden reutilizarse, lo que reduce la latencia y la carga. Una **interfaz uniforme** (recursos identificados por URI, manipulados mediante representaciones, mensajes autodescriptivos e hipermedia como motor del estado de la aplicación) permite que intermediarios genéricos, como cachés y proxies, entiendan el tráfico. Un **sistema en capas** permite que proxies, gateways y balanceadores de carga se sitúen en el camino de forma invisible. El **código bajo demanda**, como el JavaScript que se envía a un navegador, es la única restricción opcional.

La mayoría de las API que se describen como RESTful solo cumplen algunas de ellas. El modelo de madurez de Leonard Richardson, presentado en 2008 y popularizado por Martin Fowler en 2010, describe los pasos. En el **nivel 0**, HTTP es solo un túnel: cada llamada es un POST a un único endpoint con la operación indicada en el cuerpo, como en SOAP. En el **nivel 1**, la API tiene recursos separados (`/orders/1042` , `/restaurants/7`). En el **nivel 2**, usa los métodos y los códigos de estado HTTP tal como los define HTTP. En el **nivel 3**, las respuestas llevan controles hipermedia: enlaces que le dicen al cliente qué puede hacer a continuación (un pedido realizado contiene un enlace para cancelarlo, hasta que la cancelación ya no está permitida).

Los profesionales discrepan de verdad sobre el nivel 3. Fielding ha insistido en que una API sin hipermedia no es REST. La mayoría de las API públicas se quedan en el nivel 2 porque sus clientes se escriben a partir de la documentación en lugar de descubrirse en tiempo de ejecución, y el beneficio de seguir enlaces rara vez compensa el diseño adicional. Una lectura pragmática es que el nivel 2, aplicado de forma coherente, aporta la mayor parte del valor; la hipermedia se justifica cuando el servidor debe guiar a muchos clientes escritos de forma independiente a través de un flujo de trabajo que cambia.

La semántica HTTP que importa para la arquitectura

La semántica de HTTP está definida en el RFC 9110 (2022). Dos propiedades de los métodos importan más que cualquier otra para el diseño distribuido. Un método **seguro** solo lee: GET, HEAD y OPTIONS no cambian nada que el cliente haya pedido cambiar. Un método idempotente puede cambiar el estado, pero repetirlo tiene el mismo efecto previsto que enviarlo una vez: PUT (sustituir este recurso por esta representación) y DELETE son idempotentes, igual que todos los métodos seguros. POST no es ninguna de las dos cosas, y PATCH no es idempotente en general. La propiedad importa porque las redes fallan entre el momento en que se procesa una solicitud y el momento en que llega la respuesta. El RFC 9110 dice que las solicitudes idempotentes pueden repetirse automáticamente tras un fallo así, y que los clientes no deberían reintentar automáticamente solicitudes no idempotentes a menos que sepan que, de hecho, es seguro repetirlas.

Los códigos de estado forman parte del contrato, porque el código del cliente se ramifica según ellos; la Sección 0.6 presentó los más comunes. Los códigos de éxito deberían decir qué ha ocurrido: 201 Created (con un encabezado `Location`) cuando se crea un recurso, 202 Accepted cuando el trabajo solo se ha puesto en cola, 204 No Content para un éxito sin cuerpo. Para los conflictos, 409 significa que la solicitud choca con el estado actual del recurso, 412 que ha fallado una precondición y 422 que una solicitud bien formada no es válida semánticamente. La regla arquitectónica trata de los reintentos: una respuesta 4xx volverá a fallar si la solicitud se envía sin cambios (429 Too Many Requests es la excepción: espera lo indicado y luego reintenta), mientras que un timeout, un 502, un 503 o un 504 pueden tener éxito más tarde. Los clientes solo deberían reintentar estos, con backoff y jitter, y solo cuando la solicitud es idempotente o lleva una clave de idempotencia.

Otras dos funciones de HTTP resuelven problemas distribuidos a bajo costo. Las **solicitudes condicionales** proporcionan control de concurrencia optimista: un cliente lee un pedido con su etiqueta de versión (`ETag`) y luego envía `If-Match` con esa etiqueta al actualizarlo; si otra persona ha cambiado el pedido entretanto, el servidor responde 412 y no se sobrescribe nada. Los **encabezados de caché** (`Cache-Control`, `ETag`) permiten que navegadores, CDN y proxies reutilicen respuestas, lo que a menudo es la táctica de rendimiento más barata disponible.

Métodos HTTP: seguros, idempotentes o ninguna de las dos cosas (RFC 9110; PATCH según el RFC 5789)

Método	Uso típico	¿Seguro?	¿Idempotente?	¿Se puede reintentar a ciegas?
GET, HEAD	Leer un recurso	Sí	Sí	Sí
PUT	Crear o sustituir en un URI conocido	No	Sí	Sí
DELETE	Eliminar un recurso	No	Sí	Sí (una segunda llamada puede devolver 404, pero el efecto es el mismo)
POST	Crear en una colección o ejecutar una acción	No	No	Solo con una clave de idempotencia u otra comprobación de duplicados
PATCH	Actualización parcial	No	No en general	Solo si el parche está escrito para ser idempotente o está condicionado a una versión

gRPC: primero los contratos, binario en la red

gRPC es un framework de llamadas a procedimientos remotos que Google liberó como código abierto en 2015 como sucesor de Stubby, el sistema RPC que había usado internamente durante más de una década; la versión 1.0 llegó en 2016. El contrato se escribe en un archivo `.proto` usando Protocol Buffers, un formato de serialización binario compacto, y los generadores de código producen stubs de cliente tipados y esqueletos de servidor en muchos lenguajes. Las llamadas viajan sobre HTTP/2, que multiplexa muchas solicitudes simultáneas en una sola conexión.

gRPC ofrece cuatro tipos de llamada: **unaria** (una solicitud, una respuesta), **streaming del servidor** (una solicitud, un flujo de respuestas, como las posiciones de los repartidores en tiempo real), **streaming del cliente** (un flujo de solicitudes, una respuesta, como al subir telemetría) y **streaming bidireccional**. Los clientes fijan un **plazo (deadline)**; cuando vence, la llamada termina con `DEADLINE_EXCEEDED`, y los plazos pueden propagarse a lo largo de una cadena de llamadas para que el trabajo se detenga en todas partes cuando ya nadie espera la respuesta.

Sus puntos fuertes son la eficiencia, el tipado fuerte y el código generado, lo que lo hace popular para las llamadas internas entre servicios en organizaciones políglotas. Sus costos son la imagen especular. Las cargas útiles binarias son difíciles de leer en los logs y con herramientas corrientes. Los navegadores no pueden llamar a gRPC directamente, por lo que las API públicas y las orientadas a la web suelen necesitar un proxy o una fachada REST o GraphQL. La caché HTTP no se aplica de la forma habitual. La compatibilidad depende de la disciplina con los números de campo: es el número de un campo, no su nombre, lo que lo identifica en la red, así que los números nunca deben reutilizarse, y los campos eliminados deberían marcarse como reservados.

```
syntax = "proto3";

service Dispatch {
  rpc AssignCourier (AssignRequest) returns (Assignment);
  rpc TrackCourier (TrackRequest) returns (stream Position);
}

message AssignRequest {
  string order_id = 1;
  string restaurant_id = 2;
  reserved 3;            // field removed in v2: never reuse 3
}

message Position {
  double lat = 1;
  double lng = 2;
  int64 recorded_at_ms = 3;
}
```

Un contrato gRPC para el servicio de despacho de repartidores de la plataforma de reparto de comida, con una llamada unaria y una de streaming del servidor (sintaxis de Protocol Buffers).

GraphQL: el cliente elige la forma

GraphQL es un lenguaje de consulta y un entorno de ejecución para API que Facebook desarrolló internamente a partir de 2012 para sus apps móviles y publicó en 2015, con un borrador de especificación y una implementación de referencia; en 2018 pasó a la GraphQL Foundation, neutral respecto a los proveedores. El servidor publica un **esquema** tipado; los clientes envían consultas a un único endpoint indicando exactamente los campos que quieren, posiblemente a través de objetos relacionados, y reciben una respuesta JSON con la misma forma. Las mutaciones cambian datos y las suscripciones envían actualizaciones.

Su principal beneficio es para clientes muy distintos. Una app móvil de reparto de comida con una conexión lenta puede pedir el estado de un pedido, el nombre del restaurante y el nombre de pila del repartidor en un solo viaje de ida y vuelta, sin recibir los otros veinte campos que necesita el panel web. Eso evita tanto el exceso de datos (over-fetching) como la cadena de solicitudes que puede exigir una API REST recurso a recurso, y puede sustituir a un backend separado para cada frontend. El esquema, además, se documenta a sí mismo y permite que las herramientas validen las consultas antes de ejecutarlas.

Los compromisos son reales y los paga sobre todo el equipo del servidor. Cada campo lo rellena un **resolver**, así que una implementación ingenua que obtiene por separado el restaurante de cada pedido produce el problema de las consultas N+1: una consulta para cincuenta pedidos y luego cincuenta más para sus restaurantes. La solución habitual es la agrupación de consultas (batching) y la caché por solicitud, tal como popularizó la biblioteca DataLoader. Como los

clientes pueden componer consultas arbitrariamente profundas, los servidores necesitan límites de profundidad y de costo de las consultas, timeouts y, a menudo, una lista de consultas aprobadas (persistidas) para evitar sobrecargas accidentales o malintencionadas. La caché HTTP es más débil cuando todo es un POST a una única URL. La autorización debe aplicarse por campo o por objeto, no por URL. Y los errores suelen notificarse en un array `errors` dentro de una respuesta HTTP correcta, algo que pasará por alto la monitorización basada en códigos de estado.

Mensajería: colas y topics

La mensajería traslada la conversación a un intermediario. Como explica arquitectura dirigida por eventos, una cola entrega cada mensaje a uno de varios consumidores que compiten, la forma estándar de escalar un grupo de workers, mientras que un topic entrega una copia a cada suscriptor, de modo que los servicios del restaurante, de despacho de repartidores y de notificaciones pueden reaccionar cada uno a `OrderPlaced` sin que el servicio de pedidos sepa que existen. Esa sección también trata las garantías de entrega, el orden, los duplicados y las colas de mensajes fallidos; colas y back pressure explica cómo las colas absorben los picos de carga.

La mensajería también puede transportar solicitud y respuesta: el solicitante envía un comando con un **ID de correlación** y una dirección de respuesta, y empareja la respuesta que llegue con su solicitud. Así se conserva el desacoplamiento temporal de un intermediario y aun así se obtiene una respuesta, a costa de más piezas móviles.

REST, gRPC, GraphQL y mensajería comparados

Aspecto	REST sobre HTTP	gRPC	GraphQL	Mensajería (colas y topics)
Interacción	Solicitud-respuesta síncrona	Síncrona; unaria o streaming	Consultas y mutaciones síncronas; las suscripciones envían datos	Asíncrona; enviar y olvidar (fire-and-forget) o solicitud-respuesta mediante ID de correlación
Contrato	Documento OpenAPI (opcional pero habitual)	Archivos .proto (obligatorios)	Esquema (obligatorio)	Documento AsyncAPI y esquemas de mensajes (opcionales pero recomendables)
Carga útil	Normalmente texto JSON	Protocol Buffers binario	JSON	Cualquiera: JSON, Avro, Protocol Buffers
Acoplamiento temporal	Sí	Sí	Sí	No: el intermediario almacena los mensajes
Caché	Excelente: cachés HTTP estándar y CDN	Poco incorporado	Más difícil: necesita consultas persistidas o cachés en el cliente	No procede
Soporte en navegadores	Nativo	Necesita un proxy (gRPC-Web)	Nativo	No directamente; mediante un backend o WebSockets
Mejor uso	API públicas y de socios; CRUD orientado a recursos	Llamadas internas entre servicios de alto volumen y baja latencia; streaming	Muchos tipos de cliente con necesidades de datos distintas; agregar varios servicios	Trabajo que puede hacerse más tarde; fan-out a muchos consumidores; suavizar la carga
Principales trampas	API conversadoras (chatty); convenciones incoherentes	Opaco al depurar; disciplina con los números de campo	Resolvers N+1; costo de consulta sin límite; autorización a nivel de campo	Duplicados, orden, flujos invisibles, operación del intermediario

El panorama de la mensajería: elegir una cola o un log

La primera decisión es entre una cola y un log. Una **cola** reparte trabajo, y un mensaje desaparece una vez procesado, lo que encaja con tareas como enviar un recibo, donde importan el reintento por mensaje y las colas de mensajes fallidos. Un **log** conserva los mensajes para lectores posteriores, lo que encaja con hechos que necesitan varios consumidores o que deben reproducirse, como los eventos de pedidos que reconstruyen un modelo de lectura. La Sección 6.4 explica Kafka, el log más conocido. La línea se ha difuminado: Kafka 4.2 declaró listos para producción sus share groups, similares a colas, y RabbitMQ añadió streams de solo anexado.

Otros cuatro nombres aparecen a menudo. **RabbitMQ** es un intermediario de mensajes (message broker) que puedes operar tú mismo o contratar como servicio alojado; sus colas encajan con la distribución de trabajo, y sus streams añaden reproducción y fan-out (abanico) a gran escala. **Amazon SQS y SNS** son servicios gestionados: los consumidores sondean una cola SQS, y un topic SNS envía una copia a cada cola suscrita, de modo que un evento llega a las colas de restaurantes, despacho y notificaciones, que se vacían cada una a su ritmo. **Google Cloud Pub/Sub** es publicación-suscripción gestionada con entrega al menos una vez (at-least-once). **NATS** es ligero: core NATS entrega como máximo una vez, por lo que un suscriptor que no esté conectado nunca ve el mensaje, mientras que su capa JetStream añade almacenamiento, reproducción y confirmaciones. Los límites y las funciones cambian, así que consulta la versión actual de cada uno.

Los consumidores competidores y el orden estricto tiran en sentidos opuestos. Aunque un broker entregue los mensajes en orden, los workers terminan en cualquier orden: un worker lento con `OrderPaid` puede terminar después de uno rápido con `OrderCancelled`. Las reentregas lo agravan; la documentación de RabbitMQ advierte de que con varios consumidores activos "cualquier reentrega puede cambiar el orden". La respuesta habitual es renunciar al orden global y mantener el orden por clave, con un máximo de un worker por clave a la vez. Kafka lo hace con particiones por clave y las colas FIFO de SQS con grupos de mensajes: el orden se mantiene dentro de un grupo y los grupos se ejecutan en paralelo. Las colas SQS estándar solo ofrecen un orden de mejor esfuerzo. Pub/Sub usa claves de ordenación, habilitadas en la suscripción. El consumidor activo único de RabbitMQ mantiene el orden dejando que solo un consumidor reciba de la cola, lo que renuncia al paralelismo en ella.

El orden por clave tiene un precio. Un mensaje lento o que falla retiene todos los mensajes posteriores de esa clave, y una clave muy activa acumula retrasos; la documentación de Pub/Sub también dice que reentregar un mensaje reentrega los posteriores de esa clave, incluso los ya confirmados. La entrega ordenada tampoco elimina los duplicados, por lo que los

consumidores deben seguir siendo idempotentes. Cuando se puede reconocer una actualización obsoleta, una alternativa es incluir un número de versión en cada evento y dejar que los consumidores ignoren todo lo que sea más antiguo que lo que ya han aplicado.

Diseñar API que perduren: recursos, paginación y filtrado

Modela los recursos en torno al dominio, no a la base de datos. Expón los sustantivos con los que piensan tus consumidores (`/orders` , `/orders/1042/items` , `/restaurants/7/menu`), no tablas ni servicios internos. Las acciones que no encajan en crear, leer, actualizar y eliminar normalmente pueden modelarse a su vez como recursos: `POST /orders/1042/cancellation` crea una cancelación, que puede llevar un motivo y ser rechazada con un 409 si la comida ya va de camino. Así se conserva la interfaz uniforme y a la vez se expresa con honestidad el proceso de negocio.

Pagina todas las colecciones, porque las colecciones crecen. La **paginación por offset** (`?offset=40&limit=20`) es sencilla y permite saltar a la página 9, pero se vuelve lenta en páginas profundas, porque la base de datos debe saltarse todas las filas anteriores, y muestra duplicados u omite elementos cuando se insertan filas mientras alguien está paginando. La paginación por cursor devuelve un token opaco que apunta justo después del último elemento (`?after=eyJpZCI6MTA0Mn0&limit=20`); sigue siendo rápida y estable ante las inserciones, a costa de no permitir el acceso aleatorio. Para feeds y para datos grandes o cambiantes, los cursores son la opción por defecto más segura. Limita siempre el tamaño de página en el servidor.

El **filtrado y la ordenación** deberían usar parámetros de consulta predecibles (`?status=delivered&placed_after=2026-10-01&sort=-placed_at`) y apoyarse en índices; una API que permite a los clientes filtrar por cualquier campo invita a hacer consultas que la base de datos no puede responder con rapidez. Admite solo los filtros que necesitan los consumidores y documéntalos en el contrato.

Trabajo de larga duración: 202, sondeo y webhooks fiables

Algunas solicitudes tardan demasiado para mantenerlas abiertas, como importar la carta completa de una cadena de restaurantes. La API comprueba la solicitud y devuelve `202 Accepted` con una cabecera `Location` que nombra un recurso de estado, y quizá `Retry-After` . El RFC 9110 dice que el 202 es "intencionadamente evasivo": el trabajo aún puede rechazarse más tarde. El cliente sondea, recibe 200 con el estado y luego 303 hacia el resultado. Para evitar el sondeo, el servidor puede llamar a un **webhook**. Un remitente debe reintentar con espera exponencial (back-off) y jitter (fluctuación aleatoria), mantener un mismo ID de evento en los reintentos y firmar cada entrega con una marca de tiempo. El receptor debe

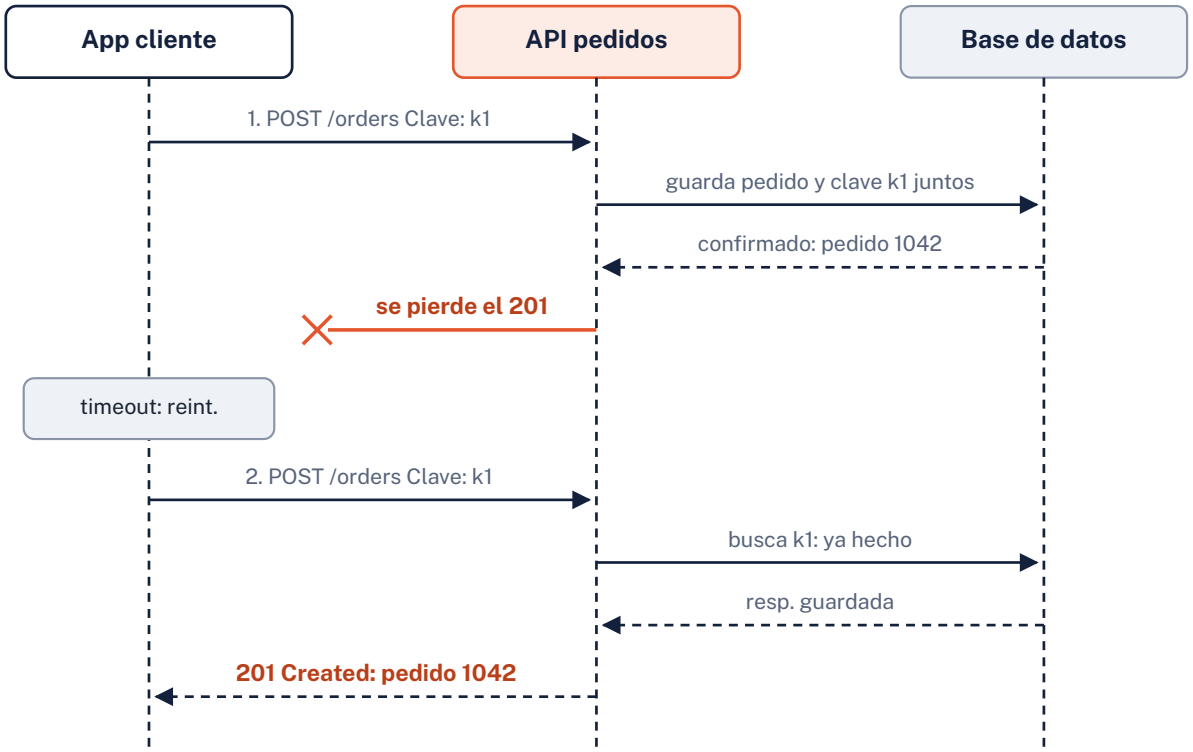
verificar la firma (Sección 8.4), responder 2xx rápidamente, poner el trabajo en cola e ignorar los duplicados. Stripe, por ejemplo, no garantiza el orden. Conserva el endpoint de estado como forma de recuperar las entregas perdidas.

Claves de idempotencia: hacer que POST sea seguro de reintentar

El fallo parcial (Sección 6.1) deja a un cliente que no ha recibido respuesta sin saber si su solicitud se procesó. Reintentar un GET es inofensivo; reintentar `POST /orders` puede crear un segundo pedido y cobrar dos veces al cliente. Una clave de idempotencia convierte el reintento peligroso en uno seguro. El cliente genera un valor único (normalmente un UUID aleatorio) para cada operación lógica y lo envía en un encabezado como `Idempotency-Key`. El servidor registra la clave junto con el resultado de la primera solicitud; cualquier solicitud posterior con la misma clave devuelve el resultado registrado en lugar de volver a hacer el trabajo.

Los proveedores de pagos popularizaron la técnica, y el grupo de trabajo HTTP APIs del IETF ha redactado un encabezado estándar `Idempotency-Key`. En el momento de escribir esto no se ha publicado como RFC (el último borrador, de octubre de 2025, ha caducado), pero sus reglas reflejan la práctica habitual: una clave reutilizada con un cuerpo de solicitud distinto debería rechazarse con un 422, un reintento que llega mientras la primera solicitud aún se está ejecutando debería recibir un 409 Conflict para que nunca se ejecuten dos copias a la vez, y las claves están limitadas a un cliente y caducan tras un periodo publicado, a menudo 24 horas. El ejemplo práctico aplica cada regla.

Un pedido reintentado con una clave de idempotencia



El primer intento tiene éxito, pero su respuesta se pierde; el reintento lleva la misma clave, así que la API devuelve el resultado guardado en lugar de crear el pedido 1043.

EJEMPLO PRÁCTICO

Ejemplo práctico: el doble toque en el proceso de pago

Un cliente con una conexión móvil débil toca Hacer pedido. La app genera la clave `7f3c…` y envía `POST /v1/orders`. El servicio de pedidos guarda la clave junto con un hash del cuerpo de la solicitud y, más adelante, la respuesta.

Intento 1	Se crea el pedido 1042 y se autoriza el pago; la clave, el hash del cuerpo y la respuesta 201 se guardan en la misma transacción de base de datos que el pedido. La respuesta se pierde cuando el teléfono pasa de wifi a datos móviles.
Intento 2 (reintento automático, misma clave)	El servicio encuentra la clave con el estado *completado*, comprueba que el hash del cuerpo coincide y devuelve el 201 guardado para el pedido 1042. Ni segundo pedido ni segundo cobro.
Intento 3 (el cliente vuelve a tocar mientras el intento 2 está en curso)	La app reutiliza la misma clave para el mismo carrito. Si la primera solicitud siguiera procesándose, el servicio respondería 409 y la app esperaría y reintentaría.
Un error de programación envía la misma clave con un carrito distinto	Los hashes del cuerpo no coinciden: el servicio responde 422 en lugar de devolver en silencio el pedido equivocado.
Pasadas 24 horas	La clave caduca y se elimina; la app nunca reintenta ese carrito antiguo.

La clave debe generarse una vez por operación lógica (por cada envío del carrito), no por cada intento HTTP; una clave nueva en cada reintento anula su propósito.

Guardar la clave en la misma transacción que el pedido es lo que hace fiable la comprobación: una caída no puede dejar un pedido sin su clave, ni una clave sin su pedido.

La misma idea, aplicada a mensajes en lugar de solicitudes HTTP, es el consumidor idempotente de arquitectura dirigida por eventos.

```
def create_order(request, db):
    key = request.headers.get("Idempotency-Key")
    if not key:
        return problem(400, "missing-idempotency-key")
    body_hash = sha256(request.body)
    with db.transaction():
        row = db.find_idempotency(request.client_id, key)
        if row and row.body_hash ≠ body_hash:
            return problem(422, "idempotency-key-reused")
        if row and row.status == "in_progress":
            return problem(409, "request-in-progress")
        if row:
            return row.saved_response          # replay, do nothing new
        db.insert_idempotency(request.client_id, key, body_hash, "in_progress")
        order = db.insert_order(request.json)
        response = created(order)
        db.complete_idempotency(request.client_id, key, response)
    return response
```

Tratamiento de una clave de idempotencia en el servidor (simplificado: una transacción de base de datos por solicitud; la restricción de unicidad sobre la clave impide que dos solicitudes simultáneas se ejecuten ambas).

Errores que las máquinas pueden leer: RFC 9457

Los clientes necesitan errores sobre los que puedan actuar de forma programática, y las personas necesitan errores que puedan entender. En lugar de que cada equipo invente una forma de error, usa Problem Details for HTTP APIs, el RFC 9457 (julio de 2023), que sustituyó al RFC 7807 (2016). Un objeto problem details tiene el tipo de medio `application/problem+json` y cinco miembros estándar: `type` (un URI que identifica el tipo de problema y que puede enlazar con documentación), `title` (un resumen breve y legible de ese tipo), `status` (el código de estado HTTP), `detail` (una explicación de esta ocurrencia concreta) e `instance` (un URI que identifica esta ocurrencia). Los miembros de extensión pueden transportar detalles legibles por máquina, como qué campos no superaron la validación.

Dos reglas mantienen seguras las respuestas de error. El código del cliente debería ramificarse según `type` y `status`, nunca según la redacción de `title` o `detail`, que pueden traducirse o mejorarse. Y los errores no deben filtrar detalles internos: las trazas de pila, los fragmentos de SQL y los nombres de host internos ayudan más a los atacantes que a quienes llaman. Registra los detalles con un ID de correlación y devuelve el ID en su lugar.

```
{
  "type": "https://api.fooddelivery.example/problems/outside-delivery-area",
  "title": "Address is outside the delivery area",
  "status": 422,
  "detail": "Restaurant 7 delivers within 4 km; the address is 6.2 km away.",
  "instance": "/v1/orders/attempts/9b1e",
  "max_distance_km": 4,
  "actual_distance_km": 6.2
}
```

Una respuesta problem details (HTTP 422, Content-Type: application/problem+json) para un pedido fuera de la zona de reparto.

Hacer evolucionar una API sin romper a sus consumidores

Una API con consumidores es una promesa, y la ley de Hyrum (Sección 0.6) advierte de que alguien depende de cada comportamiento observable. Por eso el cambio se rige por un principio: haz cambios compatibles hacia atrás libremente y evita los que rompen la compatibilidad. Los consumidores ayudan siendo **lectores tolerantes (tolerant readers)** (un término que usa Martin Fowler): ignoran los campos que no entienden y leen solo lo que necesitan, de modo que los añadidos nunca los rompen. La tabla siguiente resume las reglas habituales; la misma lógica, aplicada a los datos almacenados y a los esquemas de eventos, aparece en evolución de esquemas.

Cuando un cambio incompatible es inevitable, versiona. Las estrategias habituales son una versión en la ruta de la URL (`/v2/orders` : explícita, fácil de enrutar y de almacenar en caché, pero fomenta saltos grandes y poco frecuentes), un encabezado personalizado o un parámetro del tipo de medio (`Accept: application/vnd.fooddelivery.v2+json` : mantiene estables las URL, pero es menos visible y más difícil de probar en un navegador), y versiones con fecha fijadas por cuenta de cliente, que algunos grandes proveedores de API usan para lanzar continuamente pequeños cambios incompatibles mientras cada cliente actualiza cuando decide. Las API internas entre servicios de una misma organización a menudo evitan las versiones por completo mediante el enfoque de **expandir y contraer (expand and contract)**: añadir el nuevo campo o endpoint, migrar a todos los consumidores y después eliminar el antiguo. Sea cual sea la estrategia, anuncia las obsolescencias con antelación; los encabezados de respuesta `Deprecation` (RFC 9745, 2025) y `Sunset` (RFC 8594, 2019) permiten que un servidor indique a los clientes, de forma legible por máquina, que un endpoint está obsoleto y cuándo dejará de funcionar.

Cambios compatibles hacia atrás e incompatibles en una API

Cambio	¿Compatible?	Por qué
Añadir un campo opcional a la solicitud o un parámetro de consulta	Sí	Los clientes existentes no lo envían; el servidor debe aplicar un valor por defecto razonable.
Añadir un campo a una respuesta	Normalmente	Los lectores tolerantes lo ignoran; los clientes estrictos que rechazan campos desconocidos fallarán, así que documenta que los clientes deben tolerar los añadidos.
Añadir un nuevo endpoint o recurso	Sí	Nadie lo llama todavía.
Añadir un valor a una enumeración en una respuesta	Arriesgado	Los clientes con sentencias switch exhaustivas pueden fallar con el valor desconocido; documenta que pueden aparecer valores nuevos.
Hacer obligatorio un campo opcional de la solicitud	No	Los clientes existentes que lo omiten empiezan a fallar con un 400.
Eliminar o renombrar un campo, o cambiar su tipo o sus unidades	No	Los clientes que leen el campo antiguo no obtienen nada, u obtienen algo equivocado.
Cambiar el significado de un campo o de un código de estado	No	La ruptura más peligrosa: nada falla de forma visible, pero el comportamiento es incorrecto.
Endurecer la validación (longitud máxima más corta, formato más estricto)	No	Solicitudes que antes tenían éxito ahora fallan.

Describir contratos: OpenAPI y AsyncAPI

Un contrato escrito en un formato legible por máquina puede alimentar la documentación, la generación de clientes, los stubs de servidor, los servidores simulados (mock), el análisis estático (linting) y las pruebas. OpenAPI es el estándar para las API HTTP. Empezó como la especificación Swagger, que SmartBear donó en 2015 a la recién creada OpenAPI Initiative, bajo la Linux Foundation, y que cambió de nombre en 2016; la versión 3.1 (2021) alineó sus esquemas con JSON Schema, y la versión 3.2 llegó en septiembre de 2025. AsyncAPI aplica la misma idea a las API basadas en mensajes: describe los canales (colas y topics), las operaciones que envían o reciben por ellos y los esquemas de los mensajes; la versión 3.0 se lanzó en diciembre de 2023.

Los equipos eligen entre **primero el diseño (design-first)** (escribir y revisar el contrato, y luego implementarlo) y **primero el código (code-first)** (generar el contrato a partir de código anotado). Primero el diseño convierte la API en una decisión deliberada que los consumidores pueden revisar antes de construir nada, y es adecuado para API públicas y entre equipos; primero el código mantiene sincronizados el contrato y la implementación con menos esfuerzo, y es adecuado para API internas que cambian con rapidez. En cualquier caso, guarda el contrato en el control de versiones, analízalo con un linter en el pipeline y ejecuta una comprobación de cambios incompatibles que compare cada versión propuesta con la publicada.

Pruebas de contrato y contratos dirigidos por el consumidor

Las pruebas de extremo a extremo que despliegan todos los servicios juntos son lentas, inestables y se vuelven más lentas a medida que se multiplican los servicios; sin embargo, las pruebas unitarias no pueden detectar que un proveedor ha cambiado un campo del que depende un consumidor. Las pruebas de contrato cubren ese hueco: cada parte se prueba de forma aislada frente a una descripción compartida de sus interacciones. En los contratos dirigidos por el consumidor, un patrón que Ian Robinson describió en 2006, cada consumidor declara las partes de la API del proveedor que realmente usa, y el proveedor ejecuta las expectativas de todos sus consumidores como parte de su propia compilación. Así el proveedor sabe exactamente qué campos no lee nadie (seguros de cambiar) y qué cambios romperían a un consumidor concreto.

Pact, creado en realestate.com.au en 2013, es la herramienta más conocida. Las pruebas del consumidor se ejecutan contra un proveedor simulado de Pact y registran en un archivo de contrato las solicitudes realizadas y las respuestas esperadas; el pipeline del proveedor reproduce esas solicitudes contra el proveedor real y comprueba las respuestas. Un broker almacena los contratos y los resultados de verificación, de modo que, antes de desplegar, un pipeline puede preguntar si esta versión de un servicio es compatible con las versiones de sus consumidores y proveedores que ya se ejecutan en producción. Las pruebas de contrato no sustituyen a unas pocas pruebas de humo de extremo a extremo, y comprueban la estructura en lugar del significado de negocio, pero permiten desplegar los servicios de forma independiente con confianza, que es precisamente el objetivo de separarlos.

ATENCIÓN

Errores habituales en la comunicación entre servicios

- Construir largas cadenas síncronas para trabajo que el usuario no está esperando, de modo que un servicio lento arrastra toda la solicitud.
- Reintentar automáticamente llamadas no idempotentes y descubrir después pedidos duplicados y cobros dobles.
- Reintentar errores 4xx, o reintentar sin backoff ni jitter, lo que convierte una breve interrupción en una sobrecarga autoinfligida.
- Exponer tablas de la base de datos como recursos, de modo que cada cambio de esquema se convierte en un cambio de la API.
- Devolver colecciones sin límite y sin paginación, algo que funciona en las pruebas y falla en producción.
- Elegir GraphQL o gRPC por moda y pagar después por consultas N+1, límites al costo de las consultas o proxies para navegadores que el sistema no necesitaba.
- Hacer cambios incompatibles en una API interna porque "somos dueños de ambos lados", mientras un equipo de otra parte la estaba llamando sin decir nada.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué poner una cola entre dos servicios no elimina todo el acoplamiento entre ellos?

Elimina el acoplamiento temporal, pero ambos servicios siguen dependiendo del formato y del significado del mensaje, y del propio intermediario, así que un cambio en el mensaje todavía puede romper al receptor.

▼ ¿Por qué DELETE es idempotente aunque un segundo DELETE pueda devolver 404 en lugar de 204?

La idempotencia se refiere al efecto previsto en el servidor, no a la respuesta: tras uno o dos DELETE, el recurso ha desaparecido en cualquier caso.

▼ Un cliente móvil genera una nueva clave de idempotencia cada vez que reintenta. ¿Qué sale mal?

El servidor ve cada reintento como una operación nueva, así que una respuesta perdida seguida de un reintento todavía puede crear un pedido duplicado; la clave debe pertenecer a la operación lógica, no al intento HTTP.

▼ **¿Por qué la paginación por cursor puede mostrar una lista estable cuando llegan pedidos nuevos mientras el usuario pagina, y la paginación por offset puede no hacerlo?**

El cursor apunta justo después de un elemento concreto, así que las inserciones en otros puntos no desplazan la página siguiente; un offset cuenta filas, así que las filas nuevas empujan elementos de una página a otra y provocan duplicados u omisiones.

▼ **Inténtalo: Elige un estilo para cada caso: la app del cliente hace un pedido; dos servicios internos muy activos intercambian llamadas de precios; cinco servicios deben reaccionar cuando se hace un pedido. ¿Sería seguro un reintento del primero desde el dispositivo móvil? Da tus razones.**

Hacer un pedido: REST sobre HTTP, con un contrato público estable, pero POST no es idempotente, así que un reintento solo es seguro con una clave de idempotencia. Llamadas internas de precios: gRPC se adapta a las llamadas internas frecuentes, con código generado y mensajes compactos. Servicios que reaccionan: publica en un topic, para que cada suscriptor reciba su propia copia y el servicio de pedidos no necesite conocerlos. Una cola daría cada mensaje a un solo consumidor, lo que aquí es incorrecto.

FUENTES

- RFC 9110: HTTP Semantics (IETF, 2022): https://www.rfc-editor.org/rfc/rfc9110.html
- RFC 9457: Problem Details for HTTP APIs (IETF, 2023): https://www.rfc-editor.org/rfc/rfc9457.html
- Martin Fowler: Richardson Maturity Model: https://martinfowler.com/articles/richardsonMaturityModel.html
- gRPC documentation: Core concepts, architecture and lifecycle: https://grpc.io/docs/what-is-grpc/core-concepts/
- Ian Robinson: Consumer-Driven Contracts: A Service Evolution Pattern: https://martinfowler.com/articles/consumerDrivenContracts.html
- Apache Kafka 4.2.0 release announcement: https://kafka.apache.org/blog/2026/02/17/apache-kafka-4.2.0-release-announcement/
- RabbitMQ: Queues: https://www.rabbitmq.com/docs/queues
- RabbitMQ: Streams: https://www.rabbitmq.com/docs/streams
- RabbitMQ: Consumers: https://www.rabbitmq.com/docs/consumers
- Amazon SQS Developer Guide: queue types: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-queue-types.html
- Amazon SNS Developer Guide: fanout to Amazon SQS queues: https://docs.aws.amazon.com/sns/latest/dg/sns-sqs-as-subscriber.html

- Google Cloud Pub/Sub: Order messages: https://docs.cloud.google.com/pubsub/docs/ordering
- NATS: Core NATS: https://docs.nats.io/nats-concepts/core-nats
- NATS: JetStream: https://docs.nats.io/nats-concepts/jetstream
- RFC 9110: HTTP Semantics, section 15.3.3: https://www.rfc-editor.org/rfc/rfc9110.txt
- Microsoft Learn: Asynchronous Request-Reply pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/asynchronous-request-reply
- Stripe: Receive events in your webhook endpoint: https://docs.stripe.com/webhooks
- Standard Webhooks specification: https://raw.githubusercontent.com/standard-webhooks/standard-webhooks/main/spec/standard-webhooks.md

SECCIÓN 6.6 DE 6.8 · AVANZADO · 30 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Modelos de consistencia, CAP y PACELC

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.5 Almacenar datos
- 2.5 Disponibilidad y fiabilidad
- 6.1 Por qué distribuir

Términos anteriores que utiliza: ACID, Transacción, Disponibilidad, Latencia, Invariante

LA IDEA EN BREVE

Imagina una tienda con tres sucursales que tienen cada una una copia de la lista de existencias y se llaman por teléfono para comunicarse los cambios. Si la línea telefónica se corta y un cliente pregunta en una sucursal si queda la última bicicleta, esa sucursal tiene dos opciones: decir "no puedo estar seguro, espere, por favor", o responder con su propia lista y arriesgarse a vender una bicicleta que otra sucursal acaba de vender. Incluso cuando los teléfonos funcionan, llamar a las otras sucursales antes de cada respuesta hace esperar a los clientes. Los sistemas distribuidos se enfrentan exactamente a estas opciones, y la correcta depende de cuánto costaría una respuesta equivocada.

LO QUE NECESITAS SABER

- Un modelo de consistencia es el contrato que dice qué valores obsoletos o reordenados pueden observar los clientes; la C de CAP (copia única, linealizable) es distinta de la C de ACID (estado válido) y de los niveles de aislamiento.
- De más fuerte a más débil: linealizable (una copia, orden en tiempo real), secuencial (un orden acordado), causal (las causas antes que los efectos), garantías de sesión como leer tus propias escrituras y lecturas monótonas, y eventual (convergencia si cesan las escrituras).
- El teorema CAP (la conjetura de Brewer de 2000, demostrada por Gilbert y Lynch en 2002) dice que, durante una partición de red, un sistema replicado debe elegir, para cada solicitud, entre consistencia y disponibilidad; las particiones no son opcionales, así que "elige dos de tres" es una mala interpretación.
- El artículo de Brewer de 2012 recomienda detectar las particiones, funcionar en un modo de partición explícito y recuperarse después con fusiones y compensaciones, eligiendo de forma distinta para operaciones distintas.

- PACELC (Abadi, 2010 y 2012) añade que, en caso contrario (else), sin partición, los sistemas intercambian latencia por consistencia en cada solicitud.
- Con N réplicas, W confirmaciones de escritura y R respuestas de lectura, elegir R + W > N hace que los conjuntos de lectura y de escritura se solapen, pero los quórums por sí solos no son linealizables.
- Elige la consistencia según el caso de uso: fuerte para el dinero, los recursos únicos y los invariantes; garantías de sesión para los datos propios de un usuario; eventual para contadores, feeds y posiciones.

Qué significa "consistente" cuando los datos se copian

En cuanto un sistema guarda más de una copia de un dato, en réplicas de zonas distintas, en cachés o en otros servicios, las copias pueden discrepar durante un tiempo. Un modelo de consistencia es el contrato que dice qué discrepancias pueden observar los clientes. Es una promesa sobre lo que pueden devolver las lecturas dadas las escrituras que han ocurrido y, por tanto, una promesa sobre qué sorpresas debe manejar el código de la aplicación.

Cuidado con esta palabra tan cargada de significados. La C de ACID (Sección 0.5) significa que una transacción lleva la base de datos de un estado válido a otro, respetando reglas como "un saldo nunca baja de cero". La C de CAP significa algo distinto y más limitado: cada lectura se comporta como si hubiera una única copia actualizada. Los niveles de aislamiento de las bases de datos, como serializable o read committed, son una tercera familia, referida a las transacciones concurrentes; se tratan en transacciones y aislamiento. Esta sección trata de las garantías de los datos replicados.

Modelos de consistencia, del más fuerte al más débil

La linealizabilidad, definida por Maurice Herlihy y Jeannette Wing en 1990, es el modelo de un solo objeto más fuerte. Cada operación parece surtir efecto instantáneamente en algún momento entre su inicio y su fin, y todos los clientes ven el mismo orden, que respeta el tiempo real: una vez completada una escritura, toda lectura que empiece después la ve, o ve algo más reciente. El sistema se comporta como si hubiera una sola copia. Ejemplo de reparto de comida: dos clientes intentan canjear el último código de descuento; con un almacén linealizable, exactamente uno lo consigue y el otro ve que ya no está, sea cual sea la réplica a la que llegue.

La consistencia secuencial, definida por Leslie Lamport en 1979 para multiprocesadores, mantiene el orden global único y mantiene las operaciones de cada cliente en el orden en que ese cliente las emitió, pero elimina el requisito de tiempo real. Un cliente puede leer un valor

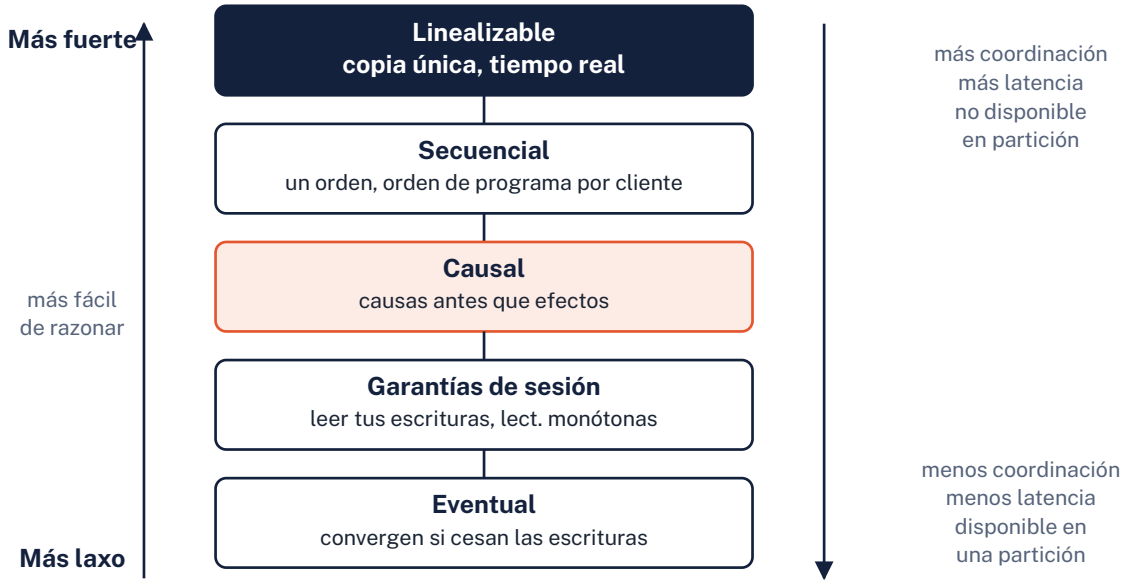
ligeramente desactualizado, siempre que todos estén de acuerdo en el orden de todas las escrituras. Las bases de datos distribuidas rara vez la ofrecen por su nombre, pero explica la diferencia entre "un orden acordado" y "un orden acordado que coincide con el reloj".

La consistencia causal solo exige que las operaciones relacionadas causalmente se vean en el mismo orden por parte de todos: si escribir B dependió de haber visto A, nadie puede ver B sin A. Las escrituras concurrentes y no relacionadas pueden verse en órdenes distintos por clientes distintos. Ejemplo: un restaurante responde "Sí, es sin gluten" a la pregunta de un cliente en el chat del pedido; nadie debería ver nunca la respuesta sin la pregunta. La consistencia causal resulta atractiva porque el mapa de modelos de consistencia de Jepsen, muy utilizado, la clasifica como alcanzable con disponibilidad *adherente (sticky)*: puede seguir funcionando durante una partición siempre que cada cliente permanezca con la misma réplica.

Las garantías de sesión, descritas por Douglas Terry y sus colegas en 1994, son promesas más débiles, por cliente, que eliminan las anomalías más confusas. Leer tus propias escrituras (read-your-writes): después de que un cliente cambie la dirección de entrega, la siguiente lectura de ese cliente muestra la nueva dirección, aunque otros usuarios la vean más tarde. Lecturas monótonas: una vez que un cliente ha visto el estado del pedido "en reparto", nunca ve después "en preparación" porque una lectura posterior llegó a una réplica retrasada. (Otras dos garantías, las escrituras monótonas y las escrituras que siguen a las lecturas, restringen el orden en que se aplican las escrituras de un cliente.)

La consistencia eventual, popularizada por Werner Vogels en 2008, es la garantía útil más débil: si no llegan escrituras nuevas, todas las réplicas acaban convergiendo en el mismo valor. No dice nada sobre cuánto tarda eso ni sobre qué puede leerse mientras tanto, y una implementación básica que resuelve las escrituras en conflicto conservando la de marca de tiempo más reciente, conocida como last write wins (gana la última escritura), descarta la otra en silencio. Las formas más fuertes, como la consistencia eventual fuerte con tipos de datos replicados libres de conflictos (CRDT), garantizan que las réplicas que han recibido las mismas actualizaciones tengan el mismo estado, usando tipos de datos cuyas actualizaciones concurrentes siempre se fusionan, como contadores y conjuntos.

La escalera de la consistencia



Cada peldaño hacia abajo elimina una garantía y algo de coordinación; las garantías de sesión son promesas por cliente que se sitúan entre la consistencia causal y la consistencia eventual simple.

SI ESTO ES NUEVO PARA TI

Si esto es nuevo para ti

Piensa en una lista de la compra familiar guardada en tres teléfonos que se sincronizan cuando tienen señal. Linealizable significa que la lista se comporta como si hubiera una sola hoja de papel en la nevera. Eventual significa que los teléfonos se ponen de acuerdo una vez que todos han tenido señal durante un rato, pero hasta entonces una persona puede comprar leche que otra ya ha comprado. Todo lo intermedio es un compromiso distinto entre "estar siempre de acuerdo" y "funcionar siempre, incluso en un túnel".

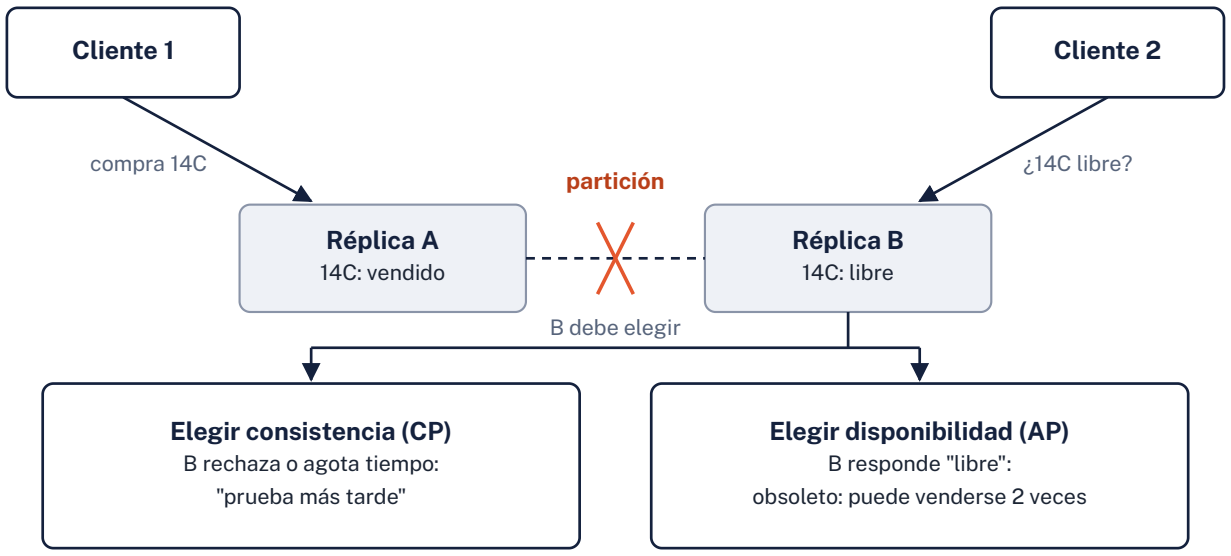
CAP: lo que dice realmente

Eric Brewer presentó la idea de CAP como una conjetura en su conferencia principal en el ACM Symposium on Principles of Distributed Computing (PODC) de 2000. En 2002, Seth Gilbert y Nancy Lynch publicaron una demostración formal, que la convirtió en un teorema para sus definiciones precisas. **Consistencia** significaba linealizabilidad. **Disponibilidad** significaba que toda solicitud recibida por un nodo que no ha fallado debe producir una respuesta. **Tolerancia a particiones** significaba que el sistema cumple sus promesas incluso cuando la red pierde un número arbitrario de mensajes entre nodos.

Leído con atención, el teorema CAP es limitado. Dice que, cuando se produce una partición de red, de modo que algunos nodos no pueden comunicarse con otros, un sistema replicado debe elegir para cada solicitud entre consistencia y disponibilidad. Un nodo aislado de las últimas escrituras puede negarse a responder (o esperar a que la partición se resuelva), preservando la consistencia, o responder con lo que tiene, preservando la disponibilidad a riesgo de devolver datos obsoletos o de aceptar escrituras en conflicto. El diagrama muestra el dilema en un sistema de venta de entradas que vende el asiento 14C.

Varias interpretaciones populares son erróneas. **"Elige dos de los tres"** sugiere que la tolerancia a particiones es opcional, pero en cualquier sistema que se comunica a través de una red las particiones ocurren lo elijas o no; la única elección real es cómo comportarse cuando ocurren. Un sistema llamado "CA" es uno que no ha decidido qué hacer cuando se produce una partición; eso solo es defendible donde las particiones son prácticamente imposibles, como en un único nodo. **"Nuestra base de datos es CP (o AP)"** suele ser impreciso: la disponibilidad de CAP es una propiedad muy fuerte que muchos sistemas comercializados como de alta disponibilidad no tienen, y muchos sistemas llamados CP tampoco son linealizables. Martin Kleppmann sostuvo en 2015 que las bases de datos no deberían etiquetarse en absoluto como CP o AP, porque la mayoría no encaja en ninguna de las dos definiciones y su comportamiento depende de la configuración. **"CAP explica nuestra latencia"** también es erróneo: CAP no dice nada sobre lo que ocurre cuando no hay partición, que es la mayor parte del tiempo.

La elección de CAP, en concreto



La réplica B no puede saber que el asiento 14C se acaba de vender en la réplica A, así que debe negarse a responder o arriesgarse a vender el asiento dos veces.

CAP doce años después

En 2012, en un artículo de IEEE Computer titulado "CAP Twelve Years Later: How the Rules Have Changed", el propio Brewer explicó que el planteamiento de dos de tres había confundido a la gente. Las particiones son poco frecuentes, así que un sistema puede ofrecer consistencia y disponibilidad casi todo el tiempo; la elección solo se aplica durante una partición, y solo a las operaciones afectadas. Tampoco hace falta que la elección sea global: operaciones distintas y datos distintos pueden elegir de forma distinta.

Brewer recomendó diseñar explícitamente para las particiones en tres pasos: **detectar** la partición, entrar en un **modo de partición** explícito en el que algunas operaciones se limitan (por ejemplo, aceptar pedidos pero aplazar los reembolsos) y ejecutar después un proceso de **recuperación** que fusione el estado y compense los errores cometidos durante la partición. También señaló que la C de CAP es la consistencia de copia única, un subconjunto más estricto de lo que protege la C de ACID, y que invariantes como "no vender nunca de más" pueden tener que repararse a posteriori. Este planteamiento conduce directamente a las compensaciones a nivel de negocio de las sagas: una aerolínea que vende más plazas de las que tiene un vuelo compensa al pasajero en lugar de rechazar todas las reservas durante un fallo de red.

Relojes vectoriales y relojes lógicos híbridos

Fusionar tras una partición exige saber qué escrituras entran realmente en conflicto, y los relojes de pared no pueden decirlo (Sección 6.1). Un **reloj vectorial** adjunta a cada versión una lista de pares de nodo y contador. Si todos los contadores de una son menores o iguales que los de otra, la primera versión es un antepasado y puede descartarse; si no, las dos son concurrentes y deben fusionarse. El artículo sobre Dynamo de Amazon, de 2007, lo usó para los carritos de compra: ninguna adición se pierde, aunque los artículos eliminados pueden reaparecer. Las listas crecen, así que Dynamo las recortaba. Un **reloj lógico híbrido** (Kulkarni y otros, 2014) es una sola marca de tiempo compacta: un tiempo cercano al reloj físico más un contador, de modo que un evento que condujo a otro siempre tiene el valor menor. Ordena los eventos, pero, a diferencia de un reloj vectorial, un valor menor no demuestra un vínculo causal. CockroachDB usa relojes lógicos híbridos.

PACELC: el compromiso que pagas cada día

Daniel Abadi señaló en una entrada de blog de 2010, y en un artículo de IEEE Computer de 2012, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story", que CAP ignora el compromiso al que se enfrentan realmente los diseñadores la mayor parte del tiempo. PACELC se lee así: si hay una **P**artición, elige entre disponibilidad (**A**vailability) y **C**onsistencia; en caso contrario (**E**lse), en funcionamiento normal, elige entre **L**atencia y **C**onsistencia.

La mitad del "else" es la aportación importante. Para que una lectura sea consistente, un sistema replicado debe coordinarse: enviar la escritura a otras réplicas y esperar las confirmaciones, o dirigir todas las lecturas a un líder, o contactar con varias réplicas y comparar. Cada una de esas opciones cuesta viajes de ida y vuelta por la red, y entre regiones un viaje de ida y vuelta puede costar decenas o incluso cientos de milisegundos. Un sistema que responde desde la réplica más cercana sin coordinarse es más rápido, pero puede devolver datos obsoletos. Ese compromiso se paga en cada solicitud, haya partición o no.

A menudo se citan clasificaciones PACELC, como PA/EL para los sistemas ajustados para la velocidad y la disponibilidad, y PC/EC para los sistemas que siempre prefieren la consistencia. Trátalas como descripciones de una configuración, no de un producto: la misma base de datos puede ser PA/EL en un despliegue y PC/EC en otro, según los ajustes elegidos por tabla, por consulta o por solicitud.

Consenso y coordinación: líderes, concesiones y bloqueos

El failover, los mapas de shards y las tareas programadas necesitan que varias máquinas se pongan de acuerdo en una respuesta, como qué nodo lidera ahora, aunque algunas se hayan caído o los mensajes lleguen tarde. Ese es el problema del **consenso**, y una decisión, una vez tomada, es definitiva. Paxos, de Leslie Lamport, es el algoritmo clásico. Raft (Ongaro y Ousterhout, 2014) ofrece la misma tolerancia a fallos y el mismo rendimiento, pero se diseñó para ser más fácil de entender. Un líder de Raft envía heartbeats (latidos); un seguidor que no recibe ninguno antes de que venza su timeout aleatorio inicia una elección en un nuevo *mandato* numerado, y una mayoría de votos lo convierte en líder. Cinco servidores siguen funcionando con dos caídos. Con más fallos, el clúster deja de avanzar, pero nunca devuelve un resultado incorrecto. Los clústeres suelen tener un número impar de miembros, porque la documentación de etcd señala que cuatro miembros no toleran más fallos que tres, y aconseja no pasar de siete, ya que cada escritura debe llegar a más máquinas.

Pocos equipos implementan esto por su cuenta. Usan **etcd**, que emplea Raft y es el almacén subyacente de todos los datos de los clústeres de Kubernetes, o **Apache ZooKeeper**, cuya receta de elección convierte en líder al candidato con el nodo temporal de número más bajo. Ambos guardan pequeñas cantidades de datos de coordinación, no registros de negocio.

Una **concesión** (lease) es un derecho con límite de tiempo: quien la tiene debe renovarla continuamente, y si muere, la concesión vence y otro nodo toma el relevo. Kubernetes usa concesiones para los heartbeats de los nodos y la elección de líder. El peligro es un titular que se detiene, quizá por la recolección de basura, y que luego despierta sin saber que su concesión ha vencido y escribe de todos modos; comprobarlo antes no ayuda, porque la pausa

puede caer entre la comprobación y la escritura. Un **token de cercado** (fencing token) resuelve esto: un número que aumenta con cada concesión y se envía con cada escritura, de modo que el almacenamiento rechace cualquier token inferior a uno que ya haya visto.

Los **bloqueos distribuidos** plantean la misma cuestión. Martin Kleppmann distingue los bloqueos para la eficiencia, en los que un fallo supone trabajo duplicado, como un correo electrónico repetido, de los bloqueos para la corrección, en los que un fallo corrompe datos. El algoritmo Redlock de Redis obtiene un bloqueo en una mayoría de maestros de Redis independientes. Kleppmann argumentó en 2016 que depende de suposiciones de temporización y que no emite tokens de cercado, por lo que no debería proteger la corrección. Salvatore Sanfilippo, creador de Redis, respondió que es seguro bajo su modelo declarado, aunque aceptó el argumento sobre los relojes monótonos. La documentación de Redis enlaza a ambos; el debate no está resuelto, así que léelos. Los almacenes de consenso necesitan el mismo cuidado: la prueba de Jepsen sobre etcd 3.4.3 halló que dos clientes podían tener un mismo bloqueo, y aconsejó el cercado. La lección práctica es que el recurso protegido debe defenderse por sí mismo, con una comprobación de cercado o una restricción de la base de datos. Si Despacho ejecuta un solo planificador de asignaciones, una doble ejecución ocasional no debe causar ningún daño.

Quórums: ajustar la consistencia con aritmética

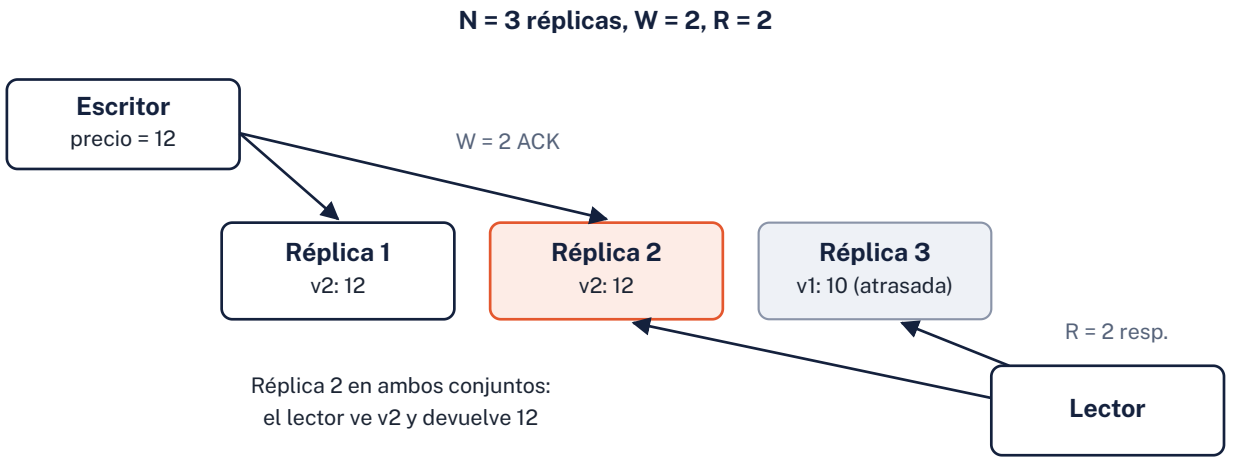
Los almacenes replicados sin líder, al estilo del artículo de Dynamo de Amazon (2007), permiten que cada operación elija cuántas réplicas intervienen. Con *N* réplicas de un elemento, una escritura se confirma cuando *W* réplicas la han almacenado, y una lectura recoge respuestas de *R* réplicas y devuelve la versión más reciente entre ellas. Si R + W > N, todo conjunto de lectura se solapa con todo conjunto de escritura en al menos una réplica, así que una lectura incluirá al menos una copia de la última escritura confirmada.

La elección de *R* y *W* intercambia consistencia, latencia y disponibilidad. Valores más bajos significan esperar a menos réplicas, y más rápidas, y tolerar más fallos, a cambio de posibles lecturas obsoletas. Valores más altos significan lo contrario. De todos modos, las escrituras suelen enviarse a las *N* réplicas; *W* solo decide cuántas confirmaciones espera el cliente.

El solapamiento del quórum es necesario para obtener lecturas actualizadas, pero no es, por sí solo, linealizabilidad. Las escrituras concurrentes pueden ordenarse con marcas de tiempo de relojes que no coinciden, de modo que una escritura posterior puede perder con last-write-wins. Una escritura que falló después de llegar a algunas réplicas puede seguir siendo visible en ellas. Los quórums "laxos" (sloppy), que aceptan escrituras en nodos sustitutos durante los fallos, rompen el solapamiento. Y dos lecturas que se solapan con una escritura todavía en curso pueden ver el valor nuevo y luego el antiguo, a menos que el lector repare las réplicas retrasadas antes de devolver el resultado. Los almacenes que necesitan una verdadera

semántica de comparar e intercambiar (compare-and-set) añaden un protocolo de consenso, como las transacciones ligeras basadas en Paxos de Apache Cassandra, solo para esas operaciones.

Por qué funciona R + W > N



La escritura la confirmaron las réplicas 1 y 2; la lectura consulta las réplicas 2 y 3; el solapamiento en la réplica 2 significa que la versión más reciente está en el conjunto de lectura.

EJEMPLO PRÁCTICO

Ejemplo práctico: elegir N, W y R para un almacén de precios de la carta

La plataforma de reparto de comida guarda los precios de la carta en un almacén sin líder con N = 3 réplicas, una por zona de disponibilidad. Cada réplica responde en unos 2 ms dentro de su zona y en unos 5 ms desde otra zona. El cliente de quórum espera la W-ésima (o R-ésima) respuesta más rápida.

W = 2, R = 2 (R + W = 4 > 3)	Solapamiento garantizado. Las lecturas y las escrituras esperan cada una a la segunda réplica más rápida (unos 5 ms). Ambas siguen funcionando con una zona caída. Una opción por defecto equilibrada y habitual.
W = 3, R = 1 (4 > 3)	Solapamiento garantizado y lecturas rápidas (la réplica más cercana, unos 2 ms), pero todas las escrituras fallan si cualquier réplica está caída: la disponibilidad de escritura es el producto de tres disponibilidades. Adecuado para datos que se escriben rara vez y se leen constantemente, si las escrituras fallidas pueden reintentarse más tarde.
W = 1, R = 3 (4 > 3)	Las escrituras son rápidas y sobreviven a dos fallos; cada lectura debe llegar a las tres réplicas, así que las lecturas fallan si cualquier zona es inaccesible. Rara vez es un buen intercambio.
W = 1, R = 1 (2, no > 3)	Lo más rápido y disponible, pero una lectura puede no ver en absoluto la última escritura. Aceptable para un contador de popularidad; no para un precio que se va a cobrar al cliente.
N = 5, W = 3, R = 3 (6 > 5)	Solapamiento garantizado tolerando dos réplicas caídas tanto para lecturas como para escrituras, a costa de más almacenamiento y de quórums algo más lentos.

Regla general: con R + W > N, las escrituras toleran N − W fallos y las lecturas toleran N − R fallos.

El solapamiento da lecturas actualizadas solo para las escrituras confirmadas y solo con un versionado fiable; consulta las salvedades anteriores antes de confiar en los quórums para invariantes como los niveles de existencias.

Para los precios, la plataforma eligió W = 2, R = 2, y cobró a los clientes con el precio guardado en el pedido durante el proceso de pago, de modo que una lectura obsoleta de la carta nunca podía cambiar lo que se cobraba.

Cómo se pueden configurar bases de datos conocidas

Los productos reales rara vez se sitúan en un solo punto de la escalera. Exponen ajustes, y la descripción correcta es "cómo está configurado este despliegue". Siguen cuatro ejemplos habituales; consulta la documentación actual de la versión que ejecutas, porque los valores por defecto cambian de una versión a otra.

Consistencia configurable en cuatro bases de datos muy utilizadas (comportamiento documentado en el momento de escribir esto)

Base de datos	Comportamiento típico por defecto	Ajustes que refuerzan la consistencia	Qué vigilar
PostgreSQL con replicación en streaming	Un único primario recibe todas las escrituras; la replicación a los servidores en espera (standbys) es asíncrona por defecto, así que las lecturas desde un standby pueden estar obsoletas.	`synchronous_standby_names` con `synchronous_commit` (por ejemplo, `remote_apply`) hace que una confirmación espere hasta que los standbys la tengan; lee desde el primario para leer tus propias escrituras.	La replicación síncrona añade latencia a las confirmaciones, y las escrituras se detienen si los standbys requeridos son inaccesibles.
Conjunto de réplicas de MongoDB	En la mayoría de los despliegues, las escrituras requieren por defecto la confirmación de una mayoría; las lecturas desde el primario usan por defecto el read concern `local`, que puede devolver datos que más tarde se revierten.	Read concern `majority` o `linearizable`; sesiones causalmente consistentes cuando tanto las lecturas como las escrituras usan majority.	Leer desde los secundarios (read preference) intercambia actualidad de los datos por latencia y carga.
Apache Cassandra	El nivel de consistencia se elige por solicitud (ONE, QUORUM, LOCAL_QUORUM, ALL y otros); los conflictos se resuelven con marcas de tiempo según last-write-wins.	QUORUM (o LOCAL_QUORUM dentro de un centro de datos) tanto para lecturas como para escrituras, de modo que R + W > factor de replicación; transacciones ligeras para comparar e intercambiar (compare-and-set).	El desfase de los relojes afecta a last-write-wins; LOCAL_QUORUM no da ninguna garantía entre centros de datos.
Amazon DynamoDB	Las lecturas son eventualmente consistentes por defecto; las tablas globales se replican entre regiones de forma asíncrona por defecto.	`ConsistentRead=true` en tablas e índices secundarios locales; tablas globales con consistencia fuerte multirregión (disponible de forma general desde 2025).	Las lecturas fuertemente consistentes no están disponibles en los índices secundarios globales, consumen el doble de capacidad de lectura y pueden ser más lentas.

Elegir la consistencia según el caso de uso

El hábito más útil es dejar de preguntar "¿es consistente nuestro sistema?" y preguntar, para cada dato y cada operación, *qué sale mal si un cliente ve un valor obsoleto o reordenado, y cuánto cuesta*. El movimiento de dinero, los recursos únicos y los invariantes suelen justificar la consistencia fuerte y su latencia. Los contadores, los feeds, las recomendaciones y la analítica normalmente no. Muchas anomalías que ve el usuario desaparecen solo con las garantías de sesión, que son baratas: dirige las lecturas de un usuario al primario durante unos segundos después de que ese usuario escriba, o lleva un token de versión en la sesión y espera hasta que la réplica se haya puesto al día con él.

Necesidades de consistencia por caso de uso

Caso de uso	Costo de una lectura obsoleta o reordenada	Garantía razonable	Mecanismo típico
Saldo bancario antes de una retirada	Descubierto por encima del límite; dinero perdido	Linealizable (y transacciones serializables)	Un único líder para la cuenta, replicación síncrona o consenso
Último asiento o entrada al abrirse la venta	Vender un asiento dos veces; reembolsos y clientes enfadados	Linealizable por asiento	Escritura condicional (compare-and-set) sobre el registro del asiento
Nombre de usuario único al registrarse	Dos cuentas con el mismo nombre	Linealizable en la comprobación de unicidad	Restricción de unicidad en una sola base de datos, o un registro respaldado por consenso
Dirección de entrega del propio cliente después de editarla	La comida se envía a la dirección antigua	Leer tus propias escrituras (read-your-writes)	Leer desde el líder después de una escritura, o un token de versión en la sesión
Estado del pedido que se muestra al cliente	El estado parece retroceder; confusión y llamadas al servicio de atención	Lecturas monótonas más leer tus propias escrituras	Enrutamiento fijo (sticky) a una réplica, o tokens de versión
Chat del restaurante con el cliente	Una respuesta aparece antes que su pregunta	Causal	Almacén causalmente consistente, u ordenación por conversación
Carrito de la compra	Reaparece un artículo eliminado; algo molesto	Eventual, con fusión	Aceptar siempre los añadidos; fusionar versiones (el ejemplo del artículo de Dynamo)
Número de "me gusta" o valoración del restaurante	El recuento se desvía en unos pocos durante unos segundos	Eventual	Replicación asíncrona, contadores o CRDT
Ubicación del repartidor en el mapa	El punto va unos segundos por detrás	Eventual, gana el último valor	Escrituras de baja latencia en la réplica más cercana; las posiciones antiguas se descartan

ATENCIÓN

Errores comunes

- Decir "elegimos AP" para todo un sistema, en lugar de decidir por operación y por dato.
- Suponer que una base de datos replicada es linealizable porque su publicidad dice "fuertemente consistente", sin leer qué cubre la garantía (¿claves individuales?, ¿una sola región?, ¿índices?).
- Añadir réplicas de lectura o cachés a un sistema y olvidar que todas las rutas de lectura que ve el usuario acaban de volverse eventualmente consistentes.
- Confiar en last-write-wins con relojes que no están sincronizados y perder actualizaciones en silencio.
- Usar quórums con R + W > N y creer que eso hace seguras las reservas de existencias sin un compare-and-set.
- Tratar la consistencia eventual como un detalle técnico, cuando es visible para los usuarios y requiere decisiones de producto sobre lo que ven mientras los datos convergen.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Cuál es la diferencia entre la linealizabilidad y la consistencia secuencial?**

Ambas exigen un único orden de operaciones en el que todos los clientes estén de acuerdo y que respete el orden propio de cada cliente; la linealizabilidad exige además que ese orden respete el tiempo real, de modo que una lectura que empieza después de completarse una escritura debe verla.

▼ **¿Por qué una base de datos distribuida "CA" no es una elección con sentido?**

Porque las particiones de red ocurren en cualquier sistema que se comunica a través de una red; un sistema que supone que nunca se producen simplemente no ha decidido qué hará cuando se produzcan.

▼ **Con N = 5 y W = 2, ¿cuál es el R más pequeño que garantiza que los conjuntos de lectura y de escritura se solapen?**

R = 4, porque R + W debe ser mayor que 5.

▼ **Un usuario edita su perfil y la página se recarga mostrando la versión antigua, aunque no importa que otros usuarios la vean con retraso. ¿Qué garantía falta y cuál es una solución barata?**

Leer tus propias escrituras; dirige las lecturas de ese usuario al líder durante un breve tiempo después de una escritura, o lleva un token de versión y espera a una réplica que se haya puesto al día.

▼ **Inténtalo: Un almacén de precios de asientos usa N = 5, W = 2, R = 3, y un colega dice que las lecturas siempre se solapan con la última escritura. Compruébalo y propón una solución. ¿Usarías solo quórums para evitar que dos fans compren un mismo asiento?**

R + W = 5, que no es mayor que N, de modo que una lectura puede perderse la última escritura. Fija W = 3 y R = 3, lo que da 6 > 5; ambas siguen funcionando con dos réplicas caídas, con una latencia algo mayor. Elige según el uso: unos precios desactualizados pueden tolerarse brevemente. Los quórums por sí solos no son linealizables, así que no protegerán un asiento único. Vender un asiento requiere consistencia fuerte, como una transacción con un solo líder o una restricción de unicidad.

FUENTES

- Eric Brewer: CAP Twelve Years Later: How the Rules Have Changed (InfoQ reprint of the IEEE Computer article): https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/
- Daniel Abadi: Problems with CAP, and Yahoo's little known NoSQL system (2010, origin of PACELC): https://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html
- Jepsen: Consistency models: https://jepsen.io/consistency/models
- Martin Kleppmann: Please stop calling databases CP or AP (2015): https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html
- Apache Cassandra documentation: Dynamo (tunable consistency and quorums): https://cassandra.apache.org/doc/latest/cassandra/architecture/dynamo.html
- Raft consensus algorithm site: https://raft.github.io/
- Ongaro and Ousterhout: In Search of an Understandable Consensus Algorithm (2014): https://raft.github.io/raft.pdf
- etcd FAQ: https://etcd.io/docs/v3.5/faq/
- Kubernetes: Operating etcd clusters: https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/
- Kubernetes: Leases: https://kubernetes.io/docs/concepts/architecture/leases/
- Apache ZooKeeper: Recipes and solutions: https://zookeeper.apache.org/doc/current/recipes.html

- Martin Kleppmann: How to do distributed locking (2016): https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html
- Salvatore Sanfilippo: Is Redlock safe?: https://antirez.com/news/101
- Redis: Distributed locks: https://redis.io/docs/latest/develop/clients/patterns/distributed-locks/
- Jepsen: etcd 3.4.3: https://jepsen.io/analyses/etcd-3.4.3
- DeCandia and others: Dynamo: Amazon's Highly Available Key-value Store (2007): https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- Kulkarni and others: Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases: https://cse.buffalo.edu/tech-reports/2014-04.pdf
- CockroachDB: Transaction layer: https://docs.cockroachlabs.com/docs/stable/architecture/transaction-layer

SECCIÓN 6.7 DE 6.8 · AVANZADO · 30 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Transacciones distribuidas, sagas y el patrón de bandeja de salida (outbox)

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.5 Almacenar datos
- 6.3 Microservicios
- 6.4 Dirigida por eventos

Términos anteriores que utiliza: ACID, Base de datos por servicio, Coreografía, Orquestación, Entrega al menos una vez (at-least-once)

LA IDEA EN BREVE

Reservar unas vacaciones significa reservar un vuelo, un hotel y un automóvil de alquiler a tres empresas distintas. Nadie puede hacer que las tres cosas ocurran exactamente en el mismo instante. Por eso las reservas una a una y, si resulta que el automóvil de alquiler no está disponible, cancelas el hotel y el vuelo, quizá pagando una penalización por cancelación. Eso es una saga: una serie de pasos separados, cada uno con una forma prevista de deshacerlo. El software que abarca varios servicios funciona igual, y la habilidad está en elegir el orden de los pasos y planificar las cancelaciones antes de que algo salga mal.

LO QUE NECESITAS SABER

- La confirmación en dos fases (two-phase commit) proporciona atomicidad entre participantes mediante las fases de preparación y de confirmación, pero se bloquea mientras el coordinador está caído y mantiene bloqueos a través de la red; XA (X/Open, 1991) la estandariza.
- Entre servicios con propietarios independientes, el 2PC multiplica la falta de disponibilidad, mantiene bloqueos remotos, acopla los despliegues y no lo admiten la mayoría de las API ni muchos brokers, por lo que en general se evita.
- Una saga (Garcia-Molina y Salem, 1987) es una secuencia de transacciones locales, cada una con una transacción compensatoria; o se completan todos los pasos o los ya completados se compensan en orden inverso.
- Las compensaciones son nuevas transacciones hacia adelante, no rollbacks: es posible que se hayan visto estados intermedios, y las compensaciones deben ser idempotentes, reintentables y capaces de llegar fuera de orden.

- Las sagas carecen de aislamiento; las contramedidas descritas por Chris Richardson incluyen los bloqueos semánticos, las actualizaciones conmutativas, la vista pesimista, releer el valor, los archivos de versiones y la elección por valor.
- La bandeja de salida transaccional (transactional outbox) escribe el cambio de negocio y su mensaje en una sola transacción local, y un retransmisor (por sondeo o mediante captura de datos de cambios, como Debezium) lo publica al menos una vez; los receptores idempotentes con una bandeja de entrada (inbox) eliminan los duplicados.
- Prefiere una sola transacción local siempre que sea posible, rediseña los límites cuando la división sea accidental y usa sagas cuando los pasos puedan compensarse o reintentarse.

El problema: una acción de negocio, varias bases de datos

Dentro de una sola base de datos, una transacción hace atómicos varios cambios: la fila del pedido, sus artículos y la referencia del pago se guardan todos o ninguno (Sección 0.5). En cuanto los datos se reparten entre servicios que tienen cada uno su propia base de datos, como recomiendan los microservicios y la base de datos por servicio, una sola acción de negocio, como hacer un pedido de comida, afecta a varias bases de datos que pertenecen a servicios distintos: la del servicio de pedidos, la del servicio de pagos y la del servicio de despacho. Ninguna transacción local puede abarcarlas todas.

Hay dos familias de respuestas. Una es hacer que muchos gestores de recursos actúen como uno solo mediante un protocolo de confirmación atómica distribuida, del que la confirmación en dos fases es el clásico. La otra es aceptar que la acción es una secuencia de transacciones locales y diseñar para los momentos intermedios: la saga. Una tercera respuesta, a menudo la mejor, es darse cuenta de que el límite de la transacción y el límite del servicio no coinciden y mover uno de ellos. Esta sección trata las tres y termina con una guía para elegir.

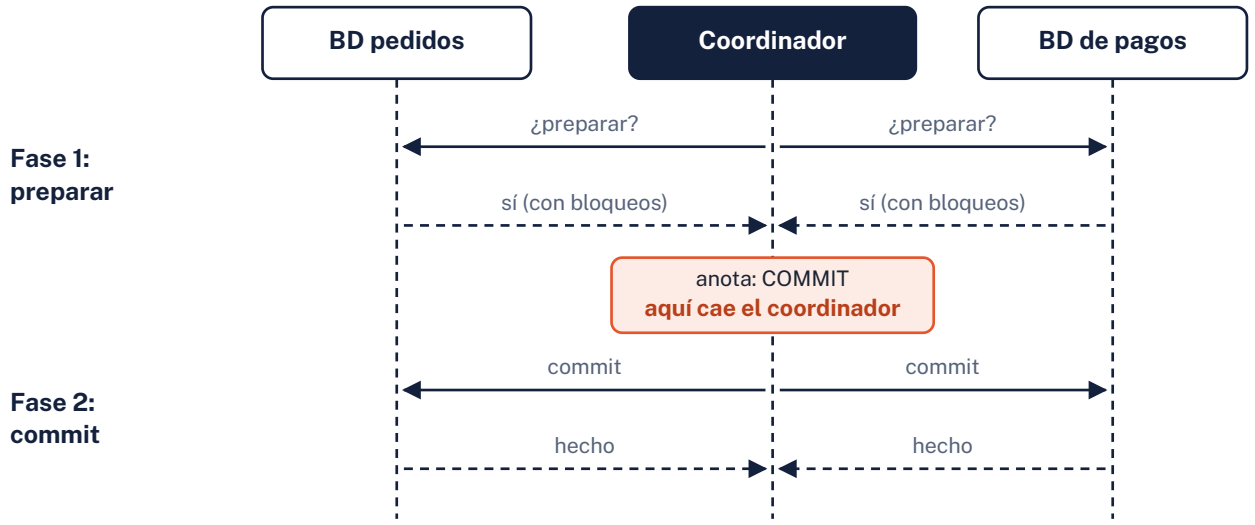
La confirmación en dos fases y XA

La confirmación en dos fases (2PC) es un protocolo, de finales de la década de 1970, que proporciona atomicidad de todo o nada entre varios participantes, como dos bases de datos o una base de datos y un intermediario de mensajes (message broker). Lo ejecuta un **coordinador** (el gestor de transacciones). En la **fase 1, preparación**, el coordinador pregunta a cada participante si puede confirmar. Un participante que responde que sí debe primero hacer duraderos los cambios y mantener los bloqueos correspondientes, porque al votar que sí renuncia al derecho de abortar por su cuenta. Si todos los votos son afirmativos, el coordinador escribe su decisión, COMMIT, en su propio registro; si algún voto es negativo o un participante no responde, la decisión es ABORT. En la **fase 2, confirmación**, comunica el resultado a cada participante, que lo aplica y libera sus bloqueos.

El punto débil es que el 2PC es un protocolo **bloqueante**. Un participante que ha votado que sí queda *en duda*: no conoce el resultado y no puede adivinarlo con seguridad, porque puede que otro participante ya haya confirmado. Si el coordinador cae en ese momento y no se recupera, los participantes en duda conservan sus bloqueos hasta que se recupere, y las demás transacciones que tocan esas filas también esperan. Los operadores pueden forzar una decisión *heurística* para liberar los bloqueos, a riesgo de romper la atomicidad.

La especificación XA, publicada por X/Open en 1991, estandarizó la interfaz entre un gestor de transacciones y los gestores de recursos para que, por ejemplo, un servidor de aplicaciones Java pueda confirmar de forma atómica una actualización de base de datos y el envío de un mensaje. Muchas bases de datos relacionales y algunos brokers de mensajes la admiten.

La confirmación en dos fases y dónde se bloquea



Si el coordinador falla tras los votos, quienes votaron que sí no pueden decidir solos y conservan sus bloqueos

Los participantes que han votado que sí mantienen sus bloqueos hasta conocer la decisión; si el coordinador falla en el punto resaltado, quedan atascados.

Por qué se evita el 2PC entre microservicios

El 2PC funciona y todavía se usa dentro de los límites de un único sistema. Entre servicios con propietarios independientes suele encajar mal, por razones que se agravan entre sí.

La disponibilidad se multiplica a la baja. Todos los participantes y el coordinador deben estar activos para que cualquier transacción se confirme, de modo que la disponibilidad de la transacción es el producto de las suyas, la misma aritmética que la de una cadena de llamadas síncronas (consulta por qué distribuir). **Los bloqueos se mantienen a través de la red** durante al menos dos viajes de ida y vuelta, y mucho más si algo va lento, lo que reduce el rendimiento (throughput) y aumenta la contención. **El acoplamiento vuelve**: todos los participantes deben

hablar el mismo protocolo, compartir un gestor de transacciones y ponerse de acuerdo en los timeouts, lo que socava la desplegabilidad independiente que justificó separar los servicios. **Muchos participantes no pueden unirse**: la mayoría de las API HTTP, los servicios de terceros como los proveedores de pagos, muchas bases de datos NoSQL y varios brokers populares no admiten XA. Y la API del proveedor de pagos nunca mantendrá un bloqueo para tu coordinador.

El panorama es distinto dentro de las bases de datos distribuidas. Sistemas como Google Spanner ejecutan internamente la confirmación en dos fases entre grupos de réplicas que a su vez se mantienen consistentes mediante consenso, de modo que un coordinador caído se sustituye en lugar de esperarlo. Eso hace viables las transacciones entre particiones dentro de una misma base de datos; no hace viables las transacciones entre servicios con propietarios distintos y con tecnologías diferentes.

Sagas: una secuencia de transacciones locales

Hector Garcia-Molina y Kenneth Salem introdujeron la saga en un artículo de SIGMOD de 1987 sobre las *transacciones de larga duración*, aquellas que mantendrían bloqueos de base de datos durante horas. Su idea era dividir una transacción así en una secuencia de transacciones ordinarias T_1, T_2 ... T_n, cada una de las cuales se confirma por su cuenta, y emparejar cada una con una transacción compensatoria C_i que la deshace semánticamente. El sistema garantiza entonces que, o bien se completa toda la secuencia, o bien los pasos completados van seguidos de sus compensaciones en orden inverso: T_1 ... T_j, C_j ... C_1.

Las arquitecturas de microservicios adoptaron la idea porque encaja exactamente con sus restricciones: cada paso es una transacción ACID local en la base de datos propia de un servicio, y los pasos se enlazan mediante mensajes. Una saga proporciona atomicidad en el sentido eventual, de negocio (todos los pasos ocurren o sus efectos se deshacen), pero no aislamiento: entre un paso y otro, otras transacciones pueden ver el estado intermedio.

Ayuda clasificar los pasos, como hace Chris Richardson en *Microservices Patterns* (2018). Las **transacciones compensables** pueden deshacerse mediante una compensación. La **transacción pivote** es el punto de decisión definitivo: una vez que se confirma, la saga se ejecutará hasta el final. Las **transacciones reintentables** van después del pivote y deben diseñarse para que acaben teniendo éxito si se reintentan. Situar los pasos con más probabilidad de fallar antes del pivote, y los pasos que no pueden deshacerse después de él, es una decisión de diseño clave.

Coreografía u orquestación

Los pasos de una saga deben coordinarse de algún modo, y las dos opciones reflejan la coreografía y la orquestación tratadas en arquitectura dirigida por eventos.

En una **saga basada en coreografía** no hay coordinador. Cada servicio escucha eventos y reacciona realizando su transacción local y publicando un evento nuevo: el servicio de pedidos publica `OrderCreated`, el servicio de pagos reacciona y publica `PaymentAuthorised`, el servicio de despacho reacciona a ese evento, y así sucesivamente; los eventos de fallo desencadenan compensaciones en los servicios que los escuchan. Es sencilla para dos o tres pasos y mantiene los servicios débilmente acoplados. A medida que crecen los pasos, el flujo solo existe de forma implícita, disperso entre los servicios, por lo que es difícil verlo, probarlo y cambiarlo; y se cuelan dependencias cíclicas de eventos entre servicios.

En una saga basada en orquestación, un único orquestador, normalmente en el servicio que es propietario del proceso de negocio, envía comandos a los participantes y reacciona a sus respuestas. El orquestador es una máquina de estados persistente: registra en su propia base de datos en qué paso está, de modo que puede reanudarse tras una caída. El flujo es explícito y se puede probar en un solo lugar, y los participantes no necesitan conocerse entre sí. El riesgo es que el orquestador acumule lógica de negocio que corresponde a los participantes. Los motores de flujos de trabajo como Temporal, Camunda y AWS Step Functions ofrecen orquestación duradera como plataforma, a costa de una dependencia más. Para más de unos tres pasos, o para cualquier flujo con ramificaciones y timeouts, la mayoría de los profesionales prefiere la orquestación.

Ejecución duradera: Temporal y Step Functions

La **ejecución duradera** significa que un flujo de trabajo, una vez iniciado, llega hasta el final a pesar de las caídas. En Temporal, un flujo de trabajo es código normal; lo que ocurre se almacena en un historial de eventos y, tras una caída, otro worker vuelve a ejecutar el código sobre ese historial para reconstruir su estado y continuar. Por eso, el código del flujo de trabajo debe ser determinista, y el trabajo externo va en *actividades*, que pueden reintentarse y, por tanto, deben tolerar la repetición. AWS Step Functions, en cambio, define una máquina de estados en Amazon States Language; los flujos Standard pueden ejecutarse hasta un año y los Express, cinco minutos. Así, la espera de una saga de pedido a que un restaurante acepte puede ser un simple temporizador. Los costos son una nueva dependencia crítica y el cuidado necesario al cambiar el código de flujos de trabajo aún en ejecución. Consulta los límites actuales.

Ejemplo práctico: la saga del pedido

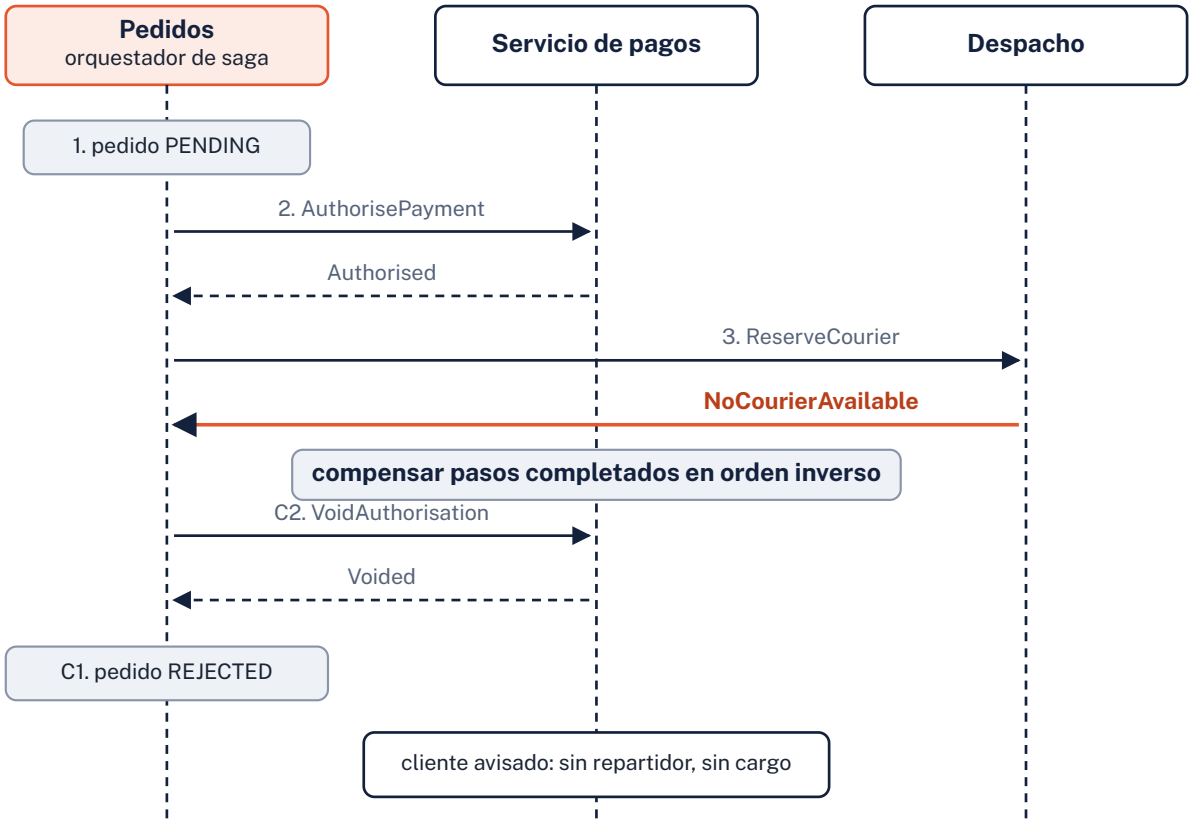
La saga del pedido de la plataforma de reparto de comida la orquesta el servicio de pedidos. El orden de los pasos es deliberado: la autorización del pago es una retención barata que es fácil de anular, la reserva del repartidor es el paso con más probabilidad de fallar en las horas de

mayor actividad, y la confirmación al restaurante es el pivote, porque una vez que una cocina ha empezado a cocinar, el pedido no puede deshacerse sin más.

La saga del pedido, paso a paso

Paso	Servicio y transacción local	Clase	Compensación si falla un paso posterior
1. Crear el pedido	El servicio de pedidos inserta el pedido con el estado `PENDING` (un bloqueo semántico).	Compensable	C1: cambiar el estado a `REJECTED` y registrar el motivo. La fila se conserva para auditoría, no se borra.
2. Autorizar el pago	El servicio de pagos pide al proveedor de la tarjeta que aplique una retención por el total, usando el ID del pedido como clave de idempotencia.	Compensable	C2: anular la autorización, liberando la retención. Puede que el cliente vea brevemente un importe pendiente en la app de su banco.
3. Reservar un repartidor	El servicio de despacho reserva un repartidor cercano para la franja de recogida.	Compensable	C3: liberar al repartidor para que pueda aceptar otros trabajos.
4. Confirmar el pedido	El servicio de pedidos cambia el estado a `CONFIRMED` y envía la comanda al restaurante.	Pivote	Ninguna: a partir de aquí la saga debe seguir adelante. Los problemas posteriores se gestionan con nuevos procesos de negocio (reembolsos, vales).
5. Avisar al cliente; capturar el pago en la recogida	El servicio de notificaciones envía "Pedido confirmado"; el servicio de pagos captura el importe retenido cuando el repartidor recoge la comida.	Reintentable	Ninguna: estos pasos se reintentan, de forma idempotente, hasta que tienen éxito o se escalan a una persona.

La saga del pedido cuando no hay ningún repartidor disponible



El paso 3 falla, así que el orquestador ejecuta las compensaciones de los pasos 2 y 1, del más reciente al más antiguo, y se avisa al cliente antes de que se haya cocinado o cobrado nada.

La compensación no es un rollback

Un rollback de base de datos hace que sea como si la transacción nunca hubiera ocurrido: nadie más llegó a ver sus cambios. Una compensación es una transacción nueva, hacia adelante, que contrarresta semánticamente otra anterior que *sí* ocurrió y que puede haberse visto. En la saga del pedido, puede que el banco del cliente ya muestre una retención pendiente y que un correo electrónico haya dicho "pago autorizado". La compensación no puede deshacer el envío del correo; solo puede anular la retención y enviar otro mensaje. Algunas acciones no pueden compensarse en absoluto, como la comida ya cocinada o una notificación ya leída; esas van después del pivote, o necesitan una compensación de negocio, como un vale o un reembolso.

Por eso las compensaciones necesitan el mismo cuidado que los pasos hacia adelante. Deben ser **idempotentes**, porque el mensaje que las desencadena puede entregarse dos veces. Deben ser **reintentables hasta que tengan éxito**, porque una compensación que falla de forma permanente deja el sistema en un estado que nadie diseñó; si de verdad no pueden tener éxito, deben lanzar una alerta para que una persona lo resuelva. Puede que tengan que ser **conmutativas con el paso hacia adelante**: si un `VoidAuthorisation` llega antes que el

`AuthorisePayment` que deshace (algo posible tras timeouts y reintentos), el servicio de pagos debe recordar la anulación y rechazar la autorización tardía en lugar de fallar. Y deben ser visibles para el negocio: "rechazado, sin repartidor" es información que el equipo de operaciones quiere ver en un panel.

Vivir sin aislamiento: anomalías y contramedidas

Como cada paso se confirma por su cuenta, las sagas carecen de la I de ACID, y las sagas concurrentes pueden interferir entre sí. Richardson describe tres anomalías resultantes. Una **actualización perdida** se produce cuando una saga sobrescribe un cambio hecho por otra sin leerlo, como cuando una saga de cancelación marca un pedido como cancelado y la saga del pedido lo marca después como confirmado. Una **lectura sucia** se produce cuando una saga lee datos que otra saga ha cambiado pero que todavía puede compensar, como una comprobación del límite de crédito que cuenta una retención que está a punto de anularse. Una **lectura difusa o no repetible** se produce cuando dos pasos de una misma saga leen los mismos datos y obtienen valores distintos porque otra saga los cambió entretanto.

También cataloga contramedidas, técnicas de diseño que reducen las anomalías, basándose en investigaciones anteriores sobre transacciones entre varias bases de datos. Las más utilizadas son estas.

Contramedidas para la falta de aislamiento en las sagas (según Richardson)

Contramedida	Idea	Ejemplo de reparto de comida
Bloqueo semántico	Un paso compensable establece un indicador a nivel de aplicación, normalmente un estado `*_PENDING`, que avisa a las demás transacciones de que el registro está en cambio; estas esperan, fallan o lo tratan de forma especial.	Mientras un pedido está en `PENDING`, una solicitud de cancelación se pone en cola o se rechaza con "el pedido se está procesando", en lugar de aplicarse a medio camino.
Actualizaciones conmutativas	Diseñar las actualizaciones para que den el mismo resultado en cualquier orden, lo que elimina las actualizaciones perdidas.	El libro de ganancias del repartidor registra los abonos y los cargos como entradas separadas en lugar de sobrescribir un saldo.
Vista pesimista	Reordenar los pasos para minimizar el riesgo de negocio de una lectura sucia, colocando las actualizaciones arriesgadas en pasos reintentables después del pivote.	Aumentar la "capacidad disponible" de un restaurante solo en un paso reintentable, cuando la cancelación sea segura, nunca en un paso que pueda compensarse.
Releer el valor	Antes de actualizar, volver a leer el registro y comprobar que no ha cambiado desde que se leyó (bloqueo optimista); si ha cambiado, abortar y, posiblemente, reiniciar la saga.	Antes de confirmar, el orquestador comprueba que la versión del pedido sigue siendo la que creó; si una cancelación la ha cambiado, compensa en su lugar.
Archivo de versiones	Registrar las operaciones aplicadas a un registro para que las operaciones que llegan fuera de orden puedan reordenarse o reconocerse.	El servicio de pagos registra que ha llegado una anulación para una autorización que todavía no ha visto, y rechaza la autorización tardía.
Por valor	Elegir el mecanismo de coordinación para cada solicitud según su riesgo de negocio.	Los pedidos normales usan la saga; un pedido de catering corporativo de gran valor podría pasar por un proceso más estricto y más lento.

SI ESTO ES NUEVO PARA TI

Paso a paso: el momento entre guardar y enviar

La idea que requiere más atención aquí es el problema de la escritura dual y cómo lo resuelve la bandeja de salida transaccional. Cada paso de la saga guarda un cambio en su propia base de datos y envía un mensaje para que pueda empezar el paso siguiente. El problema está en el momento entre las dos cosas.

- El servicio de pedidos guarda el pedido 1042 en su base de datos con el estado PENDING. A continuación debería enviar un mensaje OrderCreated al broker, para que el servicio de pagos pueda empezar.
- Supón que el servidor se cae después de guardar y antes de enviar. El pedido existe, pero el mensaje nunca salió, así que el servicio de pagos nunca se entera y el pedido 1042 espera para siempre.
- Hacerlo al revés no es mejor: si el servicio envía primero y luego falla el guardado, el servicio de pagos empieza a trabajar en un pedido que no existe.
- Escribir así en dos lugares, en pasos separados, se llama escritura dual: siempre hay un momento en el que una cosa ha ocurrido y la otra no.
- La bandeja de salida transaccional (transactional outbox) elimina ese momento. El servicio guarda el pedido y el mensaje saliente, como una fila en una tabla de bandeja de salida, en una sola transacción de base de datos, de modo que se guardan ambos o ninguno. Después, un programa independiente, el retransmisor (relay), lee la bandeja de salida y envía cada mensaje al broker.
- Si el retransmisor se cae después de enviar un mensaje pero antes de anotar que lo hizo, vuelve a enviar ese mensaje, así que cada receptor debe saber manejar los duplicados, como explica el final de esta sección.
- **La lección:** sin la confirmación en dos fases, que esta sección desaconseja, una base de datos y un broker no pueden actualizarse como si fueran uno. En su lugar, guarda el mensaje junto con el cambio, envíalo después y acepta algún duplicado ocasional.

Publicar de forma fiable: la bandeja de salida transaccional

Cada paso de una saga hace dos cosas: actualiza su propia base de datos y envía un mensaje (un evento o una respuesta). Hacerlas por separado crea el problema de la escritura dual. Si el servicio confirma y luego se cae antes de publicar, el resto de la saga nunca se entera del paso. Si publica primero y luego falla la confirmación, otros servicios actúan sobre algo que nunca ocurrió. Envolver la base de datos y el broker en un 2PC es justo lo que queríamos evitar.

La bandeja de salida transaccional lo resuelve con una sola transacción local. El servicio escribe el cambio de negocio y el mensaje saliente, como una fila en una tabla `outbox`, en la misma transacción de base de datos; o se confirman ambos o ninguno. Después, un **retransmisor** (relay) independiente lee la bandeja de salida y publica cada mensaje en el broker. Hay dos diseños de retransmisor habituales. Un **publicador por sondeo** (polling publisher) consulta cada cierto tiempo la tabla de bandeja de salida en busca de filas no publicadas y las marca como enviadas: es sencillo, pero añade carga y latencia. El **seguimiento del registro de transacciones** (transaction log tailing) usa la captura de datos de cambios (CDC): una herramienta lee el propio registro de replicación de la base de datos (el write-ahead log de PostgreSQL o el binary log de MySQL) y convierte las nuevas filas de la bandeja de salida en mensajes. Debezium, una plataforma de CDC de código abierto que suele ejecutarse con Kafka Connect, es el ejemplo más conocido e incluye un enrutador de eventos de bandeja de salida que dirige cada fila a un topic según una columna, como el tipo de agregado.

El retransmisor puede caerse después de publicar y antes de registrar que lo hizo, así que a veces publicará un mensaje dos veces. Por tanto, la bandeja de salida proporciona una entrega al menos una vez, y los consumidores deben saber manejar los duplicados. El orden de publicación se conserva por agregado si el retransmisor publica las filas en el orden de confirmación y usa el ID del agregado como clave del mensaje.

```
BEGIN;

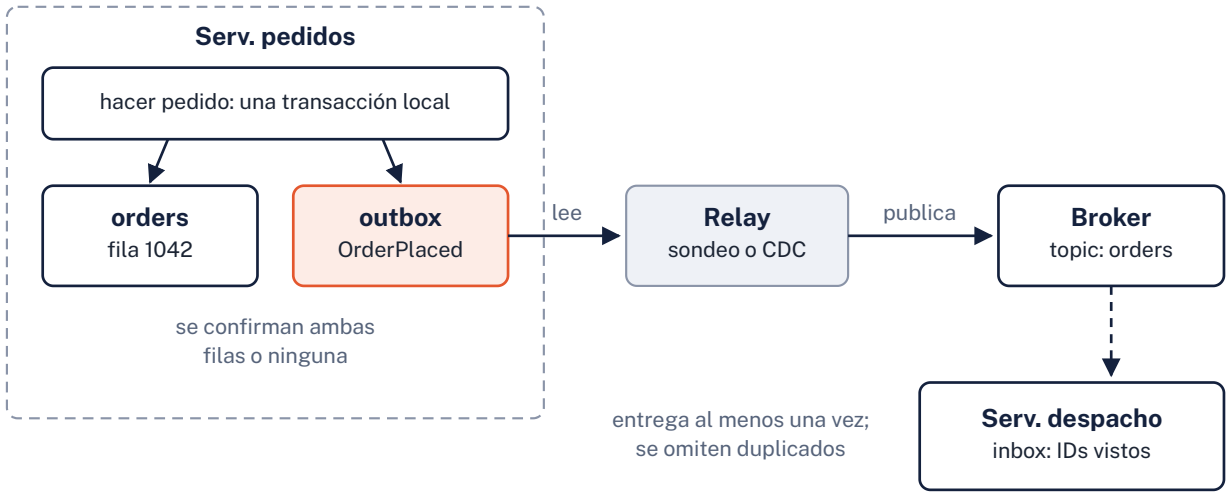
INSERT INTO orders (id, customer_id, state, total_pence)
VALUES ('1042', 'c-77', 'PENDING', 2350);

INSERT INTO outbox (id, aggregate_type, aggregate_id, type, payload)
VALUES ('e-9f1c', 'order', '1042', 'OrderCreated',
        '{"orderId":"1042","customerId":"c-77","totalPence":2350}');

COMMIT;
-- A relay (poller or CDC such as Debezium) later publishes e-9f1c to the broker.
```

Hacer un pedido con una bandeja de salida transaccional: el pedido y su evento se confirman juntos, en una sola transacción local.

Bandeja de salida al salir, bandeja de entrada al entrar



La bandeja de salida elimina la escritura dual en el lado emisor; la bandeja de entrada, o cualquier comprobación de duplicados, hace seguro el lado receptor con una entrega al menos una vez.

Recibir de forma fiable: receptores idempotentes y la bandeja de entrada

Como la bandeja de salida, los reintentos del broker y los reintentos de la saga producen duplicados, cada participante debe ser un receptor idempotente (idempotent receiver), el nombre que Gregor Hohpe y Bobby Woolf dieron al patrón en *Enterprise Integration Patterns* (2003). Algunas operaciones son idempotentes por naturaleza ("cambiar el pedido 1042 a CONFIRMED"); otras necesitan una comprobación de duplicados. La implementación habitual es una **bandeja de entrada** (inbox): una tabla de ID de mensajes procesados, escrita en la misma transacción local que el cambio de negocio, de modo que "procesar el mensaje" y "recordar que se procesó" se confirman juntos. Un mensaje que se vuelve a entregar encuentra su ID ya presente y se confirma su recepción sin hacer nada. El código del lado del consumidor aparece en arquitectura dirigida por eventos.

Juntos, la bandeja de salida, la entrega al menos una vez y los receptores idempotentes dan el resultado práctico al que la gente suele referirse con "exactamente una vez": cada efecto de negocio ocurre una vez, aunque los mensajes puedan viajar más de una vez.

GUÍA DE DECISIÓN

Guía de decisión: ¿una transacción, una saga o nuevos límites?

Úsala cuando una acción de negocio necesite cambiar datos en más de un lugar.

Q1 **¿Todos los datos que deben cambiar juntos pertenecen a un solo servicio y se almacenan en una sola base de datos?**

- **Sí** Usa una sola transacción ACID local. Es la opción más sencilla y más sólida; no la distribuyas.
- **No** Pasa a la siguiente pregunta.

Q2 **¿Las partes que deben cambiar juntas cambian casi siempre juntas, pertenecen a un solo equipo y comparten un contexto delimitado, de modo que la división es accidental?**

- **Sí** Rediseña el límite: fusiona los servicios o mueve los datos, para que la acción pase a ser una sola transacción local. Las sagas no arreglan una división equivocada.
- **No** Pasa a la siguiente pregunta.

Q3 **¿Todos los participantes deben ver todos los cambios en el mismo instante, sin que nunca sea visible ningún estado intermedio (por ejemplo, un libro contable que siempre debe cuadrar)?**

- **Sí** Mantén esos registros en una sola base de datos, o usa una base de datos distribuida que ofrezca internamente transacciones entre particiones. Usa el 2PC solo dentro de los límites de un único sistema, con participantes que controles.
- **No** Pasa a la siguiente pregunta.

Q4 **¿Cada paso puede compensarse, o bien situarse después de un pivote y reintentarse hasta que tenga éxito?**

- **Sí** Usa una saga: orquestada si tiene más de unos tres pasos o alguna ramificación, coreografiada si es corta y estable. Añade una bandeja de salida, receptores idempotentes y contramedidas como los bloqueos semánticos.
- **No** Reordena los pasos o cambia el proceso de negocio (por ejemplo, reservar antes de cobrar, o añadir una revisión manual) hasta que sea posible; si no, reconsidera los límites.

ATENCIÓN

Errores comunes

- Publicar un evento después de confirmar, sin bandeja de salida, y perder eventos cada vez que un servicio se reinicia en el momento equivocado.
- Escribir las compensaciones como "borrar la fila", lo que destruye la pista de auditoría y confunde a cualquiera que ya la hubiera visto.
- Olvidar que las compensaciones pueden fallar o llegar antes que el paso que deshacen.
- Colocar un paso irreversible, como enviar el pedido a la cocina o cobrar la tarjeta, antes de un paso que falla a menudo.
- Usar la coreografía para un flujo largo y con ramificaciones hasta que nadie sabe decir qué pasa cuando falla el paso 5.
- Usar sagas para tapar una división de servicios que atraviesa un único límite de consistencia; el arreglo más barato suele ser mover el límite.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué un participante en una confirmación en dos fases no puede simplemente confirmar por su cuenta si el coordinador desaparece después de que haya votado que sí?

Porque no conoce la decisión: puede que otro participante haya votado que no, así que el coordinador puede haber decidido abortar; adivinar podría romper la atomicidad.

▼ En la saga del pedido, ¿por qué la confirmación del pedido al restaurante es el pivote y no el primer paso?

Una vez que la cocina empieza a cocinar, el paso no puede deshacerse, así que debe ir después de los pasos que podrían fallar (la autorización del pago, la reserva del repartidor), que pueden compensarse con poco costo.

▼ ¿Qué problema resuelve la bandeja de salida transaccional y qué garantía ofrece?

Elimina la escritura dual entre la base de datos y el broker al confirmar juntos el cambio y el mensaje; después, el retransmisor entrega el mensaje al menos una vez, así que puede haber duplicados.

▼ **Indica dos diferencias entre una compensación y un rollback de base de datos.**

Es una transacción nueva que se ejecuta después de que la original se haya confirmado, y puede que otros ya hayan visto la original o actuado en consecuencia; además, puede ser solo una forma semántica de deshacer (un reembolso, una anulación) en lugar de restaurar exactamente el estado anterior.

▼ **Inténtalo: Una saga de entradas tiene estos pasos: autorizar el pago, retener los asientos, emitir la entrada, enviar un correo al fan. En el paso dos resulta que ya no quedan asientos. ¿Qué ocurre? ¿Qué paso es el pivote y por qué una compensación no es un rollback?**

La saga compensa en orden inverso: anula la autorización de pago, y se le dice al fan que los asientos ya no están disponibles. Emitir la entrada es el pivote, ya que, una vez que existe una entrada, no puede simplemente desaparecer, y los pasos posteriores, como el correo electrónico, deben reintentarse hasta que tengan éxito. Una compensación es una nueva transacción hacia adelante, no un rollback: otros pueden haber visto la retención, así que debe ser idempotente y segura de reintentar.

FUENTES

- Garcia-Molina and Salem: Sagas (Princeton technical report, 1987): https://www.cs.princeton.edu/research/techreps/598
- Chris Richardson: Pattern: Saga (microservices.io): https://microservices.io/patterns/data/saga.html
- Chris Richardson: Pattern: Transactional outbox (microservices.io): https://microservices.io/patterns/data/transactional-outbox.html
- Microsoft Azure Architecture Center: Saga design pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/saga
- Debezium documentation: Outbox event router: https://debezium.io/documentation/reference/transformations/outbox-event-router.html
- Temporal: Understanding Temporal: https://docs.temporal.io/evaluate/understanding-temporal
- Temporal: Workflows: https://docs.temporal.io/workflows
- AWS Step Functions: Standard and Express workflows: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html

SECCIÓN 6.8 DE 6.8 · INTERMEDIO · 28 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Arquitecturas serverless y basadas en espacios

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.7 Servidores y la nube
- 6.1 Por qué distribuir
- 6.4 Dirigida por eventos

Términos anteriores que utiliza: Servicio gestionado, Idempotente, Fallo parcial, Consistencia eventual, Réplica de lectura

LA IDEA EN BREVE

Piensa en dos formas de atender a los clientes. Una parada de taxis solo cobra por cada viaje: cuando nadie necesita un trayecto no pagas nada, pero en una noche de mucho movimiento cada viaje cuesta más de lo que costaría tener un automóvil propio, y el primer taxi puede tardar un momento en llegar. Eso es serverless. Ahora imagina la taquilla de un estadio antes de un gran concierto: en lugar de que cada empleado haga cola para consultar un único registro central, cada ventanilla guarda su propia copia de qué asientos están libres y envía sus ventas a la oficina central más tarde. Eso es la arquitectura basada en espacios, construida para la avalancha.

LO QUE NECESITAS SABER

- Serverless combina las funciones como servicio, donde el proveedor ejecuta código activado por eventos y lo escala desde cero, con el backend como servicio, donde servicios gestionados sustituyen a componentes del lado del servidor; AWS Lambda (anunciado en 2014) popularizó FaaS.
- Las funciones no tienen estado y duran poco, se facturan por invocación y por GB-segundo, están sujetas a arranques en frío (cold starts) y a límites estrictos de tiempo, memoria y carga útil, y a menudo se activan al menos una vez, por lo que los manejadores deben ser idempotentes.
- Todo el estado duradero vive en servicios gestionados, así que un sistema serverless es un sistema distribuido por construcción y está fuertemente acoplado a su proveedor, salvo que la lógica de negocio se mantenga detrás de puertos.

- El precio por solicitud es más barato que la capacidad siempre activa por debajo de un volumen de cruce y más caro por encima de él; el punto de cruce depende de la duración, la memoria, la forma del tráfico y los costos de personal.
- La arquitectura basada en espacios atiende las solicitudes desde unidades de procesamiento con grids de datos en memoria replicados, gestionadas por un middleware virtualizado, y actualiza la base de datos de forma asíncrona mediante bombas de datos y escritores de datos, con lectores de datos para precargar el grid.
- Su nombre procede del espacio de tuplas de Linda, de Gelernter (1985); sus principales riesgos son las colisiones de datos entre réplicas, los límites de memoria, el retraso de la base de datos, el costo y la complejidad.
- Usa la arquitectura basada en espacios solo para cargas extremas y con picos que la caché, las réplicas y las salas de espera no puedan manejar, y enruta o particiona los datos disputados para que no puedan colisionar.

Dos respuestas a la cuestión de la capacidad

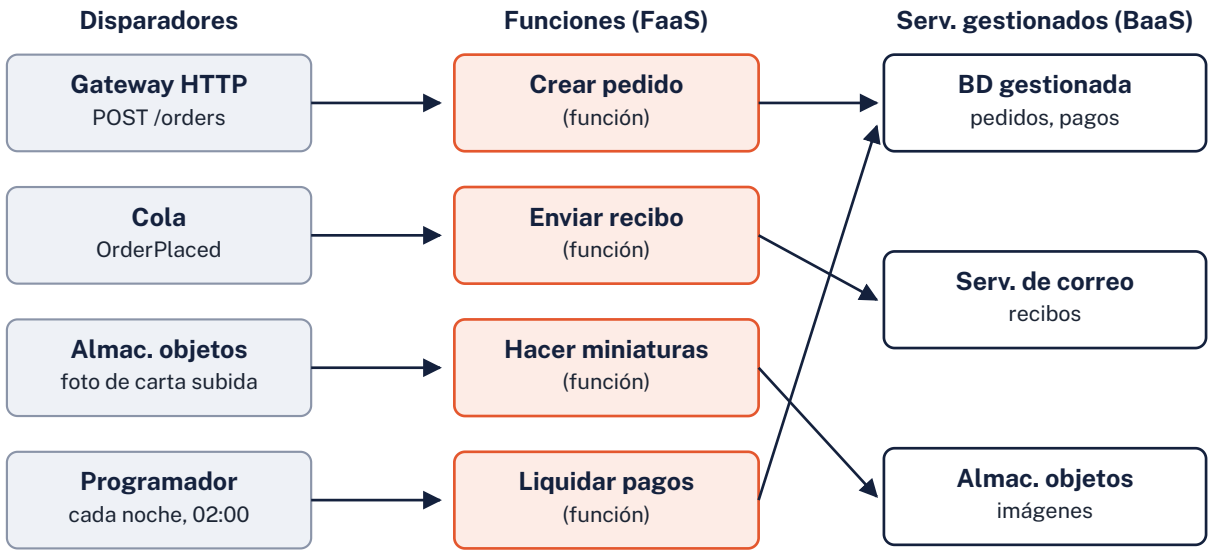
Todos los estilos de este capítulo vistos hasta ahora suponen que alguien ejecuta servidores y los dimensiona para la carga. Esta sección trata dos estilos que responden a la cuestión de la capacidad de formas opuestas. La arquitectura serverless entrega por completo los servidores a un proveedor de nube y paga por uso, lo que conviene al trabajo pequeño, dirigido por eventos y a menudo inactivo. La arquitectura basada en espacios mantiene en memoria todo lo que necesita una solicitud con mucha actividad, repartido entre muchas unidades idénticas, para que la base de datos deje de estar en la ruta crítica, lo que conviene a picos extremos y repentinos, como la puesta a la venta de entradas. Ambas son herramientas especializadas; ambas se eligen a menudo por motivos equivocados.

Serverless: funciones y backends gestionados

Serverless no significa que no haya servidores; significa que el equipo no aprovisiona, parchea ni dimensiona servidores, ni paga por ellos cuando están inactivos. El artículo de Mike Roberts de 2018, muy citado, distingue dos ingredientes. Las funciones como servicio (FaaS) ejecutan pequeñas piezas del código propio del equipo en respuesta a eventos, en contenedores que el proveedor crea, escala y destruye, y que pueden durar una sola invocación. AWS Lambda, anunciado en noviembre de 2014 y disponible de forma general en abril de 2015, popularizó el modelo; Azure Functions, Google Cloud Run functions y Cloudflare Workers son otros ejemplos. El backend como servicio (BaaS) sustituye componentes del lado del servidor por servicios de terceros totalmente gestionados a los que el cliente o la función llaman directamente: autenticación (Auth0, Amazon Cognito), bases de datos (Firebase, DynamoDB), almacenamiento de objetos, correo electrónico y colas.

Una aplicación serverless suele ser una combinación: las funciones contienen la lógica de negocio específica de la organización, y todo lo demás, incluido todo el estado duradero, vive en servicios gestionados. El proveedor se encarga de la capacidad, la disponibilidad entre zonas y el sistema operativo. El equipo sigue siendo responsable de su código, sus datos, su configuración y sus ajustes de seguridad, como explica el modelo de responsabilidad compartida en los servidores y la nube.

Los eventos activan funciones; el estado vive en otro lugar



Cada instancia de función no tiene estado y dura poco; todo el estado duradero vive en servicios gestionados

En una plataforma de reparto de comida, cuatro tipos de evento activan cuatro funciones, que leen y escriben en servicios gestionados en lugar de guardar nada por sí mismas.

Cómo se comportan las funciones: las características que condicionan el diseño

Disparadores de eventos. Las funciones se ejecutan en respuesta a eventos: una solicitud HTTP a través de un gateway, un mensaje en una cola o un flujo, un archivo que llega al almacenamiento de objetos, un cambio en una base de datos o una programación horaria. Los disparadores que invocan la función de forma asíncrona, como las colas y las notificaciones de almacenamiento, suelen entregar *al menos una vez* y reintentar en caso de fallo, por lo que los manejadores deben ser idempotentes, exactamente igual que en la arquitectura dirigida por eventos.

Escalar a cero y escalar horizontalmente. Sin tráfico, no se ejecuta ninguna instancia y no se factura nada de cómputo. A medida que llegan eventos, la plataforma arranca más instancias en paralelo, hasta un límite de concurrencia de la cuenta o de la función; en AWS Lambda, el límite regional por defecto es de 1000 ejecuciones concurrentes en el momento de escribir

esto, ampliable a petición. El escalado horizontal rápido es a la vez una ventaja y un peligro: mil funciones concurrentes pueden abrir mil conexiones de base de datos y desbordar una base de datos relacional diseñada para unos pocos cientos, y por eso existen los proxies de conexiones y los topes de concurrencia por función.

Facturación por invocación. El costo suele ser un pequeño cargo por solicitud más un cargo por la memoria multiplicada por el tiempo de ejecución (gigabytes-segundo). Por tanto, el código que espera sin hacer nada a una API lenta de la que depende está pagando por esperar.

Arranques en frío (cold starts). La primera solicitud a una instancia nueva debe esperar mientras la plataforma arranca un contenedor, carga el entorno de ejecución y ejecuta el código de inicialización: desde decenas de milisegundos hasta varios segundos, según el lenguaje, el tamaño del paquete y la configuración de red. Las instancias se reutilizan para solicitudes posteriores mientras siguen calientes, así que los arranques en frío afectan sobre todo al tráfico poco frecuente y a las ráfagas repentinas. Las mitigaciones incluyen paquetes de despliegue más pequeños, entornos de ejecución más ligeros, inicializar fuera del manejador y pagar por mantener instancias preinicializadas (AWS lo llama provisioned concurrency), lo que reintroduce en parte el costo de tener capacidad siempre activa.

Límites. Las funciones se ejecutan con límites estrictos de tiempo de ejecución, memoria, tamaño de la carga útil y disco temporal; para AWS Lambda, en el momento de escribir esto, 15 minutos, de 128 MB a 10 240 MB de memoria (con CPU asignada en proporción) y 6 MB para una solicitud o respuesta síncrona. Los trabajos largos deben dividirse en pasos, y las cargas útiles grandes deben pasarse por referencia al almacenamiento de objetos.

Ausencia de estado. Nada de lo que hay en la memoria o el disco local de una instancia tiene garantizado sobrevivir hasta la siguiente invocación. Guardar en caché en memoria entre llamadas es una optimización que puede acertar o no, nunca un lugar para datos importantes.

SI ESTO ES NUEVO PARA TI

Paso a paso: una función que olvida

El punto en el que más gente tropieza aquí es que una función serverless no puede contar con conservar nada de una ejecución a la siguiente, así que cada dato importante debe vivir en otro lugar. Un pequeño error muestra por qué.

- El equipo de reparto de comida escribe una función que se ejecuta cada vez que se hace un pedido. Para mostrar un recuento en directo de los pedidos del día, un desarrollador guarda un contador en la memoria de la función y le suma uno cada vez.
- En una mañana tranquila parece funcionar: una copia de la función gestiona los pedidos uno tras otro, y el recuento va 1, 2, 3.
- A la hora de comer los pedidos se disparan, y la plataforma arranca 50 copias de la función a la vez, cada una con su propia memoria. Cada copia lleva su propio recuento por separado, así que ninguna conoce el total real.
- Cuando el tráfico baja, la plataforma elimina discretamente las copias que ya no necesita, y sus recuentos desaparecen con ellas. El primer pedido de mañana puede empezar otra vez desde 1, o desde lo que una copia superviviente recuerde por casualidad.
- La solución es guardar el recuento en un servicio gestionado que compartan todas las copias, como una base de datos o una caché, y usar el propio comando de 'sumar uno' de ese servicio, para que dos copias que suman a la vez no puedan sobrescribirse entre sí.
- **La lección:** escribe cada función para que sea sin estado. Trata todo lo que haya en su memoria como un borrador que puede desaparecer en cualquier momento, y decide dónde vive cada dato antes de escribir la función, que es de lo que trata la siguiente parte de esta sección.

Adónde va el estado

Como las funciones no guardan nada, el trabajo de arquitectura pasa a consistir en elegir dónde vive cada tipo de estado. Los datos de negocio duraderos van a una base de datos gestionada, a menudo una que maneje bien muchas conexiones de corta duración (almacenes clave-valor o documentales, o una base de datos relacional detrás de un proxy de conexiones). Los objetos grandes van al almacenamiento de objetos y se pasan entre funciones mediante su clave. El estado compartido de corta duración, como los contadores de limitación de tasa, va a una caché gestionada. El progreso de un proceso de varios pasos va a un servicio de flujos de trabajo (AWS Step Functions, Azure Durable Functions o similares), que actúa como orquestador de sagas duradero, como se describe en sagas. Las sesiones de usuario van en tokens que guarda el cliente o en un almacén de sesiones.

La consecuencia es que un sistema serverless es un sistema distribuido por construcción, con todas las inquietudes de este capítulo: latencia de red entre cada función y sus datos, fallos parciales, eventos duplicados, consistencia eventual y la necesidad de trazado distribuido. Una sola acción de negocio puede atravesar un gateway, tres funciones, una cola y dos bases de datos. La observabilidad (consulta observabilidad) no es opcional.

Acoplamiento al proveedor: el precio de no ejecutar servidores

Serverless acopla un sistema a su proveedor más estrechamente que alquilar máquinas virtuales o ejecutar contenedores. Los formatos de los eventos, la configuración de los disparadores, los modelos de identidad y permisos, las definiciones de flujos de trabajo y los propios servicios gestionados son todos específicos del proveedor. Trasladar una aplicación serverless madura a otra nube se parece más a una reescritura que a una migración.

Los profesionales no se ponen de acuerdo sobre cuánto importa esto. Una postura sostiene que la dependencia del proveedor (lock-in) es el precio de una gran reducción del trabajo operativo y que la mayoría de las organizaciones nunca cambian de nube, así que optimizar para la portabilidad es un esfuerzo perdido. La otra sostiene que los cambios de precios, las exigencias regulatorias o las adquisiciones sí obligan a cambiar, y que un acoplamiento profundo elimina poder de negociación. Un camino intermedio es aceptar el acoplamiento en los bordes y contenerlo en el núcleo: mantener la lógica de negocio en módulos simples detrás de puertos, con manejadores de función finos que actúan como adaptadores (el enfoque de la arquitectura hexagonal), mantener la infraestructura como código y registrar el acoplamiento como una decisión explícita. El costo y la dependencia del proveedor se tratan con más detalle en FinOps y dependencia del proveedor.

EJEMPLO PRÁCTICO

Ejemplo práctico: ¿cuándo sale más barato lo siempre activo?

La API de cartas de restaurantes de la plataforma de reparto de comida podría ejecutarse como funciones o en dos máquinas virtuales siempre activas. **Todos los precios son cifras redondas ilustrativas elegidas para facilitar la aritmética, no precios de lista actuales de ningún proveedor.** Cada solicitud se ejecuta durante una media de 200 ms con 0.5 GB de memoria, así que consume 0.1 GB-segundos.

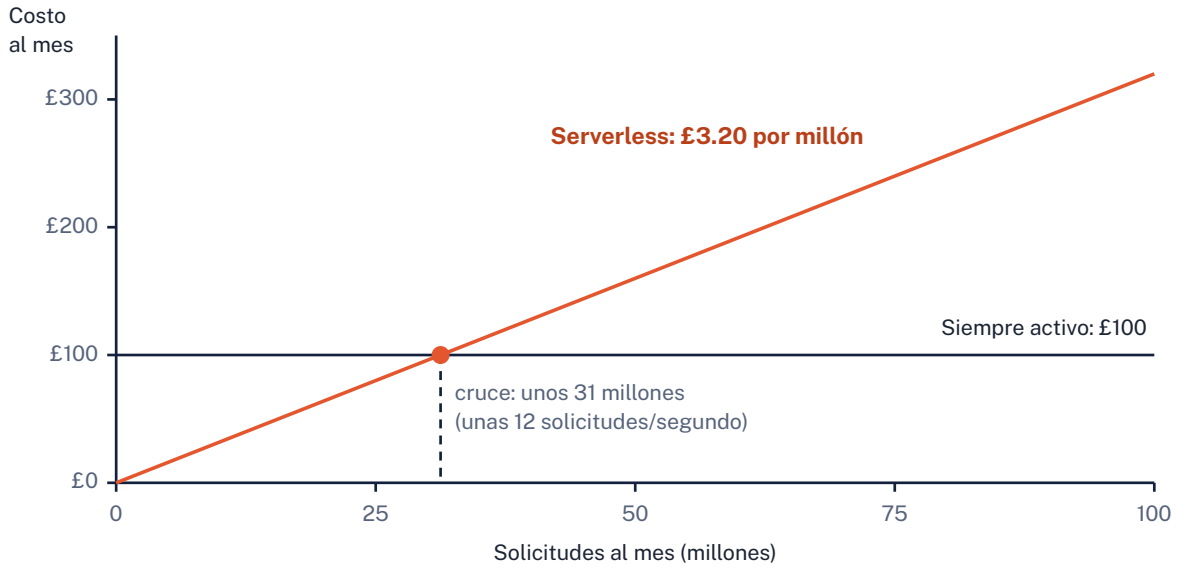
Precios serverless ilustrativos	Cómputo: £0.00002 por GB-segundo; £0.20 por millón de invocaciones; £1.00 por millón de solicitudes a través del gateway HTTP.
Costo serverless por millón de solicitudes	Cómputo: 1 000 000 × 0.1 GB-s × £0.00002 = £2.00. Invocaciones: £0.20. Gateway: £1.00. Total: £3.20 por millón.
Costo ilustrativo de lo siempre activo	Dos VM pequeñas a £40 al mes cada una (dos por redundancia) más un balanceador de carga a £20 = £100 al mes, dimensionado para picos de unas 300 solicitudes por segundo.
1 millón de solicitudes al mes (0.4 por segundo de media)	Serverless: £3.20; siempre activo: £100. Serverless es unas 30 veces más barato.
10 millones al mes (unas 4 por segundo)	Serverless: £32; siempre activo: £100.
Punto de cruce	£100 ÷ £3.20 ≈ 31 millones de solicitudes al mes, unas 12 solicitudes por segundo de media (31.25 millones ÷ 2.59 millones de segundos en 30 días).
100 millones al mes (unas 39 por segundo)	Serverless: £320; siempre activo: todavía £100, porque las dos VM aún tienen margen de capacidad.
Efecto de reducir la duración a la mitad, a 100 ms	El cómputo baja a £1.00, el total a £2.20 por millón, y el punto de cruce se desplaza a unos 45 millones. En serverless, la optimización se nota directamente en la factura.

Por debajo del punto de cruce, serverless es más barato porque el tiempo inactivo es gratis; por encima, un servicio con carga constante paga más por el mismo trabajo. El tráfico a ráfagas con largos periodos de inactividad favorece a serverless incluso con volúmenes más altos.

La comparación en bruto deja fuera el mayor costo para muchos equipos: las personas. Parchear, escalar y hacer guardias para las VM puede costar fácilmente más que la diferencia, lo que eleva el punto de cruce real. Los cargos por transferencia de datos, logging y observabilidad se aplican a ambas opciones y a menudo superan cualquiera de las dos facturas de cómputo.

Como el resultado depende de la duración, la memoria, la forma del tráfico y los precios actuales, rehaz el cálculo con precios reales y tráfico medido antes de decidir, y vuelve a hacerlo a medida que crezca el tráfico.

El punto de cruce, dibujado



Con estos precios ilustrativos, el costo serverless crece £3.20 por cada millón de solicitudes y alcanza el costo fijo de £100 de lo siempre activo en unos 31 millones de solicitudes al mes.

Cuándo encaja serverless y cuándo no

Serverless encaja en cargas de trabajo **con picos, impredecibles o de bajo volumen**, en las que pagar por servidores inactivos es un desperdicio; en el **pegamento dirigido por eventos**, como reaccionar a subidas de archivos, mensajes de cola y cambios en bases de datos; en los **trabajos programados** y la automatización administrativa; en los **prototipos y productos nuevos**, en los que el equipo debería dedicar su tiempo al producto y no a la infraestructura; y en equipos sin capacidad de plataforma ni de operaciones.

Encaja mal con el tráfico **constante y de gran volumen**, en el que el precio por solicitud supera a la capacidad siempre activa (como en el ejemplo práctico); con los procesos **de larga duración o con estado**, como la transcodificación de video que supera el límite de tiempo o los servidores WebSocket que mantienen muchas conexiones; con requisitos **estrictos de baja latencia de cola** que los arranques en frío incumplirían; con cargas de trabajo que necesitan hardware especial o un control fino del entorno de ejecución; y con diseños de grano fino en los que muchas funciones se llaman entre sí de forma síncrona, lo que añade saltos de red, costo y puntos de fallo en cada paso. El Capítulo 1 describió cómo Amazon Prime Video, en 2023, pasó

un servicio de monitorización de pasos serverless distribuidos a un único proceso e informó de una reducción de más del 90% en el costo de infraestructura (por qué importa la arquitectura): no es un veredicto sobre serverless, sino un recordatorio de que el encaje importa más que la moda.

Arquitectura basada en espacios: sacar la base de datos de la ruta crítica

La mayoría de las arquitecturas escalan con facilidad los niveles web y de aplicación, pero acaban chocando con la base de datos, que toca cada solicitud. Añadir servidores delante solo traslada la cola a la base de datos. La arquitectura basada en espacios, tal como la describen Mark Richards y Neal Ford en *Fundamentals of Software Architecture*, elimina por completo la base de datos de la ruta síncrona. Cada solicitud se gestiona íntegramente en memoria en una de muchas unidades de procesamiento idénticas, y la base de datos se actualiza de forma asíncrona, entre bastidores.

El nombre procede del espacio de tuplas, una memoria asociativa compartida introducida por el lenguaje de coordinación Linda de David Gelernter en 1985: los procesos se comunican escribiendo tuplas en un espacio compartido y leyendo o extrayendo las tuplas que coinciden, sin conocerse entre sí. JavaSpaces, de Sun, llevó la idea a Java a finales de la década de 1990, y plataformas comerciales construyeron sobre ella espacios distribuidos y replicados, que evolucionaron hasta los grids de datos en memoria actuales, como Hazelcast, Apache Ignite y Oracle Coherence.

Las partes de un sistema basado en espacios

1. **Unidad de procesamiento**
   Una unidad desplegable que contiene la lógica de la aplicación y un grid de datos en memoria (in-memory data grid) con los datos que necesita esa lógica, además de un motor de replicación. Las unidades son idénticas, así que añadir una añade capacidad. En el ejemplo de venta de entradas, cada unidad contiene el mapa de asientos y las reservas provisionales actuales de los eventos que atiende.

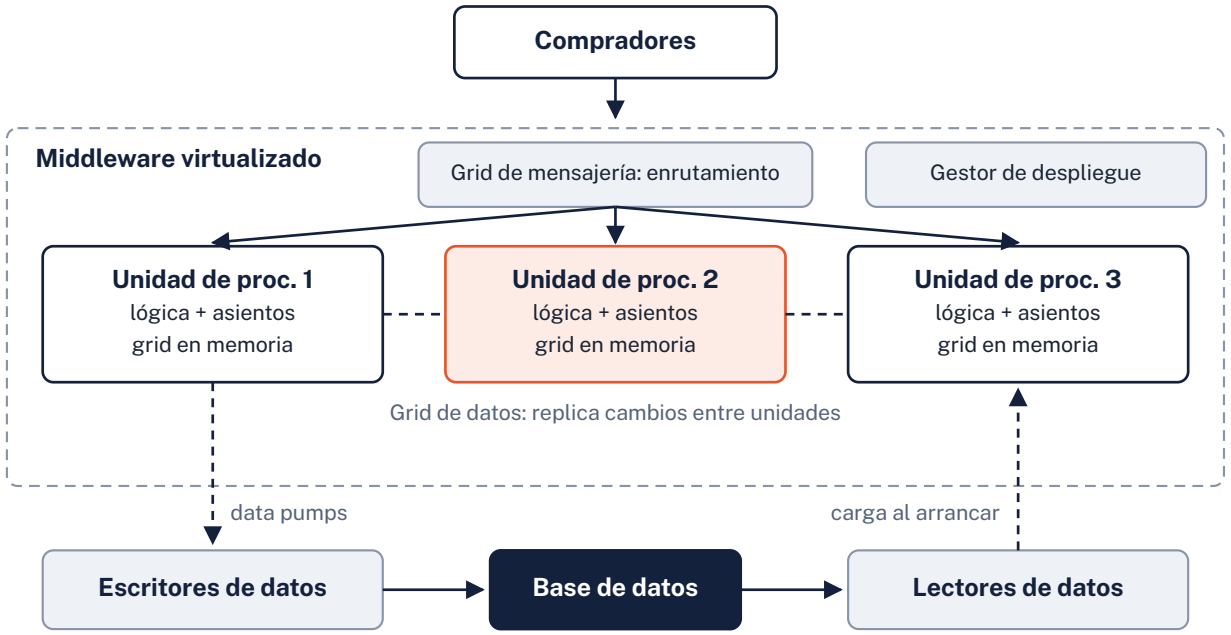
2 **Middleware virtualizado**
La infraestructura que gestiona las unidades. El **grid de mensajería** recibe las solicitudes y dirige cada una a una unidad disponible. El **grid de datos** replica los cambios entre las copias en memoria de las unidades. El **grid de procesamiento**, opcional, coordina las solicitudes que necesitan varios tipos de unidad. El **gestor de despliegue** arranca y detiene unidades a medida que la carga sube y baja.

3 **Bombas de datos (data pumps)**
Canales asíncronos, normalmente colas de mensajes persistentes, que llevan cada cambio hecho en una unidad al lado de la base de datos. Si la base de datos va lenta o está caída, los cambios esperan en la cola; los compradores no se ven afectados.

4 **Escritores de datos (data writers)**
Componentes que consumen los mensajes de las bombas y escriben los cambios en la base de datos. Pueden escalarse y pausarse con independencia de las unidades.

5 **Lectores de datos (data readers)**
Componentes que cargan datos de la base de datos en las unidades cuando el grid arranca en frío, por ejemplo después de que se hayan reiniciado o vuelto a desplegar todas las unidades. No se usan en el procesamiento normal de solicitudes.

Arquitectura basada en espacios para una puesta a la venta de entradas



Las solicitudes se atienden desde memoria en unidades de procesamiento idénticas; la base de datos se actualiza de forma asíncrona mediante bombas y escritores de datos, y solo se lee para precargar el grid.

Colisiones de datos y otros costos

Mantener copias replicadas en muchas unidades traslada a la memoria los problemas de consistencia de las secciones anteriores. Con una caché **replicada**, cada unidad tiene una copia completa y los cambios se copian de forma asíncrona, lo que da lecturas locales rápidas, pero dos unidades pueden actualizar el mismo elemento antes de que la replicación se ponga al día. Richards y Ford llaman a esto colisión de datos y estiman su tasa como el número de unidades multiplicado por el cuadrado de la tasa de actualización, dividido por el número de elementos y multiplicado por la latencia de replicación. Con 5 unidades, 20 actualizaciones por segundo, 50 000 elementos y 100 ms de latencia, eso da 5 × 400 ÷ 50 000 × 0.1 = 0.004 por segundo, unas 115 colisiones en una jornada de ocho horas; reducir la latencia a 1 ms la deja en torno a una.

Para los datos que nunca deben colisionar, como el último asiento de una fila, las alternativas son una caché **distribuida**, en la que cada elemento vive en una partición y todas las unidades lo leen y escriben allí (consistente, pero cada acceso es una llamada de red y las particiones del grid deben seguir disponibles), o dirigir todas las solicitudes de un evento o una zona a la misma unidad, de modo que solo una unidad cambie esos asientos. La mayoría de los diseños reales combinan ambas cosas: particionan por clave los datos calientes y disputados, y replican los datos que se leen mucho y se escriben poco, como los mapas del recinto y los precios.

Los demás costos son importantes. Los datos guardados en memoria están limitados por el tamaño de la memoria y deben reconstruirse tras un reinicio completo. La base de datos es eventualmente consistente con el grid, así que los informes y otros sistemas ven los cambios con retraso, y perder cambios en cola es un riesgo real que debe cubrir la durabilidad de las bombas. Probar con una carga de pico realista es caro y difícil. Y el estilo es complejo y costoso de construir: Richards y Ford le dan una puntuación baja en simplicidad y en capacidad de prueba, y alta en costo. Solo merece la pena cuando la carga de verdad no puede atenderse con caché, réplicas de lectura y una base de datos bien ajustada.

EJEMPLO PRÁCTICO

Ejemplo práctico: la puesta a la venta de un estadio

Las entradas de una gira por estadios salen a la venta a las 10:00. Hay unas 400 000 personas esperando; durante los primeros diez minutos, la demanda es cientos de veces la tasa normal y luego decae. El sistema de venta de entradas no debe vender un asiento dos veces ni debe caerse.

Antes de la puesta a la venta	Los lectores de datos cargan en el grid los mapas de asientos de cada espectáculo. El gestor de despliegue arranca suficientes unidades de procesamiento para el pico previsto.
Enrutamiento	El grid de mensajería dirige las solicitudes por espectáculo y zona, de modo que todas las reservas provisionales de la zona 112 del espectáculo del sábado las gestiona la misma unidad. Las reservas provisionales de asientos nunca colisionan.
Un comprador selecciona dos asientos	La unidad los marca como HELD para este comprador durante 8 minutos, en memoria, en microsegundos, y responde. Los mapas de asientos de los demás compradores se actualizan mediante la replicación.
La reserva provisional se convierte en venta	Se ejecuta el paso de pago (una llamada externa, con una clave de idempotencia); si tiene éxito, la unidad marca los asientos como SOLD y una bomba de datos pone la venta en cola.
La base de datos	Los escritores de datos escriben las ventas a un ritmo constante que la base de datos puede sostener. Durante el pico, la cola crece; se vacía unos minutos después de que baje la demanda. Ninguna solicitud de un comprador espera a la base de datos.
Una reserva provisional caduca	La unidad libera los asientos en memoria; la bomba pone en cola un registro de liberación.

Muchos sistemas de venta de entradas colocan además una sala de espera virtual delante de la venta, que admite a los compradores a un ritmo controlado. Es mucho más sencilla y a menudo basta por sí sola; la arquitectura basada en espacios es para cuando la propia carga admitida sigue siendo demasiado alta para un diseño centrado en la base de datos.

Enrutar por zona es la decisión de diseño que convierte el problema de las colisiones de "raro pero posible" en "imposible" para los datos que más importan.

Comparación de características (cualitativa, para diseños típicos)

Característica	Serverless (FaaS y BaaS)	Basada en espacios
Elasticidad	Muy alta, automática, incluido el escalado a cero	Muy alta para los picos, pero las unidades tardan en arrancar y cargar los datos
Rendimiento	Bueno en caliente; los arranques en frío y los saltos de red añaden latencia	Muy alto: las solicitudes se atienden desde memoria
Perfil de costos	Pago por uso; excelente con inactividad o picos, caro con carga constante	Alto: muchas unidades, mucha memoria, middleware complejo, siempre activo
Simplicidad	Sencilla por función; el sistema completo puede volverse difícil de seguir	Baja: replicación, colisiones, bombas y carga en el arranque en frío
Consistencia	Lo que ofrezcan los almacenes de datos gestionados	La replicación en memoria es eventual salvo que los datos disputados se particionen; la base de datos va con retraso
Capacidad de prueba (testabilidad)	Las unidades son fáciles de probar; la integración necesita entornos en la nube	Difícil: probar con picos realistas es caro
Esfuerzo operativo	Bajo en infraestructura; alto en observabilidad y límites	Alto
Portabilidad	Baja: disparadores y servicios específicos del proveedor	Media: depende del producto de grid de datos

Lista de comprobación: antes de elegir serverless o basada en espacios

Responde a estas preguntas con cifras medidas siempre que puedas.

- [ ] **Forma del tráfico**
  ¿La carga tiene picos, está inactiva gran parte del tiempo o es constante? Serverless premia el tiempo inactivo; la arquitectura basada en espacios premia los picos extremos; la carga constante favorece los servicios normales siempre activos.
- [ ] **Punto de cruce de costos**
  ¿Has calculado el costo por solicitud frente al costo de lo siempre activo con precios, duraciones y tráfico reales, incluidos el gateway, la transferencia de datos y las personas?
- [ ] **Presupuesto de latencia**
  ¿La respuesta más lenta aceptable puede absorber un arranque en frío, o el salto de red hasta donde vive el estado?
- [ ] **Límites**
  ¿Algún trabajo supera los límites de tiempo de ejecución, memoria o carga útil? ¿Puede dividirse en pasos o pasarse por referencia?
- [ ] **Estado y conexiones**
  ¿Dónde vive cada tipo de estado, y pueden los almacenes de datos soportar ráfagas repentinas de conexiones concurrentes?
- [ ] **Idempotencia**
  ¿Puede cada manejador ejecutarse dos veces sin riesgo para el mismo evento?
- [ ] **Contención**
  En los diseños basados en espacios, ¿qué datos están disputados, y se particionarán o enrutarán para que no puedan colisionar?
- [ ] **Durabilidad del retraso**
  Si el grid o las bombas fallan, ¿qué se pierde? ¿Las bombas son persistentes, y es aceptable el retraso de la base de datos para los informes y otros sistemas?
- [ ] **Alternativas más sencillas**
  ¿La caché, las réplicas de lectura, una cola delante de la base de datos o una sala de espera virtual cubrirían la necesidad con menos complejidad?
- [ ] **Decisión sobre la dependencia del proveedor**
  ¿Está registrado el acoplamiento al proveedor o al producto de grid como una decisión consciente, con la lógica de negocio detrás de puertos?

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué debe ser idempotente una función activada por una cola?**

Porque los disparadores asíncronos suelen entregar al menos una vez y reintentan en caso de fallo, así que el mismo evento puede invocar la función más de una vez.

▼ **Con los precios ilustrativos del ejemplo práctico, ¿cuánto costarían aproximadamente 50 millones de solicitudes al mes en serverless, y qué opción es más barata?**

50 × £3.20 = £160 al mes, así que la opción siempre activa de £100 es más barata en costo bruto de infraestructura, antes de contar a las personas y otros cargos.

▼ **En la arquitectura basada en espacios, ¿qué hacen los lectores de datos y cuándo?**

Cargan datos de la base de datos en los grids en memoria de las unidades de procesamiento cuando el grid arranca en frío, por ejemplo tras un reinicio o un nuevo despliegue completos; no se usan para las solicitudes normales.

▼ **¿De qué sirve dirigir todas las solicitudes de una zona del estadio a una sola unidad de procesamiento?**

Solo esa unidad cambia esos asientos, así que dos unidades no pueden actualizar el mismo asiento antes de que la replicación se ponga al día, lo que elimina las colisiones de datos para esos datos.

▼ **Inténtalo: Precios ilustrativos: cómputo a £0.00002 por GB-segundo, £0.20 por millón de invocaciones, £1 por millón de solicitudes al gateway. Cada solicitud tarda 300 ms con 1 GB. Dos servidores siempre encendidos cuestan £288 al mes. Encuentra el punto de cruce. ¿Qué elegirías con 5 millones de solicitudes?**

Cada solicitud usa 0.3 GB-segundos, o £6 por millón, de modo que el total es £7.20 por millón. Punto de cruce: 288 / 7.20 = 40 millones de solicitudes al mes. Con 5 millones, serverless cuesta £36 frente a £288, así que elige funciones. También importan la forma del tráfico y los costos de personal, y los arranques en frío pueden descartarlo si la latencia es ajustada. Si el volumen se acerca a 40 millones, las cuentas se invierten, así que mantén la lógica detrás de puertos para facilitar el cambio.

FUENTES

- Mike Roberts: Serverless Architectures (martinfowler.com, 2018): https://martinfowler.com/articles/serverless.html
- AWS: Introducing AWS Lambda (announcement, 13 November 2014): https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/

- AWS Lambda Developer Guide: Lambda quotas: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html
- Adrian Colyer: Generative Communication in Linda (summary of Gelernter, 1985): https://blog.acolyer.org/2015/02/17/generative-communication-in-linda/

Lecturas recomendadas

- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. Capítulos claros sobre los estilos basados en servicios, dirigidos por eventos, basados en espacios y de microservicios, con valoraciones de sus características, y la lista ampliada de falacias de la computación distribuida.
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. Análisis de compromisos para sistemas distribuidos: cuantos de arquitectura, granularidad de los servicios (desintegradores e integradores), propiedad de los datos y flujos de trabajo distribuidos.
- *Building Microservices (2nd edition)*, Sam Newman (2021), O'Reilly Media. La guía de referencia para profesionales sobre microservicios, construida en torno a la desplegabilidad independiente, con consejos equilibrados sobre cuándo no usarlos.
- *Microservices*, James Lewis and Martin Fowler (2014), martinfowler.com. El artículo que definió el término para la mayor parte del sector, con sus nueve características comunes.
  https://martinfowler.com/articles/microservices.html
- *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Gregor Hohpe and Bobby Woolf (2003), Addison-Wesley. El vocabulario de la mensajería (canales, mensajes de evento y de comando, receptores idempotentes, canales de mensajes fallidos) que todo diseño dirigido por eventos sigue usando.
  https://www.enterpriseintegrationpatterns.com/
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. La mejor explicación de la replicación, la consistencia, los logs, el procesamiento de flujos y los problemas de los sistemas distribuidos.
  https://martin.kleppmann.com/2026/03/24/designing-data-intensive-applications-2e.html
- *Building Event-Driven Microservices (2nd edition)*, Adam Bellemare (2025), O'Reilly Media. Los flujos de eventos como columna vertebral de una arquitectura de servicios: diseño de eventos, esquemas, propiedad y procesamiento de flujos.

Glosario

API gateway (puerta de enlace de API)
Un punto de entrada único delante de los servicios que se encarga de inquietudes transversales como TLS, autenticación, limitación de tasa (rate limiting) y enrutamiento.

Arquitectura basada en espacios
Estilo arquitectónico que atiende las solicitudes desde muchas unidades de procesamiento idénticas que guardan los datos en mallas replicadas en memoria y actualizan la base de datos de forma asíncrona, para que quede fuera de la ruta de la solicitud.

Arquitectura basada en servicios
Estilo distribuido con unos pocos servicios de dominio de grano grueso desplegados por separado (normalmente de 4 a 12) que a menudo comparten una base de datos; bautizado por Mark Richards y Neal Ford.

Arquitectura dirigida por eventos
Un estilo en el que los componentes se comunican publicando eventos sobre cosas que han ocurrido y reaccionando a los eventos publicados por otros, normalmente a través de un intermediario.

Arquitectura orientada a servicios (SOA)
Enfoque, dominante en las empresas desde principios de los años 2000, que expone las capacidades de negocio como servicios compartidos con contratos explícitos, normalmente conectados mediante un bus de servicios empresarial.

Arranque en frío (cold start)
El retraso adicional que se produce cuando una plataforma serverless debe crear e inicializar una nueva instancia de función antes de atender una solicitud, desde decenas de milisegundos hasta varios segundos.

AsyncAPI
Una especificación legible por máquinas para describir API dirigidas por mensajes: los canales, las operaciones que envían o reciben en ellos y los esquemas de los mensajes.

Backend como servicio (BaaS)
Servicios de terceros totalmente gestionados, como autenticación, bases de datos, almacenamiento y mensajería, que sustituyen componentes del lado del servidor que una organización tendría que construir y operar por su cuenta.

Backend for frontend (BFF)
Un backend ligero construido para un tipo de cliente, como la app móvil, que llama a los servicios y adapta sus datos para ese cliente.

Bandeja de salida transaccional (transactional outbox)
Patrón en el que un servicio escribe los mensajes salientes en una tabla de bandeja de salida (outbox) en la misma transacción local que su cambio de negocio; más tarde un retransmisor los publica, lo que da una entrega al menos una vez sin transacción distribuida.

Base de datos por servicio
La regla de que cada servicio es propietario de sus datos y ningún otro servicio los lee ni los escribe directamente; los demás usan la API del propietario o sus eventos.

Bus de servicios empresarial (ESB)
Middleware situado en el centro de la SOA clásica que enruta, transforma y media en los mensajes entre servicios y que a menudo también ejecuta procesos de negocio.

Cambio compatible hacia atrás
Un cambio en una interfaz o un esquema que los consumidores existentes pueden soportar sin modificaciones, como añadir un campo opcional. Eliminar o renombrar un campo es un cambio incompatible (breaking change).

Captura de datos de cambios (CDC)
Leer el propio registro de cambios o de replicación de una base de datos para transmitir a otros sistemas cada inserción, actualización y eliminación confirmada. Debezium es una plataforma de CDC de código abierto muy utilizada.

Clave de idempotencia
Un valor único que un cliente envía con una solicitud no idempotente, como POST, para que el servidor pueda reconocer un reintento de la misma operación y devolver el resultado original en lugar de hacer el trabajo dos veces.

Cola (canal punto a punto)
Canal de mensajería en el que cada mensaje se entrega exactamente a uno de los consumidores que leen de él, de modo que varias instancias pueden repartirse el trabajo.

Colisión de datos
En una caché en memoria replicada, dos unidades que actualizan el mismo elemento antes de que se haya propagado la replicación de la primera actualización, de modo que una actualización sobrescribe la otra.

Comando
Un mensaje que pide a un manejador concreto que haga algo, como AuthorisePayment. A diferencia de un evento, puede rechazarse.

Comunicación asíncrona
Una interacción en la que el emisor entrega un mensaje a un intermediario, como un intermediario de mensajes (message broker), y continúa sin esperar; el receptor lo procesa cuando está listo.

Comunicación síncrona
Interacción en la que quien llama envía una solicitud y espera la respuesta antes de continuar, como en una llamada telefónica. Ambas partes deben estar disponibles en el mismo momento.

Confirmación en dos fases (two-phase commit, 2PC)
Protocolo que hace que varios participantes confirmen o aborten juntos: un coordinador pide primero a todos que se preparen y voten, y después comunica a todos el resultado. Los participantes que votaron sí quedan bloqueados si el coordinador falla.

Consistencia causal
Un modelo de consistencia en el que todos ven en el mismo orden las operaciones relacionadas causalmente (una pudo haber influido en la otra); las operaciones concurrentes no relacionadas pueden verse en órdenes distintos.

Consistencia eventual
Una garantía de que las copias de unos datos que han dejado de actualizarse convergerán al mismo valor al cabo de un tiempo, mientras que entretanto los lectores pueden ver valores obsoletos.

Consistencia secuencial
Modelo de consistencia en el que todos los clientes ven las operaciones en un único orden acordado que conserva el orden propio de cada cliente, pero ese orden no tiene por qué coincidir con el tiempo real.

Consumidor idempotente
Un receptor de mensajes que registra qué mensajes ha procesado, en la misma transacción que sus efectos, de modo que los duplicados no cambian nada.

Contramedida de saga
Técnica de diseño que reduce las anomalías causadas por la falta de aislamiento de las sagas, como un bloqueo semántico (un estado PENDING), las actualizaciones conmutativas o volver a leer un valor antes de actualizarlo.

Contrato dirigido por el consumidor
Un contrato construido a partir de lo que cada consumidor usa realmente de la API de un proveedor; el proveedor ejecuta en su propia compilación las expectativas de todos sus consumidores, de modo que sabe qué cambios romperían a quién.

Controles hipermedia (HATEOAS)
Enlaces en una respuesta de una API que indican al cliente qué acciones puede realizar a continuación, de modo que el cliente sigue enlaces en lugar de construir las URL por sí mismo. Es el nivel 3 del modelo de madurez de Richardson.

Coreografía
Coordinación de un proceso de varios pasos mediante participantes que reaccionan a los eventos de los demás, sin que ningún componente esté al mando.

CQRS
Segregación de responsabilidades de comando y consulta (command query responsibility segregation): usar modelos separados, y a menudo almacenes separados, para actualizar los datos y para leerlos.

Descubrimiento de servicios
Mecanismo, como un registro o DNS, mediante el cual los clientes encuentran las direcciones de red actuales de las instancias de servicio en buen estado.

Desplegabilidad independiente
La capacidad de modificar y lanzar a producción un servicio sin modificar, probar ni lanzar ningún otro servicio al mismo tiempo.

Entrega al menos una vez (at-least-once)
Una garantía de que cada mensaje se procesa, posiblemente más de una vez; el receptor debe gestionar los duplicados.

Escritura dual
Actualizar dos sistemas, como una base de datos y un intermediario de mensajes, en operaciones separadas; una caída entre ambas deja los dos sistemas incoherentes.

Espacio de tuplas
Memoria asociativa compartida en la que los procesos escriben tuplas y de la que leen o toman las tuplas que coinciden, sin saber unos de otros. Presentado con el lenguaje Linda de David Gelernter (1985).

Event sourcing
Almacenar cada cambio de una entidad como un evento, de modo que el registro de eventos es la fuente de verdad y el estado actual se obtiene reproduciéndolo.

Evento
Un registro inmutable de algo que ha ocurrido, nombrado en tiempo pasado, como OrderPlaced. Cualquier número de receptores puede reaccionar a él.

Falacias de la computación distribuida
Ocho suposiciones falsas sobre las redes enumeradas en Sun Microsystems por L Peter Deutsch y sus colegas, con James Gosling añadiendo la octava: por ejemplo, que la red es fiable y que la latencia es cero.

Fallo en cascada
Un fallo que se propaga: un componente lento o caído hace que quienes lo llaman esperen y agoten sus propios recursos, de modo que también fallan, y así sucesivamente a lo largo de la cadena.

Fallo parcial
Situación en la que algunas partes de un sistema distribuido funcionan mientras otras han fallado, son lentas o no son accesibles, a menudo sin que las partes que funcionan puedan saber cuáles.

Garantías de sesión
Promesas de consistencia por cliente, descritas por Terry y colaboradores en 1994: leer tus propias escrituras, lecturas monótonas, escrituras monótonas y escrituras que siguen a las lecturas.

GraphQL
Un lenguaje de consulta y un entorno de ejecución para API, publicado por Facebook en 2015, en el que el cliente envía una consulta que nombra exactamente los campos que quiere de un esquema tipado y recibe una respuesta con la misma forma.

Grid de datos en memoria (in-memory data grid)
Software que mantiene datos en la memoria de muchas máquinas como un único almacén lógico, con particionado y replicación, para un acceso muy rápido. Hazelcast, Apache Ignite y Oracle Coherence son ejemplos.

gRPC
Un framework de llamada a procedimiento remoto, liberado como código abierto por Google en 2015, que envía mensajes de Protocol Buffers sobre HTTP/2 y genera código tipado de cliente y de servidor. Admite streaming en una o en ambas direcciones.

Grupo de consumidores
Un conjunto de consumidores que se reparten el trabajo de leer un topic: cada partición la lee un miembro, y cada grupo mantiene sus propios desplazamientos (offsets).

Idempotencia
La propiedad de una operación que tiene el mismo efecto tanto si se realiza una vez como varias, lo que hace que los reintentos sean seguros. Normalmente se consigue con un identificador único de solicitud o de evento que el receptor recuerda.

Last write wins (gana la última escritura)
Regla de resolución de conflictos que conserva la escritura con la marca de tiempo más reciente y descarta las demás. Es simple, pero las actualizaciones concurrentes se pierden sin aviso y las diferencias entre relojes pueden elegir al ganador equivocado.

Lecturas monótonas
Garantía de que, una vez que un cliente ha visto un valor, sus lecturas posteriores nunca devuelven uno más antiguo, de modo que los datos nunca parecen retroceder en el tiempo.

Leer tus propias escrituras (read-your-writes)
Garantía de que, después de que un cliente escribe un valor, las lecturas posteriores de ese mismo cliente ven la escritura o algo más reciente, aunque otros clientes todavía no la vean.

Linealizabilidad
El modelo de consistencia de objeto único más fuerte: cada operación parece ocurrir en un instante entre su inicio y su fin, en un orden que comparten todos los clientes y que respeta el tiempo real, como si hubiera una sola copia.

Mensaje
Cualquier dato enviado de un programa a otro a través de un canal; puede contener un evento, un comando, una consulta o una respuesta.

Microservicios
Estilo en el que una aplicación se construye como pequeños servicios, cada uno en su propio proceso, organizados en torno a una capacidad de negocio, desplegables de forma independiente y comunicados mediante mecanismos ligeros.

Modelo de consistencia
El contrato que ofrece un almacén de datos sobre qué valores pueden devolver las lecturas en función de las escrituras que se han producido y, por tanto, para qué resultados obsoletos o reordenados deben estar preparados los clientes.

Modelo de datos canónico
Una única representación de cada entidad de negocio para toda la empresa, a la que se traducen todos los mensajes y desde la que se traducen. Rara vez encaja bien en todas las partes de un negocio.

Notificación de eventos
Un evento que solo señala un cambio, a menudo con apenas un identificador, de modo que los receptores que necesiten detalles deben preguntar al publicador.

Offset (desplazamiento)
La posición secuencial de un evento dentro de una partición; los consumidores registran el offset al que han llegado para poder reanudar después de un reinicio.

OpenAPI
Formato estándar y legible por máquina para describir API HTTP (rutas, operaciones, parámetros, esquemas), antes llamado especificación Swagger. Sirve de base para la documentación, la generación de código y las pruebas.

Orquestación
Coordinación de un proceso de varios pasos por parte de un componente que indica a cada participante qué hacer y decide qué ocurre a continuación.

Orquestador de sagas
Componente que coordina una saga enviando comandos a los participantes, siguiendo el progreso en una máquina de estados persistente y desencadenando compensaciones en caso de fallo.

PACELC
La extensión de CAP de Daniel Abadi: si hay una Partición (Partition), elige entre Disponibilidad (Availability) y Consistencia (Consistency); si no (Else), en funcionamiento normal, elige entre Latencia (Latency) y Consistencia (Consistency).

Paginación por cursor
Recorrer una colección por páginas pasando un token opaco que marca el último elemento visto, en lugar de un desplazamiento numérico. Sigue siendo rápida y estable cuando se añaden elementos durante la paginación.

Partición (de un topic)
En Kafka y registros similares, una parte ordenada de solo anexado (append-only) de un topic. Los eventos con la misma clave van a la misma partición; el orden solo está garantizado dentro de ella.

Partición de red
Fallo de red que divide un sistema en grupos de nodos que no pueden comunicarse entre sí, aunque cada grupo pueda seguir funcionando y ser accesible para algunos clientes.

Problem details (RFC 9457)
Formato JSON estándar para los errores de las API HTTP, con los miembros type, title, status, detail e instance, servido como application/problem+json. El RFC 9457 (2023) sustituyó al RFC 7807.

Problema de las consultas N+1
Fallo de rendimiento en el que el código hace una consulta para obtener una lista de N elementos y luego una consulta más por elemento, lo que da N+1 viajes de ida y vuelta donde bastarían uno o dos.

Protocol Buffers
Formato binario compacto e independiente del lenguaje para datos estructurados, definido en archivos .proto en los que cada campo tiene un número permanente. Lo usan gRPC y muchos sistemas de mensajería.

Pruebas de contrato
Probar cada lado de una integración de forma aislada frente a una descripción compartida de sus interacciones, en lugar de desplegar ambos juntos para hacer pruebas de extremo a extremo.

Publicación-suscripción (publish-subscribe)
Patrón de mensajería en el que un publicador envía un mensaje a un topic y cada suscriptor recibe su propia copia, sin que el publicador sepa quiénes son.

Quórum (R + W > N)
En un almacén con N réplicas, una escritura espera W confirmaciones y una lectura R respuestas; si R + W > N, cada conjunto de lectura se solapa con cada conjunto de escritura en al menos una réplica.

Saga
Transacción de negocio implementada como una secuencia de transacciones locales, cada una emparejada con una transacción compensatoria, de modo que o se completan todos los pasos o los pasos completados se deshacen semánticamente. Presentada por Garcia-Molina y Salem en 1987.

Service mesh (malla de servicios)
Capa de infraestructura formada por proxies de red, configurados de forma centralizada, que proporciona cifrado entre servicios, reintentos, timeouts, control del tráfico y telemetría.

Servicio de dominio
En la arquitectura basada en servicios, un servicio desplegado por separado que abarca toda un área de negocio, como pedidos o pagos, desde la API hasta el acceso a los datos.

Sistema distribuido
Un conjunto de programas que se ejecutan en distintas computadoras y cooperan enviándose mensajes a través de una red, de modo que los usuarios ven un único servicio.

Teorema CAP

El resultado, conjeturado por Eric Brewer en 2000 y demostrado por Gilbert y Lynch en 2002, según el cual, durante una partición de red, un sistema replicado no puede ser a la vez linealizable y estar disponible para todas las solicitudes.

Topología de intermediario (broker)

Una topología dirigida por eventos sin coordinador central: los procesadores reaccionan a los eventos y publican nuevos eventos, de modo que el flujo de trabajo surge de sus reacciones.

Topología de mediador

Topología dirigida por eventos en la que un mediador recibe un evento inicial y dirige el flujo de trabajo enviando comandos a los procesadores en un orden definido.

Transacción compensatoria

Una nueva transacción que contrarresta semánticamente el efecto de otra anterior ya confirmada, como anular una autorización de tarjeta. A diferencia de un rollback, puede que la original ya se haya visto.

Transferencia de estado transportada por eventos

Eventos que transportan datos suficientes para que los receptores mantengan su propia copia, de modo que no necesitan llamar al publicador.

XA

La interfaz estándar de X/Open (1991) entre un gestor de transacciones y gestores de recursos como bases de datos e intermediarios de mensajes, usada para ejecutar una confirmación en dos fases entre ellos.