BE SOFTWARE ARCHITECTURE WISE · CAPÍTULO 9

Rendimiento, escalabilidad, resiliencia y operabilidad

Este capítulo convierte los escenarios de rendimiento y disponibilidad del Capítulo 2 en sistemas que funcionan. Trata el escalado vertical y horizontal y el cubo de escalado, el balanceo de carga y los servicios sin estado, las colas, el back pressure y la limitación de tasa (rate limiting), los patrones de resiliencia que impiden que los fallos se propaguen (timeouts, reintentos, circuit breakers, bulkheads), los SLI, los SLO y los presupuestos de errores, la observabilidad con logs, métricas y trazas, la planificación de capacidad y las pruebas de rendimiento, y la ingeniería del caos y el aprendizaje a partir de incidentes. Está pensado para desarrolladores y arquitectos que conocen los estilos distribuidos del Capítulo 6 y deben mantener un sistema rápido y en funcionamiento bajo carga real.

bewiseapps.com/es/software-architecture-wise/resilience-and-operability/ · Versión 2026-10-26. Solo información educativa, no asesoramiento profesional.

DE UN VISTAZO

Aprenderás a

- Elegir entre el escalado vertical y el horizontal y entre los ejes X, Y y Z del cubo de escalado, y dimensionar una flota para un pico conocido con la ley de Little y un margen de capacidad (headroom).
- Seleccionar señales y ajustes de autoescalado que eviten el retraso y el flapping (oscilación), y explicar por qué la CPU suele ser la métrica equivocada.
- Comparar el balanceo de carga de capa 4 y de capa 7 y los principales algoritmos, diseñar comprobaciones de estado y el drenaje de conexiones, y externalizar el estado de sesión.
- Aplicar colas, búferes acotados, back pressure, descarte de carga y limitación de tasa (incluido el token bucket) para mantener la sobrecarga bajo control.
- Combinar timeouts, plazos, reintentos con backoff y jitter, presupuestos de reintentos, circuit breakers, bulkheads y alternativas (fallbacks) para contener los fallos.
- Definir SLI y SLO desde el punto de vista del usuario, calcular presupuestos de errores y tasas de consumo (burn rates), y diseñar alertas basadas en ellos.
- Instrumentar un sistema con logs, métricas y trazas usando OpenTelemetry, y usar los métodos RED y USE para diagnosticar problemas.
- Planificar pruebas de capacidad y de rendimiento, ejecutar experimentos de caos de forma segura y dirigir revisiones posteriores a incidentes sin culpas.

Antes de empezar

El Capítulo 2, en especial las secciones sobre rendimiento y escalabilidad y sobre disponibilidad y fiabilidad; el Capítulo 6, en especial por qué distribuir, la comunicación y las API; y las secciones del Capítulo 7 sobre replicación, particionamiento y caché.

Tiempo

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

Términos clave

cubo de escalado · autoescalado · servicio sin estado · balanceador de carga de capa 7 · poder de dos elecciones aleatorias (power of two choices) · back pressure (contrapresión) · descarte de carga (load shedding) · token bucket (cubeta de tokens) · circuit breaker (disyuntor) · bulkhead (mamparo) · presupuesto de errores · tasa de consumo (burn rate) · trazado distribuido · ingeniería del caos

De las cualidades sobre el papel a las cualidades en producción

El Capítulo 2 hizo precisos el rendimiento y la disponibilidad: percentiles de latencia, rendimiento (throughput), la ley de Little, la utilización y los nueves, escritos como escenarios de atributos de calidad que se pueden probar. Este capítulo trata de cómo lograr esos escenarios en un sistema en ejecución y de mantenerlos ciertos el día de más actividad del año. Un diseño que cumple sus objetivos en un diagrama puede venirse abajo a las 12:30 de un

viernes, cuando todas las oficinas de la ciudad piden el almuerzo, o a las 10:00 de la mañana en que salen a la venta las entradas de un concierto. La diferencia entre ambos rara vez es un código ingenioso. Es un conjunto de decisiones arquitectónicas sobre cómo se añade capacidad, cómo se reparte la carga, cómo espera el trabajo, cómo se contienen los fallos y cómo saben las personas que operan el sistema qué está haciendo.

Dos ejemplos recurrentes recorren el capítulo. La **plataforma de reparto de comida** tiene un pico diario muy marcado a la hora del almuerzo y otro por la noche, decenas de servicios y dependencias de terceros como los proveedores de pago y las API de mapas. La **puesta a la venta de entradas** es el caso extremo: una demanda que pasa de casi nada a muchas veces la capacidad del sistema en cuestión de segundos, para un número fijo de asientos, con todo el país mirando. Entre los dos ponen a prueba todas las técnicas del capítulo.

Gestionar la carga: escalar, balancear y esperar

Las cuatro primeras secciones tratan de la carga. Estrategias de escalado compara el escalado vertical con el horizontal, usa el cubo de escalado de Abbott y Fisher para separar la clonación, la división por función y la división por datos, y explica por qué el autoescalado basado en la CPU suele ser la elección equivocada y cómo dimensionar una flota para un pico conocido con la ley de Little y un margen de capacidad. Balanceo de carga, ausencia de estado y sesiones trata de lo que se sitúa delante de las instancias: los balanceadores de carga de capa 4 y de capa 7, los algoritmos que usan (incluido el poder de dos elecciones aleatorias), las comprobaciones de estado que no dejan fuera de servicio una flota entera de golpe, el drenaje de conexiones y por qué las sesiones persistentes deben dar paso a un estado de sesión externalizado.

Asincronía, colas y back pressure trata de lo que ocurre cuando el trabajo llega más deprisa de lo que se puede hacer. Las colas convierten los picos en trabajo acumulado (backlog), lo cual solo es útil si ese trabajo acumulado está acotado y se vacía en un tiempo aceptable. El back pressure, el descarte de carga y la limitación de tasa deciden qué trabajo se acepta, cuál espera y cuál se rechaza educadamente con HTTP 429. Timeouts, reintentos, circuit breakers, bulkheads y degradación elegante se ocupa después del fallo entre servicios: timeouts y plazos, reintentos con backoff y jitter y los presupuestos que evitan las tormentas de reintentos, circuit breakers, bulkheads, alternativas (fallbacks), idempotencia y solicitudes de cobertura (hedged requests). Su ejemplo práctico muestra cómo la falta de un solo timeout permite que una única dependencia lenta tumbe todo lo que la llama.

Operar el sistema: objetivos, observabilidad, pruebas y aprendizaje

La segunda mitad pasa de los mecanismos a la operación. SLI, SLO, SLA y presupuestos de errores muestra cómo expresar la fiabilidad desde el punto de vista del usuario, cómo un presupuesto de errores convierte un objetivo de disponibilidad en una cantidad de fallo permitido que los equipos pueden gastar en cambios, y cómo las alertas por tasa de consumo (burn rate) avisan a las personas solo cuando ese presupuesto está realmente en peligro. Observabilidad trata de los logs, las métricas y las trazas que permiten a los ingenieros hacer preguntas nuevas a un sistema en producción, los métodos RED y USE y las cuatro señales doradas, el trazado distribuido con OpenTelemetry y el costo de los datos de alta cardinalidad.

Planificación de capacidad y pruebas de rendimiento cierra el círculo entre las previsiones y la evidencia: pruebas de carga, de estrés, de resistencia (soak), de picos y de punto de ruptura, modelos de carga abiertos y cerrados, la omisión coordinada (coordinated omission) y un plan de capacidad para una puesta a la venta de entradas. Ingeniería del caos, incidentes y aprendizaje sin culpas se ocupa de los fallos que ocurren de todos modos: experimentar deliberadamente en producción para encontrar las debilidades antes que los clientes, gestionar los incidentes con roles claros y celebrar revisiones sin culpas cuyas acciones cambian el sistema y no a las personas.

Cómo encajan las secciones

Las secciones forman una cadena. El escalado y el balanceo de carga añaden capacidad; las colas y el back pressure deciden qué ocurre cuando, aun así, la capacidad se agota; los patrones de resiliencia impiden que un problema local se convierta en uno global. Los SLO dicen cuándo algo es lo bastante bueno, la observabilidad muestra si el sistema cumple ese nivel y por qué no, las pruebas y la planificación de capacidad predicen cuándo dejará de cumplirlo, y la ingeniería del caos y las revisiones de incidentes encuentran las lagunas de todo lo anterior. La mayoría de los mecanismos son baratos de añadir en la fase de diseño y caros de incorporar después, y por eso son arquitectura y no un detalle de operaciones: si un servicio es sin estado, si una llamada es idempotente, si una cola está acotada y si una solicitud lleva un plazo se decide mucho antes del primer incidente.

El capítulo presupone los estilos distribuidos del Capítulo 6 y la arquitectura de datos del Capítulo 7. Remite a ellos para las claves de idempotencia, las garantías de entrega, la replicación y la caché en lugar de repetirlos, y adelanta el Capítulo 10 para las plataformas en la nube, las funcionalidades de Kubernetes y los diseños multirregión sobre los que suelen ejecutarse estos sistemas.

La idea que debes llevarte

Todo recurso es finito, así que todo sistema tiene un punto de ruptura; el trabajo del arquitecto es decidir de antemano qué ocurre cuando se alcanza. Un sistema bien diseñado sometido a una carga excesiva se ralentiza de forma elegante, descarta primero el trabajo menos importante, rechaza con claridad en lugar de agotar el tiempo en silencio, aísla la parte que falla y se recupera por sí solo cuando pasa la presión. Uno mal diseñado encola sin límite, reintenta hasta derrumbarse y arrastra consigo a sus vecinos. La diferencia está hecha de pequeñas decisiones deliberadas (un timeout aquí, una cola acotada allá, un presupuesto de reintentos, una comprobación de preparación) y del hábito de medir, probar y aprender del sistema tal como se ejecuta en realidad.

SECCIÓN 9.1 DE 9.8 · INTERMEDIO · 28 MIN DE LECTURA

Estrategias de escalado: vertical, horizontal y el cubo de escalado

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.7 Servidores y la nube
- 2.4 Rendimiento y escalabilidad
- 6.1 Por qué distribuir

Términos anteriores que utiliza: Balanceador de carga, Ley de Little, Ley de Amdahl, Utilización, Réplica de lectura

LA IDEA EN BREVE

Piensa en una cafetería con mucho movimiento. Puedes hacer que atienda a más clientes comprando una máquina de café más grande y más rápida (escalar verticalmente) o abriendo más mostradores con más personal (escalar horizontalmente). Escalar horizontalmente llega más lejos, pero solo funciona si cualquier mostrador puede atender a cualquier cliente, de modo que el pedido de nadie puede quedar guardado en la cabeza de un solo barista. También puedes dividir el trabajo, con un mostrador para las bebidas y otro para la comida, o una cafetería por barrio. Y como el personal adicional tarda en llegar, un gerente sensato lo programa antes de la hora punta del almuerzo en lugar de llamarlo cuando la cola ya sale por la puerta.

LO QUE NECESITAS SABER

- El escalado vertical añade recursos a una sola máquina sin cambiar el software, pero choca con un techo de tamaño y de precio y deja un único dominio de fallo; el escalado horizontal añade instancias, escala más lejos y tolera fallos, pero necesita un trabajo sin estado y divisible.
- El cubo de escalado (Abbott y Fisher) separa tres formas de escalar horizontalmente: X clona copias idénticas, Y divide por función en servicios y Z divide por datos para que cada copia atienda a una parte de los clientes o de los registros.
- La ausencia de estado es la condición previa del escalado en el eje X: las sesiones, los archivos, las cachés usadas como fuente de verdad, las tareas programadas, los bloqueos y los contadores deben pasar a almacenes compartidos o viajar con la solicitud.

- El autoescalado puede ser reactivo, programado o predictivo; el escalado reactivo va minutos por detrás de la carga, así que los cambios bruscos necesitan margen de capacidad, colas o capacidad programada de antemano.
- Escala según una señal ligada al recurso que se agota, normalmente las solicitudes en curso por instancia o el trabajo acumulado en cola por trabajador; la CPU engaña en los servicios limitados por la E/S, y la latencia engaña cuando una dependencia compartida es lenta.
- Evita el flapping escalando hacia fuera deprisa y hacia dentro despacio, con ventanas de estabilización, bandas de tolerancia y recuentos mínimos y máximos, y dimensiona las flotas para el pico con una zona perdida, usando la ley de Little.
- Escalar la capa de aplicación traslada la presión a la capa de datos; revisa las conexiones y las filas calientes, y recurre al ajuste de consultas, la caché, las réplicas, los poolers de conexiones y el escalado vertical antes que al particionamiento.

Qué significa escalar en la práctica

La Sección 2.4 definió la escalabilidad como la capacidad de gestionar más carga añadiendo recursos, y mostró con la ley de Amdahl y la ley de escalabilidad universal por qué añadir recursos no siempre añade capacidad. Esta sección trata de las decisiones prácticas: *qué* recursos añadir, *dónde*, *cuándo* y *con cuánta automatización*. El punto de partida es siempre la misma pregunta: ¿cuál es el parámetro de carga y qué recurso se agota primero? La API de pedidos de una plataforma de reparto de comida a la hora del almuerzo está limitada por las solicitudes por segundo; su servicio de despacho, por el número de repartidores activos cuyas posiciones hay que emparejar cada pocos segundos; su base de datos, por las transacciones de escritura por segundo y por las conexiones. Cada uno necesita una respuesta distinta.

Escalar un sistema significa en realidad escalar su cuello de botella actual y después encontrar el siguiente. Duplicar la capa web de un sistema cuya base de datos está ocupada al 95% no aporta nada salvo una factura mayor y una base de datos a la que ahora se presiona más. Por eso toda decisión de escalado empieza por la medición: la utilización y la saturación de cada recurso (el método USE de Brendan Gregg, que se trata en Observabilidad) y una prueba de carga que encuentre dónde empieza a subir la latencia (Planificación de capacidad y pruebas de rendimiento).

Escalado vertical: una máquina más grande

El escalado vertical (scaling up) consiste en dar a la máquina existente más núcleos de CPU, más memoria, discos más rápidos o una red más rápida. Su gran virtud es que no cambia nada en el software. Una única base de datos relacional en un servidor más grande conserva sus transacciones, sus joins y su consistencia sencilla; una aplicación conserva sus cachés en

memoria y sus archivos locales. En la nube, redimensionar es una llamada a una API seguida de un reinicio. Las mayores máquinas virtuales en la nube ofrecen hoy cientos de CPU virtuales y terabytes de memoria, mucho más de lo que la mayoría de los sistemas de negocio necesitarán jamás, y algunos sitios web con muchísimo tráfico han explicado públicamente que atienden un tráfico enorme con un pequeño número de potentes servidores de bases de datos.

Los límites, sin embargo, son reales. Existe una máquina más grande que todas, y el precio por unidad de capacidad sube de forma pronunciada cerca de lo más alto de la gama. Una máquina es un dominio de fallo: cuando falla, o se reinicia para redimensionarla o aplicarle parches, todo lo que hay en ella se detiene, así que escalar verticalmente por sí solo no hace nada por la disponibilidad. A menudo el software no puede aprovechar del todo una máquina más grande: un proceso de un solo hilo, un bloqueo global o una fila caliente de la base de datos limitan la aceleración exactamente como predice la ley de Amdahl. Y la capacidad llega en escalones grandes y lentos, así que pagas el pico a todas horas. Aun así, el escalado vertical es el primer paso correcto con mucha más frecuencia de lo que sugiere la moda, sobre todo en las bases de datos, donde escalar horizontalmente es caro en esfuerzo de diseño.

Escalado horizontal: más máquinas

El escalado horizontal (scaling out) consiste en ejecutar más instancias y repartir el trabajo entre ellas, normalmente con un balanceador de carga delante. Escala mucho más lejos, en escalones pequeños y baratos, y la misma redundancia que añade capacidad también tolera los fallos: con ocho instancias, perder una cuesta un octavo de la capacidad y no toda. Las plataformas en la nube convierten añadir una instancia en cuestión de segundos o minutos, y esa es la base del autoescalado.

El precio es que el software debe estar diseñado para ello. Cualquier instancia debe poder gestionar cualquier solicitud, lo que significa que las instancias no deben guardar nada de lo que dependa una solicitud posterior (la siguiente sección). El trabajo debe ser divisible: una tarea que necesita un bloqueo global no se ejecuta más deprisa en diez máquinas. Los datos deben ser accesibles desde todas las instancias, lo que traslada la presión a la capa de datos compartida. Y el sistema pasa a ser distribuido, con los fallos parciales, los timeouts y los reintentos que describe la Sección 6.1. La mayoría de los sistemas reales combinan ambos: las capas de aplicación sin estado escalan horizontalmente, mientras que las bases de datos escalan verticalmente mientras pueden hacerlo de forma razonable y después escalan horizontalmente con réplicas y particiones.

Escalado vertical y horizontal comparados

Inquietud	Vertical (scale up)	Horizontal (scale out)
Cambios en el software	Ninguno	Las instancias deben ser sin estado y el trabajo, divisible
Límite superior	La mayor máquina disponible	Normalmente, la capa de datos compartida o los costos de coordinación
Curva de costos	Sube de forma pronunciada en los tamaños más grandes	Aproximadamente lineal, en escalones pequeños
Fallo (failure)	Un solo dominio de fallo; redimensionar exige un reinicio	Perder una instancia supone perder una fracción de la capacidad
Velocidad del cambio	Minutos más un reinicio	De segundos a minutos; puede ser automático
Consistencia y transacciones	Sin cambios y sencillas	Sin cambios en las capas sin estado; difícil para los datos
Uso típico	Bases de datos, software heredado y con estado, primera respuesta al crecimiento	Capas web y de API, trabajadores, todo lo que no tenga estado

El cubo de escalado: tres formas distintas de escalar horizontalmente

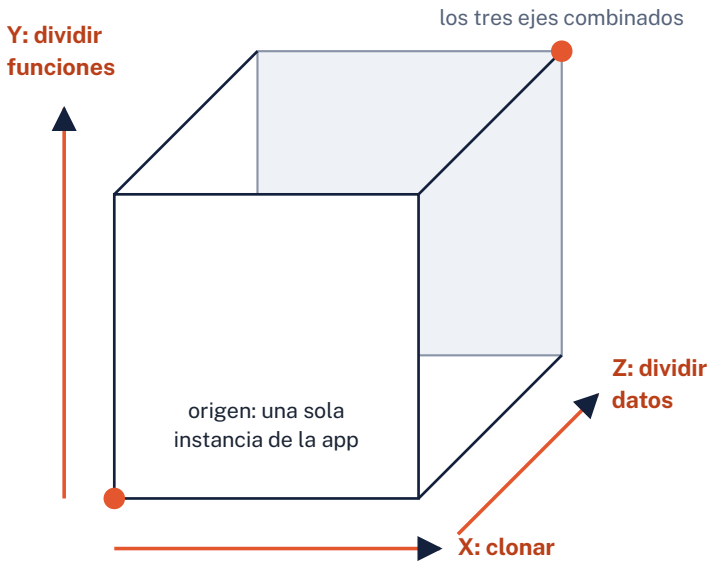
Martin Abbott y Michael Fisher, que dirigieron la tecnología en eBay y PayPal, describen el escalado horizontal con el cubo de escalado en *The Art of Scalability* (primera edición de 2009, segunda edición de 2015). Separa tres formas independientes de dividir un sistema, cada una dibujada como un eje de un cubo cuyo origen es una única instancia que lo hace todo.

El **eje X** es la duplicación horizontal, o clonación: ejecutar N copias idénticas de toda la aplicación o servicio y repartir las solicitudes entre ellas. Es el eje más sencillo y barato, y funciona bien para la gestión de solicitudes sin estado y para el tráfico de lectura. No hace nada por el tamaño de los datos ni por el volumen de escritura, porque todos los clones siguen hablando con los mismos datos, y tampoco hace nada por el tamaño del código base ni por el número de equipos que trabajan en él.

El **eje Y** divide el sistema por función: servicios distintos para las cartas, los pedidos, el despacho y los pagos, cada uno de los cuales ejecuta solo el código y los datos de su capacidad. Así, cada servicio puede escalar, fallar y desplegarse por separado, y cada uno es más pequeño y más fácil de entender. Este es el eje que hay detrás de los microservicios (Sección 6.3), con todos sus costos. El **eje Z** divide por datos: todas las copias ejecutan el

mismo código, pero cada una es responsable solo de una parte de los clientes, inquilinos (tenants), regiones o registros, elegida mediante una clave como la ciudad o el identificador del cliente. El sharding de bases de datos es el eje Z aplicado al almacenamiento (Sección 7.4); una arquitectura *basada en células* lo aplica a pilas completas, de modo que un defecto o un despliegue erróneo en una célula afecta solo a los clientes de esa célula.

El cubo de escalado



Eje X: clonar
N copias idénticas tras un balanceador, cada una hace todo el trabajo: 14 API de pedidos

Eje Y: dividir por función
Servicios separados por capacidad: cartas, pedidos, despacho, pagos

Eje Z: dividir por datos
Mismo código; cada copia tiene parte de clientes o datos: una célula o shard por ciudad

El cubo de escalado de Abbott y Fisher: cada eje es una forma distinta de dividir un sistema, y los sistemas reales suelen combinar los tres.

Los tres ejes en la plataforma de reparto de comida

Eje	Qué se divide	Ejemplo	Resuelve	Costos y límites
X: clonar	Las solicitudes, entre copias idénticas	14 instancias de la API de pedidos detrás de un balanceador de carga	Rendimiento de solicitudes, fallos de instancias	Exige ausencia de estado; todos los clones comparten una base de datos
Y: función	Código y datos, por capacidad	Servicios separados de carta, pedidos, despacho y pagos	Necesidades distintas de escalado y de fallo; tamaño de los equipos	Llamadas de red, datos distribuidos, sobrecarga operativa
Z: datos	Clientes o registros, por clave	Una célula por grupo de ciudades, o pedidos fragmentados (sharding) por ciudad	Tamaño de los datos, volumen de escritura, radio de impacto	Capa de enrutamiento, consultas entre partes, reequilibrado, claves calientes

Los ejes se combinan. El servicio de pedidos de la plataforma (una división en el eje Y) se ejecuta como muchos clones (X) en cada célula regional (Z). El consejo práctico de Abbott y Fisher es usar el eje más barato que resuelva el problema real: X para el volumen de solicitudes, Y cuando partes del sistema tienen necesidades de escalado, de fallo o de equipo realmente distintas, y Z cuando un conjunto de datos o un radio de impacto se ha vuelto demasiado grande. Elegir Z porque suena impresionante trae el enrutamiento, el reequilibrado y las consultas entre particiones mucho antes de que hagan falta.

La ausencia de estado: la condición previa para clonar

El escalado en el eje X solo funciona si cualquier clon puede atender cualquier solicitud. Un servicio sin estado no guarda nada entre solicitudes en su propia memoria ni en su propio disco; todo lo que necesita una solicitud posterior vive en un almacén compartido o viaja con la solicitud. El estado que suele esconderse en una instancia es fácil de enumerar en cuanto lo buscas: sesiones de usuario guardadas en memoria, archivos subidos escritos en el disco local, cachés en memoria que se tratan como fuente de verdad, tareas programadas que deben ejecutarse exactamente en una máquina, bloqueos dentro del proceso que protegen un recurso compartido y contadores para los límites de tasa.

Cada uno tiene un remedio estándar: las sesiones pasan a un almacén compartido o a un token firmado (Sección 9.2); los archivos pasan al almacenamiento de objetos; las cachés se convierten en una optimización de rendimiento que puede perderse en cualquier momento, o

pasan a una caché compartida (Sección 7.6); las tareas programadas usan un planificador con elección de líder o una cola; los bloqueos pasan a la base de datos o a un servicio de coordinación; los contadores de límite de tasa pasan a un almacén compartido. La regla de la aplicación de doce factores según la cual los procesos son sin estado y no comparten nada (Sección 10.3) es esta misma idea. La ausencia de estado es barata de incorporar en el diseño y cara de añadir después, porque el estado se filtra en el código en muchos sitios pequeños.

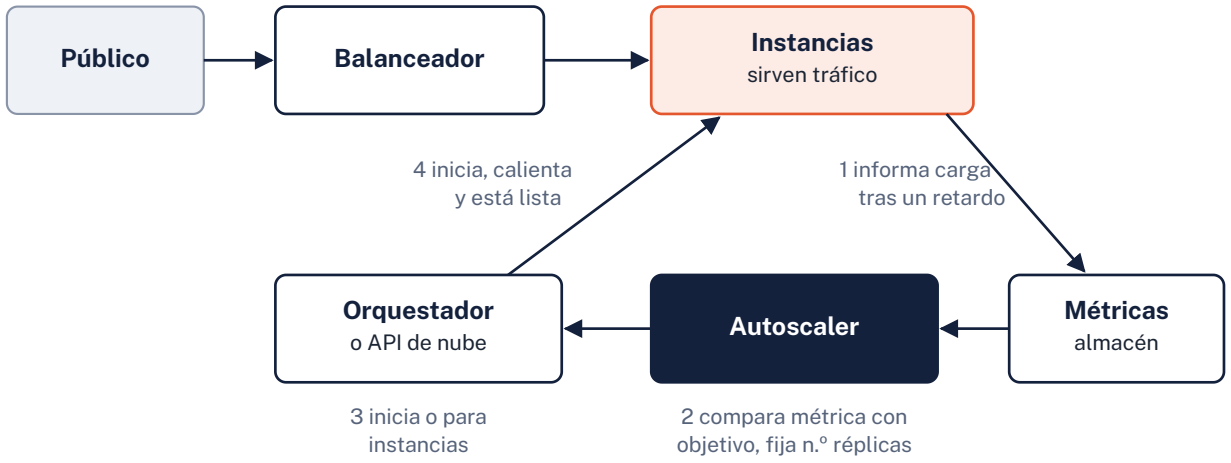
Autoescalado: reactivo, programado y predictivo

El autoescalado cierra un bucle de retroalimentación: mide la carga, la compara con un objetivo y cambia el número de instancias. El autoescalado **reactivo** lo hace de forma continua a partir de una métrica. El seguimiento de objetivo (target tracking) es la forma habitual: mantener la CPU media al 60%, o las solicitudes en curso por instancia en 30. El Horizontal Pod Autoscaler de Kubernetes calcula `desired = ceil(current replicas × current metric ÷ target)`: seis pods con una media de 45 solicitudes en curso frente a un objetivo de 30 pasan a ser ceil(6 × 45 ÷ 30) = 9. De forma predeterminada, vuelve a evaluar cada 15 segundos e ignora las desviaciones de hasta el 10% respecto al objetivo.

El autoescalado **programado** cambia la capacidad en momentos conocidos: subir el mínimo a 18 instancias a las 11:15 todos los días laborables antes del pico del almuerzo y bajarlo a las 14:00. Es rudimentario y muy eficaz para patrones predecibles, y es la única manera de tener la capacidad lista *antes* de un cambio brusco, como una puesta a la venta de entradas a las 10:00 en punto. El autoescalado **predictivo**, que ofrecen los principales proveedores de nube, prevé la carga a partir del historial (ciclos diarios y semanales) y escala por delante de ella, con el escalado reactivo como red de seguridad para las sorpresas.

La propiedad crucial de cualquier esquema reactivo es el retraso. Las métricas se recogen y se promedian durante decenas de segundos; el autoescalador actúa en su siguiente evaluación; una máquina virtual nueva tarda un minuto o más en arrancar, hay que descargar una imagen de contenedor, y un servicio JVM o Python debe iniciarse, calentar sus cachés y sus pools de conexiones y superar su comprobación de preparación antes de que el balanceador de carga le envíe tráfico. En un clúster, puede que primero haya que añadir los propios nodos mediante un segundo autoescalador, más lento. Desde el momento en que sube la carga hasta que la nueva capacidad es útil suelen pasar de dos a diez minutos. Por tanto, el autoescalado reactivo gestiona los cambios graduales; los cambios repentinos deben absorberse con margen de capacidad (headroom), con colas (Sección 9.3) o con capacidad programada de antemano.

El bucle de retroalimentación del autoescalado y de dónde viene el retraso



De subida de carga a capacidad útil: a menudo 2 a 10 min

Cada paso del bucle añade retraso, así que el autoescalado reactivo sigue a la carga con minutos de retraso; el margen de capacidad y la capacidad programada cubren el hueco.

Qué métrica usar para escalar, y por qué la CPU suele ser un error

La utilización de la CPU es la señal predeterminada en la mayoría de las plataformas porque siempre está disponible. Solo es la señal correcta cuando el servicio está limitado por la CPU: redimensionado de imágenes, cifrado, serialización intensiva. La mayoría de los servicios de negocio no lo están. Una API de pedidos pasa la mayor parte de cada solicitud esperando a la base de datos, al proveedor de pagos y a otros servicios; su CPU puede estar al 25% mientras todos los hilos de trabajo están ocupados esperando y las solicitudes nuevas se encolan. Escalar según la CPU no hace entonces nada hasta que es demasiado tarde. También ocurre el fallo contrario: una tormenta de recolección de basura o el calentamiento del JIT elevan la CPU sin que aumente la carga, y el autoescalador añade instancias que no hacen falta.

Una señal mejor es la que está directamente ligada al recurso que se agota. En los servicios de solicitud-respuesta, suele ser la **concurrencia**: el número de solicitudes en curso por instancia, comparado con el número de hilos de trabajo o de conexiones de que dispone. La ley de Little hace exacta la relación (en curso = tasa de llegada × tiempo de respuesta), y reacciona tanto a más tráfico como a respuestas más lentas. En los trabajadores de colas, es el **trabajo acumulado por trabajador** o la antigüedad del mensaje más antiguo, que expresa cuánto tardará la cola en vaciarse. La latencia es tentadora, pero peligrosa como señal de escalado: sube por razones que más instancias no pueden arreglar, como una base de datos lenta, y escalar horizontalmente un servicio cuya dependencia está en apuros envía a esa dependencia todavía más carga. Usa la latencia para las alertas y escala según una medida del trabajo.

Señales de escalado y cuándo engañan

Señal	Adecuada para	Engaña cuando
Utilización de la CPU	Trabajo limitado por la CPU: procesamiento de imágenes, cifrado, serialización intensiva	El trabajo espera a la E/S, a hilos o a conexiones; la recolección de basura o el calentamiento inflan la CPU
Solicitudes por segundo por instancia	Servicios cuyas solicitudes cuestan más o menos lo mismo	Cambia la mezcla de solicitudes, por ejemplo más búsquedas y menos lecturas de la carta
Solicitudes en curso por instancia	La mayoría de los servicios de solicitud-respuesta; se corresponde con los límites de hilos y de conexiones	Una dependencia aguas abajo se ralentiza; escalar horizontalmente le añade entonces carga, así que limita el máximo
Trabajo acumulado en cola por trabajador, o antigüedad del mensaje más antiguo	Trabajadores asíncronos	Los trabajadores están limitados por un límite de tasa aguas abajo y no por su número
Latencia	Alertas y SLO (Sección 9.5)	La causa es una dependencia compartida que más instancias no pueden arreglar
Métrica de negocio (pedidos por minuto, sesiones activas)	Carga con una relación estable y medida con la capacidad	La relación se desvía a medida que cambia el producto; recalibra con pruebas de carga

SI ESTO ES NUEVO PARA TI

Paso a paso: hilos ocupados, procesador inactivo

La idea más difícil de esta sección es que un servidor puede estar lleno mientras su procesador parece casi inactivo, así que la utilización de la CPU puede ser la señal equivocada para escalar.

- Una instancia de la API del historial de pedidos tiene 48 hilos de trabajo, así que puede trabajar como máximo en 48 solicitudes a la vez. Cada solicitud pasa la mayor parte de su tiempo esperando: a las consultas a la base de datos y a las respuestas del servicio de la carta.
- Un hilo que espera casi no usa tiempo de procesador, pero sigue ocupado: no puede empezar otra solicitud hasta que vuelva la respuesta.
- Cada solicitud retiene un hilo durante unos 0.14 segundos. Según la ley de Little, 250 solicitudes por segundo mantienen ocupados unos 35 hilos (250 × 0.14). A 450 por segundo harían falta 63, más de los 48 que hay, así que las solicitudes se encolan y los tiempos de respuesta suben.
- Sin embargo, el procesador puede estar en torno al 28%. Un autoescalador configurado para mantener la CPU al 65% no ve nada raro y no añade instancias mientras los clientes esperan.
- Contar las solicitudes en curso por instancia muestra el problema directamente: la cifra sube hacia 48. Escalar según esa cifra, con un objetivo de unas 36 (tres cuartas partes de los hilos), empieza a añadir instancias mientras todavía quedan hilos libres.
- La CPU sigue siendo adecuada para el trabajo que realmente es cálculo, como redimensionar imágenes. Y la cifra de solicitudes en curso también sube cuando es la propia base de datos la que está en apuros; entonces, más instancias solo le envían más carga, y por eso la sección también fija un máximo.
- **La lección:** escala según el recurso que realmente se agota. En la mayoría de los servicios de negocio son los hilos o las conexiones, así que mide el trabajo en curso, no cuánto se esfuerza el procesador.

Cooldowns, estabilización y flapping

Un autoescalador que reacciona a cada vaivén oscila (flapping): añade instancias cuando llega una ráfaga, las quita un minuto después cuando la ráfaga pasa y luego las vuelve a añadir, pagando cada vez el costo de arranque y alterando las cachés y los pools de conexiones. También puede perseguir su propia cola: las instancias nuevas están frías y son lentas, así que la latencia o la CPU por solicitud suben brevemente después de escalar horizontalmente, lo que parece más carga.

Las defensas habituales son asimétricas. Escala hacia fuera deprisa, porque la falta de capacidad perjudica a los usuarios; escala hacia dentro despacio, porque el exceso de capacidad solo cuesta dinero. Un cooldown o ventana de estabilización hace que el autoescalador espere después de una acción, o que tome la recomendación más alta de una ventana reciente antes de reducir: el autoescalador de Kubernetes usa de forma predeterminada una ventana de cinco minutos (300 segundos) para reducir y ninguna para ampliar. Una banda de tolerancia ignora las desviaciones pequeñas. Los recuentos mínimo y máximo de instancias acotan el bucle: el mínimo mantiene capacidad suficiente para sobrevivir a la pérdida de una zona y a los primeros minutos de una avalancha, y el máximo protege la base de datos, los servicios aguas abajo y el presupuesto frente a un autoescalador que confunde un problema con carga. La reducción también debe ser elegante: una instancia elegida para retirarse debe dejar de recibir solicitudes nuevas y terminar las que tiene en curso (drenaje de conexiones, Sección 9.2).

EJEMPLO PRÁCTICO

Ejemplo práctico: dimensionar la API de pedidos para el pico del almuerzo

La API de pedidos de la plataforma de reparto de comida debe soportar el almuerzo de los días laborables. En las últimas ocho semanas, el minuto de más actividad llegó a 2050 solicitudes por segundo (rps) hacia las 12:30; marketing espera un crecimiento del 15% este trimestre. Una prueba de carga de una instancia (4 vCPU, 32 hilos de trabajo) encontró un tiempo de respuesta medio de 120 ms y una latencia que empezaba a subir hacia los 260 rps. El servicio se ejecuta en tres zonas de disponibilidad y debe sobrevivir a la pérdida de una zona en el pico.

Pico previsto λ	2050 × 1.15 ≈ 2360, redondeado al alza a 2400 rps
Ley de Little para la flota	L = λ × W = 2400 × 0.12 s = 288 solicitudes en curso en el pico
Capacidad de una instancia	32 hilos ÷ 0.12 s ≈ 267 rps, lo que concuerda con el codo que encontró la prueba de carga (unos 260 rps)
Utilización objetivo	70% de 32 hilos = 22.4 solicitudes en curso por instancia (unos 187 rps), lo que deja un 30% de margen de capacidad para las ráfagas
Instancias para el pico	288 ÷ 22.4 = 12.9, así que 13 instancias (2400 ÷ 13 ≈ 185 rps cada una)
Sobrevivir a la pérdida de una zona	13 repartidas 5/4/4: perder 5 deja 8 instancias con 288 ÷ 8 = 36 en curso cada una, más que los 32 hilos: sobrecarga. Permitir hasta el 85% (27.2 en curso) durante la pérdida de una zona exige 288 ÷ 27.2 = 10.6, así que 11 en dos zonas: 6 por zona, 18 en total
Rampa y retraso del escalado horizontal	La demanda sube de 600 rps a las 11:30 a 2400 a las 12:15: 40 rps por minuto. El escalado horizontal reactivo tarda unos 5 minutos (1 minuto en detectar, 4 en arrancar y calentarse), durante los cuales la demanda sube 200 rps; la capacidad sobrante al 70% es de unos 80 rps por instancia, así que el escalado reactivo por sí solo podría seguir el ritmo de esta rampa gradual; lo que no puede dar es protección frente a la pérdida de una zona
Decisión	El escalado programado sube el mínimo a 18 (6 por zona) a las 11:15 y lo baja a 6 (2 por zona) a las 14:00; el escalado reactivo según las solicitudes en curso por instancia (objetivo 22) se mantiene como red de seguridad, con un máximo de 30
Comprobación de la base de datos	18 instancias × un pool de 10 conexiones = 180; con el máximo de 30 instancias, 300. La base de datos admite 200, así que el equipo añade un pooler de conexiones antes de subir el máximo

La ley de Little convierte una tasa prevista y un tiempo de respuesta medido en una cifra de concurrencia que se puede comparar directamente con los límites de hilos y de conexiones.

Es el requisito de pérdida de zona, y no la carga media, el que fija el tamaño de la flota en el pico: 18 instancias en lugar de 13. El autoescalado reactivo no puede dar esta protección porque sustituir una zona perdida lleva minutos.

Escalar horizontalmente la capa de aplicación multiplicó la carga sobre las conexiones de la base de datos compartida, que es el siguiente cuello de botella. Todo ejercicio de dimensionamiento debería terminar comprobando la capa de debajo.

Escalar la capa de datos

Las capas de aplicación sin estado son fáciles de escalar; para eso se hacen sin estado. La presión recae entonces sobre las bases de datos, las cachés y los intermediarios que comparten, y por eso la capa de datos suele ser el verdadero límite de un sistema. Las herramientas se trataron en el Capítulo 7 y aquí se resumen en el orden en que los profesionales suelen recurrir a ellas. Primero, reduce el trabajo: índices, mejores consultas, menos idas y vueltas por solicitud. Segundo, absorbe las lecturas con la caché (Sección 7.6) y las réplicas de lectura, aceptando el retardo de replicación cuando el caso de uso lo permita. Tercero, protege las conexiones: cada instancia de la aplicación tiene un pool, así que el total crece con la flota, y un pooler de conexiones como PgBouncer permite que miles de conexiones de clientes compartan unos pocos cientos de conexiones a la base de datos (el `max_connections` predeterminado de PostgreSQL es de solo 100). Cuarto, escala verticalmente la base de datos principal, lo que a menudo es más barato que cualquier rediseño. Solo entonces particiona las escrituras en el eje Z según una clave como la ciudad o el cliente (Sección 7.4), o pasa a una base de datos SQL distribuida.

Hay dos advertencias. Primera, autoescalar la capa de aplicación puede perjudicar a la capa de datos: cada instancia nueva abre conexiones y calienta su caché desde la base de datos, así que un gran escalado horizontal puede sobrecargar por sí mismo la base de datos. Segunda, algunos límites son lógicos, no físicos: una sola fila caliente (el recuento de existencias de las últimas 500 entradas de un concierto, o un número de secuencia global) serializa las escrituras por grande que sea el servidor. Esos límites se eliminan con el diseño, por ejemplo dividiendo las existencias en varios contadores o reservando asientos por lotes, no con hardware.

GUÍA DE DECISIÓN

Guía de decisión: cómo añadir capacidad a un componente

Recorre las preguntas en orden para el componente que limita hoy el sistema.

Q1 **¿Se ha medido el cuello de botella (utilización, saturación y encolamiento de CPU, memoria, conexiones, bloqueos y llamadas aguas abajo) en lugar de suponerlo?**

- **Sí** Continúa con el componente que se satura primero.
- **No** Haz primero una prueba de carga y mide. Escalar la capa equivocada añade costo y ninguna capacidad, y puede empujar más carga hacia el verdadero cuello de botella.

Q2 **¿Se puede abaratar el propio trabajo (un índice, una caché, menos idas y vueltas, una carga útil más pequeña, menos logging)?**

- **Sí** Haz eso primero. Reduce el costo a cualquier escala y a menudo elimina por completo la necesidad de escalar.
- **No** Continúa: hay que añadir capacidad.

Q3 **¿El componente es sin estado, o se puede trasladar su estado (sesiones, archivos, bloqueos en memoria, programaciones) a almacenes compartidos?**

- **Sí** Escala horizontalmente en el eje X detrás de un balanceador de carga, con autoescalado según una señal de carga y margen de capacidad suficiente para el retraso del escalado horizontal.
- **No** Escala verticalmente por ahora y planifica el trabajo para externalizar el estado; una única instancia con estado es también un punto único de fallo.

Q4 **¿Está el límite en la capa de datos: la tasa de escritura, el tamaño de almacenamiento o el número de conexiones de una base de datos?**

- **Sí** Usa la caché y las réplicas de lectura para las lecturas, un pooler de conexiones para las conexiones y el particionamiento en el eje Z (shards o células) para las escrituras y el almacenamiento.
- **No** Continúa.

Q5 **¿Tienen partes del componente necesidades de carga, de escalado o de fallo muy distintas?**

- **Sí** Considera una división en el eje Y para que cada parte escale y falle por separado, aceptando los costos de la distribución descritos en el Capítulo 6.
- **No** Quédate con la clonación en el eje X: es el eje más barato de operar.

ATENCIÓN

Errores comunes

- Escalar la capa que es fácil de escalar en lugar de la que está saturada, normalmente añadiendo servidores web delante de una base de datos que ya es el cuello de botella.
- Autoescalar según la CPU servicios limitados por la E/S, de modo que la capacidad llega solo después de que la latencia se haya desplomado.
- Confiar en el autoescalado reactivo para cambios bruscos, como puestas a la venta, emisiones o campañas de marketing, que llegan más deprisa de lo que pueden arrancar las instancias.
- No fijar un número máximo de instancias, de modo que una dependencia lenta se confunde con carga y el autoescalador multiplica las conexiones hacia ella.
- Dimensionar para el día medio en lugar de para el pico con una zona o una instancia perdida.
- Suponer que las instancias son sin estado sin comprobar si hay sesiones en memoria, archivos locales, tareas programadas de una sola instancia y bloqueos dentro del proceso.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ La CPU de una API de pedidos está al 30%, pero la latencia p99 se ha triplicado y todos los hilos de trabajo están ocupados. ¿Por qué no ha reaccionado el autoescalador basado en la CPU y qué señal lo habría hecho?

El servicio está esperando a la E/S, así que los hilos se agotan mientras la CPU está inactiva. Las solicitudes en curso por instancia, comparadas con el número de hilos, habrían subido con la carga.

▼ ¿Qué eje del cubo de escalado aborda una base de datos que ha superado la capacidad de escritura de un solo servidor?

El eje Z: particionar los datos según una clave para que cada shard o célula gestione una parte de las escrituras.

▼ ¿Por qué los autoescaladores suelen escalar hacia fuera deprisa, pero hacia dentro despacio?

La falta de capacidad perjudica a los usuarios de inmediato, mientras que el exceso solo cuesta dinero, y reducir con demasiado entusiasmo provoca flapping cuando vuelve la carga.

▼ **¿Por qué el autoescalado reactivo no puede proteger una puesta a la venta de entradas que empieza exactamente a las 10:00?**

La demanda se dispara en segundos, pero detectar la subida, arrancar instancias y calentarlas lleva minutos, así que la capacidad debe programarse antes de que empiece la venta.

FUENTES

- Kubernetes documentation: Horizontal Pod Autoscaling: https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/
- Chris Richardson: The Scale Cube (microservices.io): https://microservices.io/articles/scalecube.html
- The Twelve-Factor App: VI. Processes: https://12factor.net/processes
- Brendan Gregg: The USE method: https://brendangregg.com/usemethod.html

SECCIÓN 9.2 DE 9.8 · INTERMEDIO · 27 MIN DE LECTURA

Balanceo de carga, ausencia de estado y sesiones

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.4 Comunicación en red
- 7.4 Replicación y particionamiento
- 9.1 Escalado

Términos anteriores que utiliza: Hashing consistente, DNS, TCP, gRPC

LA IDEA EN BREVE

Imagina un supermercado con un empleado que dirige a los clientes hacia las cajas. Enviar a la gente a cada caja por turno es justo pero ingenuo: un cliente con un carro para una familia de seis personas bloquea una caja mientras la siguiente está libre. Enviar a cada cliente a la caja con la cola más corta funciona mucho mejor. El empleado también necesita saber qué cajas están abiertas, debe dejar de enviar gente a una caja que está a punto de cerrar y dejar que su cola termine, y su trabajo funciona mejor si cualquier caja puede atender a cualquier cliente, para que a nadie se le diga que vuelva al mismo cajero que tiene su tarjeta de fidelización.

LO QUE NECESITAS SABER

- Un balanceador de carga reparte las conexiones o las solicitudes entre las instancias sanas, oculta los cambios de instancias tras una sola dirección y a menudo termina TLS; puede estar en el borde o dentro de los clientes y los sidecars.
- Los balanceadores de capa 4 reenvían conexiones según la dirección y el puerto, rápidos y ajenos al protocolo; los de capa 7 leen cada solicitud HTTP y pueden enrutar, reintentar y limitar la tasa, algo que exigen los protocolos por solicitud como gRPC sobre HTTP/2.
- El round robin ignora el costo de las solicitudes; menos conexiones y el poder de dos elecciones aleatorias (Mitzenmacher) se adaptan a la carga, y las dos elecciones aleatorias evitan el efecto manada que provocan muchos balanceadores cuando todos persiguen la instancia menos cargada.
- Los fallos de actividad (liveness) reinician una instancia y los fallos de preparación (readiness) solo la retiran de la rotación; deja las dependencias compartidas fuera de las comprobaciones de actividad para que un tropiezo de la base de datos no pueda reiniciar toda la flota.

- El drenaje de conexiones deja de enviar solicitudes nuevas a una instancia que se retira y permite que termine el trabajo en curso, así que los despliegues y las reducciones no provocan errores.
- Las sesiones persistentes provocan una carga desigual y la pérdida de sesiones cuando hay fallos; traslada el estado de sesión a tokens firmados, a un almacén compartido o a la base de datos para que cualquier instancia pueda atender cualquier solicitud.
- El balanceo de carga global dirige a los usuarios hacia las regiones mediante las respuestas DNS, limitadas por la caché y los TTL, o mediante anycast, que anuncia una sola dirección desde muchos sitios y deja que el enrutamiento elija el más cercano.

Qué hace un balanceador de carga

Un balanceador de carga se sitúa entre los clientes y un pool de instancias intercambiables y decide, para cada conexión o solicitud, qué instancia la recibe. Repartir es solo una parte del trabajo. Un balanceador de carga también comprueba qué instancias están sanas y deja de enviar tráfico a las que no lo están; da a los clientes una dirección estable mientras las instancias van y vienen con los despliegues y el autoescalado; a menudo termina TLS, de modo que los certificados se gestionan en un solo sitio; y en la capa 7 puede enrutar, reintentar, limitar la tasa y registrar cada solicitud. Es, por tanto, el componente que hace que el escalado horizontal sea invisible para los clientes.

El balanceo de carga aparece en varios niveles. En el borde suele haber un balanceador de carga gestionado en la nube o un par de proxies como HAProxy, NGINX o Envoy. Entre servicios, el balanceo se hace cada vez más en el lado del cliente: una biblioteca cliente de gRPC o un proxy sidecar de una service mesh (Sección 6.3) mantiene la lista de instancias sanas obtenida del descubrimiento de servicios y elige una por sí mismo, lo que elimina un salto de red y un componente central. Sea cual sea la forma, se repiten las mismas decisiones: qué puede ver el balanceador, qué algoritmo usa, cómo decide que una instancia está sana y cómo salen las instancias sin perder trabajo.

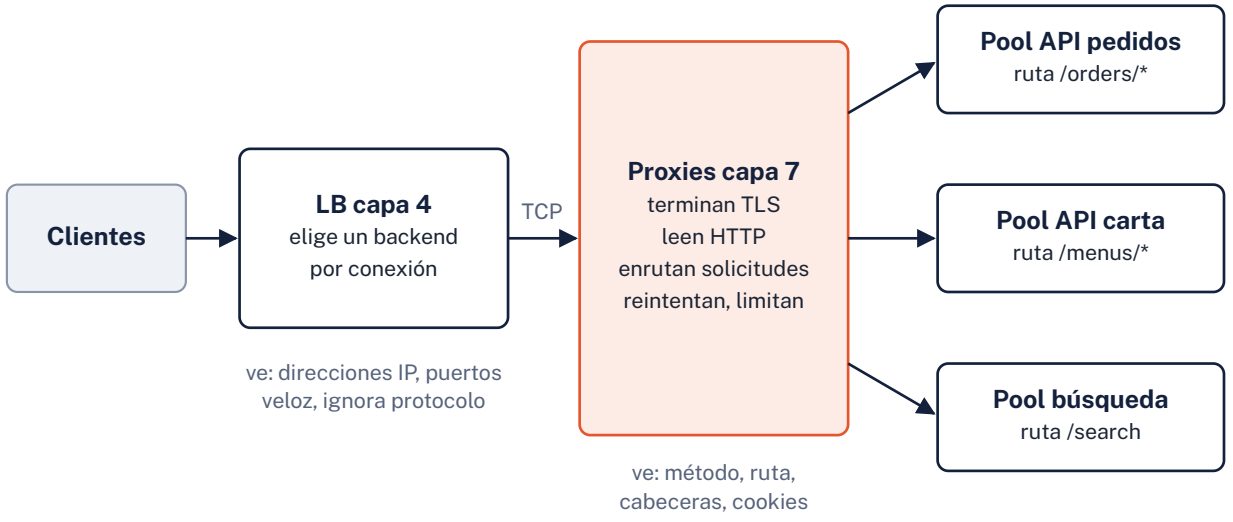
Balanceo de carga de capa 4 y de capa 7

Los números de capa proceden del modelo de red OSI. Un balanceador de carga de capa 4 trabaja en la capa de transporte: ve direcciones IP y puertos TCP o UDP, elige un backend cuando se abre una conexión y después reenvía los paquetes de esa conexión sin mirar dentro de ellos. Como hace tan poco, es extremadamente rápido y funciona con cualquier protocolo, incluidas las bases de datos, los intermediarios de mensajes y el tráfico cifrado que no puede leer. Maglev, de Google, y los balanceadores de carga de red de las principales nubes son diseños de capa 4 que gestionan millones de paquetes por segundo.

Un balanceador de carga de capa 7 es un proxy inverso que entiende el protocolo de aplicación, casi siempre HTTP. Termina la conexión del cliente (y normalmente TLS), lee cada solicitud y abre o reutiliza sus propias conexiones con los backends. Eso le permite enrutar por ruta, host, cabecera o cookie (`/orders/*` al pool de pedidos, `/menus/*` al pool de la carta), balancear cada solicitud en lugar de cada conexión, reintentar una solicitud fallida en otra instancia, aplicar límites de tasa, añadir cabeceras de trazado y registrar cada solicitud con su estado y su latencia. El costo es más procesamiento por solicitud, un componente que debe guardar las claves TLS y un sitio más en el que hay que configurar los timeouts y los reintentos de forma coherente con el resto del sistema (Sección 9.4).

Hay una diferencia que pilla desprevenidos a muchos equipos. HTTP/2 y gRPC multiplexan muchas solicitudes sobre una sola conexión de larga duración. Un balanceador de capa 4 elige un backend una vez por conexión, así que un cliente que abre una conexión envía todas sus solicitudes a una sola instancia, por muchas instancias que existan: las instancias nuevas que añade el autoescalado no reciben nada hasta que los clientes se vuelven a conectar. Por eso, balancear el tráfico gRPC exige un proxy de capa 7 o un balanceo en el lado del cliente por solicitud.

Una disposición típica en dos niveles



Un nivel de capa 4 reparte las conexiones a bajo costo; un nivel de capa 7 lee cada solicitud y la enruta por ruta al pool adecuado.

Algoritmos de balanceo de carga

El algoritmo decide qué instancia sana recibe la siguiente conexión o solicitud. Los sencillos ignoran lo ocupadas que están las instancias; los mejores usan alguna medida de la carga; los basados en hash envían deliberadamente las solicitudes relacionadas al mismo sitio. La tabla

resume las opciones habituales.

Algoritmos de balanceo de carga habituales

Algoritmo	Cómo elige	Ventajas	Puntos débiles
Round robin	Cada instancia por turno	Sencillo, predecible, sin estado sobre la carga	Ciego al costo de las solicitudes y a las instancias lentas
Round robin ponderado	Por turno, en proporción a pesos configurados	Mezcla tamaños de instancia; aumenta gradualmente el tráfico de las versiones nuevas (canarios)	Los pesos son estimaciones estáticas; sigue siendo ciego a la carga real
Aleatorio	Al azar, de forma uniforme	No necesita estado compartido, así que muchos balanceadores pueden usarlo de forma independiente	Desigual por azar, sobre todo con pocas solicitudes
Menos conexiones (least connections) (menos solicitudes pendientes)	La instancia con menos solicitudes en curso	Se adapta a costos de solicitud desiguales y a instancias lentas	Necesita recuentos precisos; cada uno de muchos balanceadores solo ve los suyos; las instancias nuevas pueden verse inundadas
Ponderado por latencia (por ejemplo, EWMA del tiempo de respuesta)	La instancia con la mejor latencia reciente, a menudo combinada con la carga	Se aparta enseguida de las instancias degradadas	Puede oscilar; una instancia que falla deprisa parece atractiva salvo que se penalicen los errores
Hash (IP de origen, cabecera, hashing consistente)	La misma clave va siempre a la misma instancia	Afinidad para cachés y sesiones; el hashing consistente mueve pocas claves cuando cambia el pool	Las claves calientes sobrecargan una instancia; distribución desigual de las claves
Poder de dos elecciones aleatorias (power of two choices)	Elige dos instancias al azar y envía a la menos cargada	Casi tan bueno como menos conexiones, sin efecto manada, con poca coordinación	Algo peor que una elección global perfecta con información perfecta

EJEMPLO PRÁCTICO

Ejemplo práctico: por qué menos conexiones supera al round robin cuando los costos de las solicitudes difieren

Dos instancias, A y B, procesan cada una una solicitud a la vez (las instancias reales tienen más hilos, pero el efecto es el mismo a mayor escala). Llegan ocho solicitudes separadas por 10 ms. La primera es un informe de ventas de un restaurante que cuesta 100 ms; las otras siete son lecturas de la carta que cuestan 10 ms cada una.

Round robin: asignación	A recibe las solicitudes 1, 3, 5 y 7; B recibe la 2, 4, 6 y 8, sea cual sea su costo
Round robin en A	R1 se ejecuta de 0 a 100 ms. R3 (llegó en 20) espera y termina en 110: 90 ms. R5 (40) termina en 120: 80 ms. R7 (60) termina en 130: 70 ms
Round robin en B	R2, R4, R6 y R8 terminan cada una 10 ms después de llegar; B está inactiva la mitad del tiempo
Resultado del round robin	Lecturas de la carta: 90, 80, 70, 10, 10, 10, 10 ms, media 290 ÷ 7 ≈ 41 ms, peor caso 90 ms. Las ocho solicitudes: media (100 + 290) ÷ 8 ≈ 49 ms
Menos conexiones: asignación	R1 va a A (ambas inactivas). A los 10 ms, A tiene 1 solicitud en curso y B tiene 0, así que R2 va a B; B siempre vuelve a estar libre antes de la siguiente llegada, así que de R3 a R8 van todas a B
Resultado de menos conexiones	Cada lectura de la carta tarda 10 ms; el informe tarda 100 ms. Media (100 + 70) ÷ 8 ≈ 21 ms, peor lectura de la carta 10 ms
Por qué	El round robin cuenta solicitudes, no trabajo. Menos conexiones cuenta el trabajo en curso, que, según la ley de Little (en curso = tasa × tiempo), crece en la instancia que sea más lenta o esté más ocupada, así que las solicitudes nuevas se alejan de ella automáticamente

El mismo efecto aparece siempre que los costos de las solicitudes varían mucho (búsquedas frente a consultas puntuales, exportaciones frente a lecturas) o que una instancia es más lenta que sus pares por un vecino ruidoso, una pausa de recolección de basura o una caché fría.

Menos conexiones necesita un recuento significativo. Con HTTP/2 o gRPC muchas solicitudes comparten una conexión, así que los balanceadores cuentan en su lugar las solicitudes pendientes; con conexiones WebSocket de larga duración, el recuento mide sesiones, no carga.

Una instancia recién arrancada tiene cero conexiones y puede verse inundada antes de que sus cachés estén calientes; muchos balanceadores ofrecen un periodo de arranque lento (slow start) que aumenta su parte gradualmente durante un minuto más o menos.

El poder de dos elecciones aleatorias

Si menos conexiones funciona tan bien, ¿por qué no elegir siempre la instancia menos cargada a nivel global? Porque en un sistema grande no existe una visión única y actual de la carga. Decenas de balanceadores de carga o miles de balanceadores en el lado del cliente ven cada uno solo sus propias solicitudes, o un informe de carga de hace unos segundos. Si todos eligen la instancia que parecía menos cargada, todos eligen *la misma*, que enseguida se ve desbordada, y luego todos pasan a la siguiente: una manada que persigue la carga por todo el pool.

La tesis doctoral de Michael Mitzenmacher (1996), publicada como *The Power of Two Choices in Randomized Load Balancing* (IEEE Transactions on Parallel and Distributed Systems, 2001) y basada en trabajos anteriores de Azar, Broder, Karlin y Upfal, analizó una alternativa sorprendentemente sencilla: elegir dos instancias al azar y enviar la solicitud a la menos cargada de las dos. En el modelo clásico de bolas en urnas, colocar n solicitudes en n servidores al azar da un servidor más cargado con unas log n ÷ log log n solicitudes; con dos elecciones aleatorias esto baja a unas log log n ÷ log 2, una mejora exponencial, mientras que una tercera elección añade muy poco más. El azar evita el efecto manada y la comparación evita las peores instancias.

El poder de dos elecciones aleatorias (power of two choices) es hoy la opción predeterminada o una opción estándar en proxies muy utilizados: el balanceador least-request de Envoy compara de forma predeterminada dos hosts elegidos al azar, y NGINX ofrece `random two least_conn`. Es una buena opción predeterminada para el balanceo en el lado del cliente en cualquier sistema con muchos balanceadores.

Hashing y afinidad

A veces lo que se busca es enviar las solicitudes relacionadas a la *misma* instancia: a un nodo de caché que ya tiene la carta de un restaurante, o a la instancia que mantiene una sesión WebSocket. Aplicar un hash a una clave (identificador del restaurante, identificador del usuario, IP de origen) para asignarla a una instancia lo consigue. Un simple `hash mod N` reasigna la mayoría de las claves cada vez que cambia N, lo que vacía todas las cachés en el peor momento, durante un escalado horizontal. El hashing consistente (presentado en la Sección 7.4) mueve solo alrededor de 1 ÷ N de las claves cuando una instancia entra o sale. Su

punto débil son las claves calientes: el restaurante más popular a la hora del almuerzo cae en una sola instancia. Variantes como el hashing consistente con cargas acotadas limitan la parte de cada instancia y desvían el exceso a la siguiente instancia del anillo.

Comprobaciones de estado: actividad y preparación

Un balanceador de carga debe saber qué instancias pueden recibir tráfico. Una comprobación de estado (health check) activa sondea cada instancia periódicamente, por ejemplo `GET /healthz` cada cinco segundos, y la marca como caída tras unos cuantos fallos consecutivos y de nuevo como disponible tras unos cuantos éxitos. Las comprobaciones pasivas, a menudo llamadas detección de valores atípicos (outlier detection), observan el tráfico real y expulsan una instancia que devuelve, por ejemplo, cinco errores de servidor consecutivos. Las dos se complementan: las comprobaciones activas encuentran instancias que no reciben tráfico, y las pasivas reaccionan a fallos que una sonda trivial no ejercita.

Kubernetes hizo explícita una distinción importante. Una sonda de actividad (liveness probe) pregunta si el proceso está atascado sin posibilidad de recuperarse (en un interbloqueo, sin memoria en un bucle); si falla, la instancia se reinicia. Una sonda de preparación (readiness probe) pregunta si la instancia debe recibir tráfico en este momento; si falla, la instancia se retira del balanceador de carga pero sigue en ejecución, por ejemplo mientras calienta sus cachés al arrancar o mientras descarta carga. Una tercera, la sonda de arranque (startup probe), mantiene en suspenso las otras dos mientras arranca una aplicación lenta.

El error clásico es comprobar las dependencias compartidas en estas sondas. Si la sonda de actividad de cada instancia llama a la base de datos, una breve caída de la base de datos hace que fallen todas las sondas, el orquestador reinicia todas las instancias a la vez y la tormenta de reinicios (cachés frías, miles de conexiones nuevas) prolonga la caída mucho después de que la base de datos haya vuelto. Las comprobaciones de preparación que incluyen una dependencia compartida pueden igualmente marcar toda la flota como no preparada, convirtiendo errores parciales en una caída total. Mantén la sonda de actividad local y mínima; deja que la de preparación refleje la capacidad de la propia instancia para atender; y gestiona los fallos de las dependencias con los patrones de la Sección 9.4. Los balanceadores de carga se protegen del caso extremo fallando en abierto: un AWS Application Load Balancer enruta a todos los destinos si todos fallan sus comprobaciones de estado, y Envoy ignora el estado de salud cuando la proporción de hosts sanos cae por debajo de un umbral de pánico (50% de forma predeterminada), con el razonamiento de que enviar tráfico a instancias dudosas es mejor que no enviarlo a ninguna parte.

SI ESTO ES NUEVO PARA TI

Paso a paso: una comprobación que lo reinicia todo

La idea más difícil de esta sección es que las dos comprobaciones de estado hacen cosas distintas, y que poner una dependencia compartida en la equivocada puede convertir un tropiezo en una caída. Una sonda de actividad fallida hace que se reinicie una instancia; una sonda de preparación fallida solo la saca de la rotación del balanceador de carga.

- Supón que la API de pedidos se ejecuta como 30 instancias, cada una con una sonda de actividad que solo informa de que está sana si puede llegar al intermediario de mensajes compartido.
- El intermediario está inaccesible durante alrededor de un minuto mientras se sustituye un nodo. Todas las sondas fallan, aunque las propias instancias están bien.
- Tras unos cuantos fallos seguidos, el orquestador concluye que todas las instancias están atascadas sin posibilidad de recuperarse y reinicia las 30 a la vez.
- El intermediario vuelve, pero ahora 30 instancias recién arrancadas se vuelven a conectar a él a la vez y rellenan cachés vacías desde la base de datos de pedidos. Nadie puede hacer un pedido mientras se calientan, y la tormenta de reinicios golpea a ambos, convirtiendo un tropiezo de un minuto en una caída larga.
- Si la sonda de actividad hubiera comprobado solo la propia instancia, nada se habría reiniciado: las instancias habrían conservado sus cachés calientes y habrían seguido atendiendo; solo habría esperado el trabajo que necesitaba el intermediario.
- La preparación también requiere cuidado. Si la comprobación de preparación de cada instancia también llama a ese intermediario, toda la flota se marca como no preparada a la vez y el balanceador de carga no tiene adónde enviar nada. Por eso algunos balanceadores de carga fallan en abierto: cuando todas las instancias parecen no estar sanas, les envían tráfico a todas de todos modos.
- **La lección:** mantén la sonda de actividad local y mínima, deja que la de preparación refleje si esta instancia puede atender en este momento, y gestiona el fallo de una dependencia compartida con los patrones de la Sección 9.4, no reiniciando la flota.

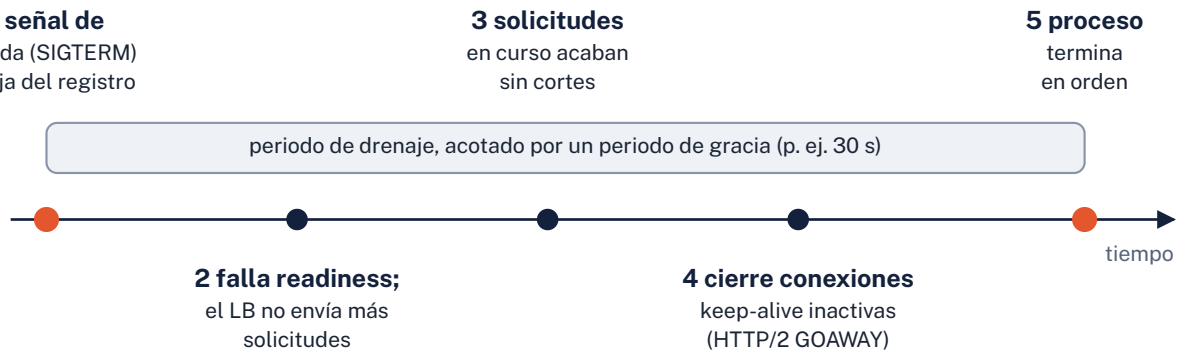
Drenaje de conexiones y apagado elegante

Las instancias salen del pool constantemente: cada despliegue las sustituye, el autoescalado las retira y la plataforma las mueve durante el mantenimiento. Si una instancia simplemente se mata, todas las solicitudes que estaba procesando fallan, y los clientes ven errores durante lo que debería ser un lanzamiento rutinario. El drenaje de conexiones (connection draining)

(llamado deregistration delay en AWS) retira la instancia de la rotación del balanceador para que no reciba solicitudes *nuevas*, espera a que terminen las solicitudes que ya están en curso y solo entonces la deja detenerse.

La instancia debe colaborar. Al recibir su señal de apagado, debe fallar su comprobación de preparación, seguir atendiendo las solicitudes en curso, indicar a los clientes con conexiones keep-alive que vayan a otro sitio (una cabecera `Connection: close` en HTTP/1.1, una trama GOAWAY en HTTP/2), terminar o devolver el trabajo en segundo plano y salir antes de que acabe el periodo de gracia. En Kubernetes, retirar un pod del balanceador de carga y enviarle SIGTERM ocurren a la vez, así que todavía puede llegar una solicitud justo después de la señal; una breve espera previa a la parada (pre-stop) de unos segundos cubre esa carrera. El periodo de gracia predeterminado es de 30 segundos, después de los cuales se mata el proceso. Las solicitudes de larga duración (subidas grandes, respuestas en streaming, WebSockets) necesitan un periodo de gracia más largo o un protocolo con el que los clientes se vuelvan a conectar en otro sitio sin perder trabajo.

Apagado elegante en cinco pasos



El drenaje convierte la retirada de una instancia, que antes era una fuente de errores, en algo sin consecuencias: no llega trabajo nuevo, el trabajo existente se completa y después el proceso sale.

Las sesiones persistentes y sus problemas

Las aplicaciones web más antiguas guardaban la sesión de cada usuario (quién ha iniciado sesión, qué hay en el carrito) en la memoria del servidor que la había creado. Detrás de un balanceador de carga, esto solo funciona si todas las solicitudes de un usuario vuelven a ese servidor, así que los balanceadores ofrecen sesiones persistentes (sticky sessions) (afinidad de sesión), normalmente mediante una cookie que identifica la instancia. La persistencia es una forma rápida de poner una aplicación con estado detrás de un balanceador, y tiene un uso legítimo como pista de rendimiento, por ejemplo para mantener a un usuario en una instancia cuya caché local ya contiene sus datos.

Como mecanismo de corrección, causa problemas. La carga se vuelve desigual, porque unos pocos usuarios intensivos o una gran oficina detrás de una sola dirección IP se quedan fijados a una instancia sea cual sea su carga. Cuando una instancia falla, sus usuarios pierden sus sesiones: los carritos desaparecen y las personas ven cerrada su sesión. Las reducciones y cada despliegue hacen lo mismo salvo que se drenen las sesiones, lo que puede tardar tanto como la sesión más larga. Las instancias nuevas que añade el autoescalado solo reciben usuarios nuevos, así que alivian despacio a las instancias sobrecargadas. Y el sistema ya no es sin estado en el sentido de la Sección 9.1, así que todos estos problemas reaparecen en el failover entre zonas o regiones.

Externalizar el estado de sesión

El remedio es sacar el estado de sesión de la instancia, para que cualquier instancia pueda atender cualquier solicitud. Hay tres opciones principales, que a menudo se combinan. Un **token en el lado del cliente** lleva el estado con cada solicitud: una cookie firmada (y, si hace falta, cifrada) o un token como un JWT que contiene la identidad del usuario y algunos datos. Los servidores no necesitan ningún almacén, pero los tokens deben ser pequeños, no se pueden revocar fácilmente antes de que caduquen y deben protegerse contra la manipulación y el robo (Sección 8.3). Un **almacén de sesiones compartido**, normalmente Redis o Memcached, guarda las sesiones en un almacén externo rápido, indexadas por un identificador de sesión que va en la cookie; admite un estado más grande y el cierre de sesión inmediato, a costa de una llamada de red por solicitud y de una dependencia que a su vez debe tener alta disponibilidad. Una **base de datos** es adecuada para el estado que debe sobrevivir a cualquier cosa, como un carrito que el usuario espera encontrar mañana, que en realidad son datos de negocio y no datos de sesión.

Un reparto práctico para la plataforma de reparto de comida: la identidad, en un token firmado de corta duración; el carrito, en la base de datos del servicio de pedidos (los usuarios pasan del teléfono a la computadora portátil); y el estado transitorio de la interfaz, en el cliente. Ninguna instancia guarda nada que necesite una solicitud posterior, así que el balanceador de carga puede usar un balanceo por menos solicitudes sin afinidad, y las instancias se pueden añadir, retirar y sustituir a voluntad.

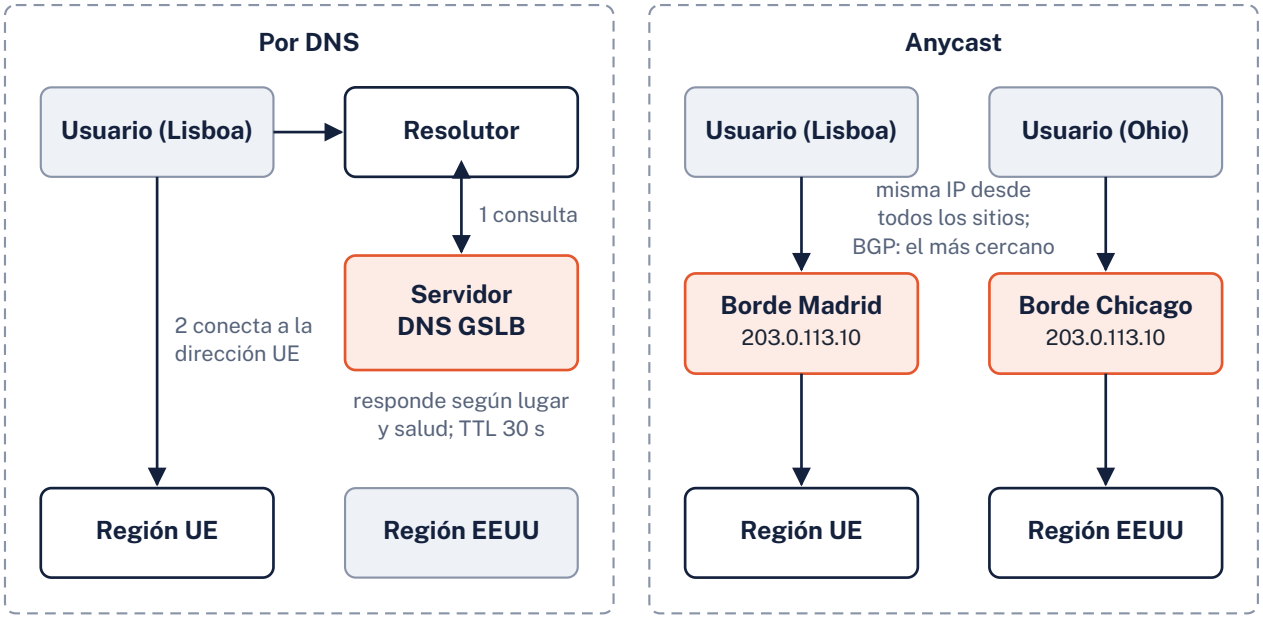
Balanceo de carga global: DNS y anycast

Cuando un sistema se ejecuta en varias regiones, algo debe enviar a cada usuario a una región adecuada: la región sana más cercana o la región que tiene sus datos. El balanceo de carga global suele hacerse de una de dos formas, que a menudo se combinan.

El balanceo global de servidores **basado en DNS** usa un servidor de nombres autoritativo que responde al mismo nombre con direcciones distintas según el origen de la consulta, la salud de cada región y unos pesos configurados: a un usuario de Lisboa se le da la dirección de la región de la UE, y a uno de Ohio, la de la región de EE. UU. No necesita ninguna red especial, pero es tosco y lento en reaccionar. Los resolutores guardan las respuestas en caché durante al menos el tiempo de vida (TTL) del registro (a menudo de 30 a 300 segundos) y a veces más, así que, después de que falle una región, algunos usuarios siguen llegando a ella durante minutos. El servidor de nombres ve la ubicación del resolutor y no la del usuario, aunque la extensión EDNS Client Subnet (RFC 7871) permite que los resolutores transmitan parte de la dirección del cliente.

Con anycast, la misma dirección IP se anuncia desde muchos sitios mediante el protocolo de enrutamiento de internet (BGP), y los paquetes de cada usuario se entregan al sitio que esté más cerca en términos de enrutamiento. El failover consiste en retirar el anuncio de un sitio que ha fallado, lo que surte efecto enseguida sin esperar a las cachés de DNS. Anycast se usa desde hace mucho en los servicios raíz y de resolución de DNS; las CDN y los balanceadores de carga globales de los grandes proveedores de nube lo usan para llevar a los usuarios a un sitio de borde cercano, donde proxies de capa 7 terminan la conexión y reenvían las solicitudes por la red del proveedor a una región sana. El diseño multirregión, la ubicación de los datos y el failover son el tema de la Sección 10.6.

Balanceo de carga global basado en DNS y con anycast



El DNS dirige a los usuarios respondiendo con direcciones distintas; anycast da a todos la misma dirección y deja que el enrutamiento de internet los lleve al sitio más cercano.

ATENCIÓN

Errores comunes

- Poner tráfico gRPC o HTTP/2 detrás de un balanceador de capa 4 y preguntarse por qué las instancias nuevas no reciben solicitudes.
- Usar round robin en servicios cuyas solicitudes varían mucho en costo, o en los que algunas instancias son más lentas que otras.
- Llamar a la base de datos desde las sondas de actividad, de modo que un tropiezo de la base de datos reinicia toda la flota.
- Matar instancias sin drenarlas, de modo que cada despliegue y cada reducción producen una ráfaga de errores.
- Confiar en las sesiones persistentes para la corrección y perder después carritos e inicios de sesión cada vez que una instancia falla o se sustituye.
- Suponer que el failover por DNS es instantáneo; los clientes y los resolutores pueden seguir usando una respuesta antigua durante minutos.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué un servidor gRPC recién añadido detrás de un balanceador de carga de capa 4 podría no recibir tráfico?

El balanceo de capa 4 elige un backend por conexión, y los clientes gRPC mantienen una sola conexión HTTP/2 de larga duración, así que los clientes existentes nunca le envían solicitudes hasta que se vuelven a conectar.

▼ ¿Qué diferencia de efecto hay entre fallar una sonda de actividad y fallar una sonda de preparación?

Fallar la de actividad hace que el orquestador reinicie la instancia; fallar la de preparación solo la retira del balanceador de carga hasta que vuelva a superarla.

▼ ¿Por qué funcionan mal muchos balanceadores de carga independientes que eligen la instancia menos cargada?

Actúan sobre la misma visión desfasada y todos eligen la misma instancia, que se sobrecarga por turnos; elegir la mejor de dos instancias aleatorias evita este efecto manada.

▼ **Da dos problemas que causa guardar las sesiones en la memoria de la instancia con sesiones persistentes.**

Los usuarios pierden sus sesiones cuando la instancia falla o se sustituye, y la carga se vuelve desigual porque los usuarios intensivos se quedan fijados a una instancia.

FUENTES

- Michael Mitzenmacher: The Power of Two Choices in Randomized Load Balancing (IEEE TPDS, 2001): https://www.eecs.harvard.edu/~michaelm/postscripts/tpds2001.pdf
- Kubernetes documentation: Liveness, Readiness, and Startup Probes: https://kubernetes.io/docs/concepts/configuration/liveness-readiness-startup-probes/
- Envoy documentation: Supported load balancers: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancers
- NGINX documentation: HTTP load balancing: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/

SECCIÓN 9.3 DE 9.8 · INTERMEDIO · 29 MIN DE LECTURA

Asincronía, colas y back pressure

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.4 Rendimiento y escalabilidad
- 6.4 Dirigida por eventos
- 6.5 Comunicación y API

Términos anteriores que utiliza: Ley de Little, Idempotente, Acoplamiento temporal, Rendimiento (throughput)

LA IDEA EN BREVE

Una panadería concurrida a la hora de abrir reparte tickets numerados: los clientes esperan en una fila ordenada en lugar de agolparse en el mostrador, y los panaderos trabajan a un ritmo constante. Eso es una cola. Pero una fila solo puede ser tan larga antes de que se acabe el pan o la gente se canse de esperar, así que una buena panadería también deja de repartir tickets cuando la fila es demasiado larga, atiende primero los encargos, limita a cada cliente a un número razonable de barras y dice con franqueza a los que llegan tarde que vuelvan en diez minutos, en lugar de dejarlos esperar para nada.

LO QUE NECESITAS SABER

- Una cola entre productor y consumidor convierte un pico en un trabajo acumulado (backlog) que los consumidores vacían a su propio ritmo, lo que permite dimensionarlos para la media; el costo es el retraso de cada elemento que queda detrás del backlog.
- Los consumidores competidores escalan el trabajo en segundo plano añadiendo workers; la entrega es al menos una vez (at-least-once), así que los consumidores deben ser idempotentes, y los timeouts de visibilidad y el prefetch deben ajustarse a los tiempos de procesamiento.
- Una cola no puede absorber un exceso sostenido; limita cada cola con la ley de Little (longitud máxima = tasa de procesamiento × espera aceptable) y monitoriza la antigüedad del mensaje más antiguo además de la longitud.
- El back pressure hace que las etapas más rápidas vayan más despacio; cuando una cola acotada está llena, el sistema debe bloquear al productor, rechazar el trabajo nuevo o descartar parte del trabajo, y debería elegir de forma deliberada.

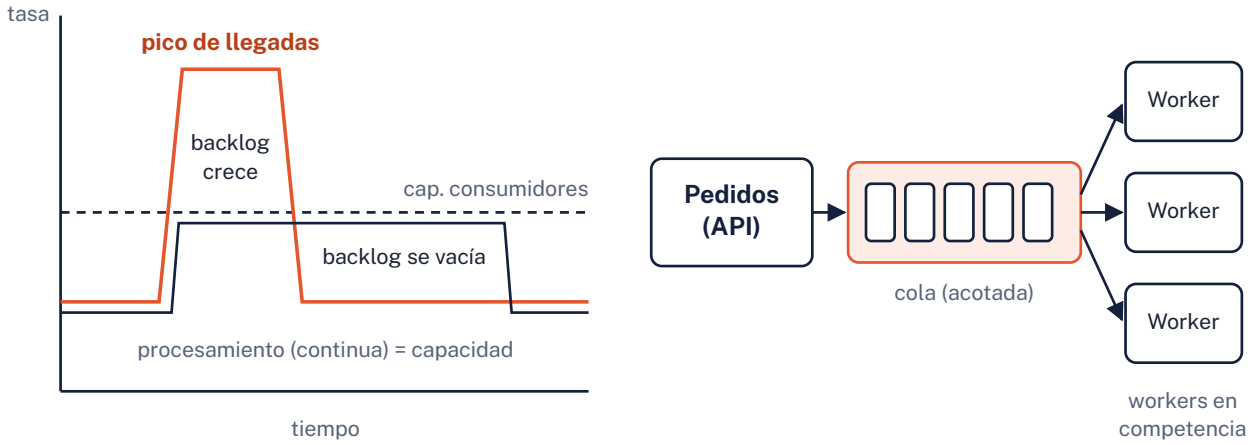
- El descarte de carga (load shedding) rechaza parte del trabajo pronto y a bajo costo, por prioridad y por antigüedad, para que el goodput se mantenga alto en situaciones de sobrecarga.
- La limitación de tasa (rate limiting) reparte la capacidad por cliente: el token bucket permite ráfagas hasta su capacidad y una tasa de recarga constante, el leaky bucket suaviza la salida, y las ventanas fijas permiten ráfagas dobles en los cambios de ventana.
- Responde a los límites con 429 (o 503 cuando todo el servidor está sobrecargado) más Retry-After, y envía los mensajes que fallan repetidamente a una cola de mensajes fallidos (dead-letter queue) con un responsable y monitorizada.

Las colas convierten los picos en trabajo acumulado

La carga nunca llega de forma uniforme. Los pedidos del almuerzo se concentran a las 12:30, los compradores de entradas pulsan todos el botón a las 10:00:00, y un trabajo por lotes vuelca un millón de eventos a medianoche. Un servicio síncrono debe tener capacidad para el pico más alto o fallar durante él. Poner una cola entre quien produce el trabajo y el servicio que lo realiza cambia la aritmética: durante un pico, el trabajo sobrante espera en la cola, y los consumidores lo procesan a su propio ritmo constante, y se ponen al día cuando el pico ha pasado. El catálogo de patrones de nube de Microsoft llama a esto nivelación de carga basada en colas (queue-based load levelling). La Sección 2.4 mostró por qué se forman colas siempre que las llegadas se concentran; esta sección trata de usarlas de forma deliberada.

El beneficio es real: los consumidores pueden dimensionarse para la media en lugar de para el pico, un consumidor lento o que se reinicia no pierde trabajo, y el productor ya no está acoplado temporalmente al consumidor (Sección 6.5). El precio es que el trabajo ahora se hace más tarde. Una cola convierte un pico en un *trabajo acumulado (backlog)*, y un backlog es simplemente un retraso que sufrirá cada elemento que quede detrás. Eso está bien para enviar correos electrónicos de confirmación o actualizar un índice de búsqueda, es discutible para asignar un repartidor a un pedido, e inaceptable para un pago cuya confirmación espera ver el cliente. Por eso, la primera pregunta de diseño para cualquier cola es: ¿cuánto tiempo puede esperar un elemento y qué ocurre con los elementos que esperarían más?

Una cola que absorbe un pico



Durante el pico, la cola absorbe las llegadas que superan la capacidad de los consumidores; después, los consumidores vacían el backlog a su ritmo constante.

Colas de trabajo y consumidores competidores

El uso más habitual es una cola de trabajo con consumidores competidores: varios workers idénticos leen de una misma cola, cada mensaje va solo a uno de ellos, y el rendimiento (throughput) crece añadiendo workers. Es el eje X del cubo de escalado aplicado al trabajo en segundo plano, y escala de forma natural con la señal adecuada: el backlog por worker o la antigüedad del mensaje más antiguo.

La mecánica importa para la corrección. Un worker toma un mensaje, lo procesa y después confirma su recepción; si el worker se cae antes, el intermediario vuelve a hacer visible el mensaje tras un timeout (SQS lo llama visibility timeout, timeout de visibilidad) y otro worker lo procesa. Por tanto, la entrega es al menos una vez, así que los consumidores deben ser idempotentes (Sección 6.4). Un timeout de visibilidad más corto que el tiempo real de procesamiento hace que el mismo mensaje se procese dos veces en paralelo. El orden se pierde en cuanto hay más de un consumidor, salvo que los mensajes se particionen por clave, en cuyo caso el número de particiones limita el número de consumidores útiles. Y el prefetch (que un worker tome varios mensajes a la vez para ahorrarse viajes de ida y vuelta) debe mantenerse bajo para las tareas lentas, o un worker acaparará mensajes que podrían haber procesado otros workers ociosos.

Una cola no crea capacidad

Una cola absorbe un exceso temporal; no puede absorber uno permanente. Si el trabajo llega más rápido de lo que se completa durante el tiempo suficiente, el backlog crece sin límite, y la ley de Little ($L = \lambda \times W$) dice que el tiempo de espera crece con él. Una cola en memoria sin límite acaba agotando la memoria y haciendo caer el proceso, con lo que se pierde todo lo que contiene; una cola duradera sin límite llena su disco o conserva trabajo durante horas después de que a nadie le importe. Peor aún, la cola oculta el problema: los productores reciben confirmaciones rápidas, las tasas de error se quedan en cero, y el único síntoma es que los resultados llegan cada vez más tarde. Los ingenieros de Amazon describen estos backlogs como potencialmente insuperables, porque un servicio que se queda lo bastante atrás puede gastar toda su capacidad en procesar trabajo que sus clientes abandonaron hace tiempo.

De ahí se siguen dos prácticas. Primero, mide la antigüedad del mensaje más antiguo además de la longitud de la cola: la antigüedad es lo que experimentan los usuarios, y reacciona de inmediato cuando los consumidores se atascan. Segundo, da a cada cola un límite derivado del tiempo de espera que estás dispuesto a aceptar, usando la ley de Little, y decide de antemano qué ocurre cuando se alcanza el límite. Esa decisión es el tema de las tres partes siguientes de esta sección.

EJEMPLO PRÁCTICO

Ejemplo práctico: dimensionar la cola de pedidos para una salida a la venta de entradas

Las entradas para un concierto en un estadio salen a la venta a las 10:00. Los envíos de pedidos van a una cola; los workers autorizan el pago y asignan los asientos, y cada uno tarda 250 ms. La previsión es de 600 envíos por segundo durante los dos primeros minutos. La promesa a los clientes es una respuesta en menos de dos minutos. Hay 25 000 entradas y el pedido medio es de 2.5 entradas.

Capacidad de los workers con 40 workers	40 ÷ 0.25 s = 160 pedidos por segundo
Backlog tras el pico de dos minutos	(600 − 160) × 120 s = 52 800 pedidos en espera
Espera del último pedido de la cola (ley de Little, W = L ÷ λ)	52 800 ÷ 160 = 330 s, cinco minutos y medio: la promesa se incumple
Cola más larga que cumple la promesa	L = λ × W = 160 × 120 = 19 200 pedidos
¿Más workers?	El contrato del proveedor de pagos permite 300 solicitudes por segundo, así que como máximo 75 workers (300 pedidos por segundo). El backlog pasa a ser (600 − 300) × 120 = 36 000 y la espera 36 000 ÷ 300 = 120 s: la promesa se cumple por muy poco, sin ningún margen
Lo que permiten las existencias	25 000 ÷ 2.5 = 10 000 pedidos pueden tener éxito. A 600 por segundo, esa cantidad llega en menos de 17 segundos; poner en cola los otros 62 000 pedidos aproximadamente solo retrasa el inevitable 'agotado'
Decisión	75 workers; una sala de espera delante de la página de pedidos admite a 300 compradores por segundo; la cola de pedidos se limita a 12 000 (las existencias más un 20% por los pagos fallidos); una vez que las reservas cubren las existencias, los nuevos envíos reciben de inmediato la respuesta 'agotado, únete a la lista de espera' en lugar de entrar en la cola

La ley de Little convierte una promesa sobre el tiempo de espera en una longitud máxima de cola. Cualquier cola a la que se deje crecer por encima de ella es una promesa incumplida que todavía no se ha anunciado.

El límite de tasa del sistema posterior, y no el número de workers que podía arrancar la plataforma, fijó la capacidad real, así que añadir consumidores tenía un techo infranqueable.

El mejor límite para una cola suele ser un límite de negocio: no tiene sentido retener trabajo que no puede tener éxito de ninguna manera.

Back pressure: hacer que los productores vayan más despacio

El back pressure (contrapresión) es cualquier mecanismo por el que una etapa posterior más lenta hace que una etapa anterior más rápida vaya más despacio, en lugar de dejar que el trabajo se amontone entre ambas. El término viene de los fluidos en tuberías, y la idea apareció en tuberías y filtros (Sección 5.4). Está integrado en muchas capas de la pila que ya usas. El control de flujo de TCP detiene a un emisor cuando el búfer del receptor está lleno. Una cola bloqueante acotada hace que el `put` del productor espere a que haya espacio. Los consumidores basados en pull, como en Kafka, toman trabajo solo cuando están listos, de modo que es el log del broker, y no la memoria del consumidor, lo que guarda el backlog, y un productor de Kafka se bloquea o falla cuando su propio búfer de envío está lleno. La especificación Reactive Streams permite que un suscriptor solicite un número concreto de elementos, de modo que un flujo nunca entrega más de lo que el suscriptor ha pedido.

El back pressure funciona bien dentro de un proceso o a lo largo de un pipeline. A través de una frontera de solicitud-respuesta con usuarios humanos, normalmente se convierte en rechazo: un servicio HTTP sobrecargado no puede hacer que un teléfono espere educadamente, así que responde `429 Too Many Requests` o `503 Service Unavailable`, idealmente con una cabecera `Retry-After`, y confía en que el cliente reduzca el ritmo. Cuando una cola acotada está llena solo hay tres opciones: bloquear al productor, rechazar el trabajo nuevo o descartar algo ya aceptado. Cada una es la correcta en algún caso; el error es no elegir.

Tres respuestas a una cola llena

Cuando una cola acotada está llena, el sistema debe bloquear, rechazar o descartar; elegir una opción de forma deliberada es lo que mantiene contenida la sobrecarga.

Elegir qué ocurre cuando una cola está llena

Estrategia	Comportamiento	Adecuada para	Riesgo
Bloquear al productor	El productor espera hasta que haya espacio	Pipelines y trabajos por lotes en los que el productor puede hacer una pausa	Los productores bloqueados retienen hilos y conexiones; entre servicios, esto propaga la ralentización hacia arriba
Rechazar el trabajo nuevo	Fallar rápido con un error como 429 o 503	Solicitudes interactivas; el cliente puede reintentar, esperar o mostrar un mensaje	Los clientes que reintentan de inmediato convierten el rechazo en más carga (Sección 9.4)
Descartar el más antiguo	Desechar el elemento que lleva más tiempo esperando	Telemetría, actualizaciones de ubicación, precios en vivo: los datos más recientes son los más valiosos	Pérdida silenciosa; debe contarse y ser visible
Descartar por prioridad	Desechar primero los elementos menos importantes	Cargas de trabajo mixtas: conservar los pagos, descartar los eventos de analítica	Las prioridades bajas pueden quedar desatendidas indefinidamente; las prioridades deben asignarse con honestidad
Desbordar a almacenamiento duradero	Desbordar a disco o a una cola duradera	Trabajo que debe hacerse tarde o temprano, pero puede esperar horas	Solo es una cola más grande: sigue necesitando un límite y una estimación del tiempo de vaciado

Descarte de carga y priorización

Un servidor con más carga de la que puede gestionar lo hace peor que si atendiera su capacidad y rechazara el resto. Lo acepta todo, cada solicitud espera en sus colas internas, los tiempos de respuesta superan los timeouts de los clientes, y acaba gastando toda su capacidad en solicitudes cuyos clientes ya se han rendido. El rendimiento (throughput) se mantiene alto mientras que el *goodput*, la tasa de solicitudes completadas a tiempo para ser útiles, se desploma hacia cero. El descarte de carga (load shedding) lo evita rechazando algunas solicitudes de forma deliberada para que el resto tenga éxito.

Un buen descarte tiene tres propiedades. Es **temprano y barato**: rechazar una solicitud debe costar mucho menos que atenderla, así que se hace en la puerta de entrada, antes de la autenticación costosa o del trabajo con la base de datos, a partir de una señal barata como las

solicitudes en curso o la antigüedad de la cola. Está **priorizado**: el libro de SRE de Google describe cómo etiquetar cada solicitud con una criticidad (de CRITICAL_PLUS a SHEDDABLE) para que el tráfico por lotes y de precarga se descarte antes que el tráfico de cara al usuario. En la plataforma de reparto de comida, los pagos y las actualizaciones de ubicación de los repartidores tienen prioridad sobre las recomendaciones, que a su vez la tienen sobre los eventos de analítica. Y es **consciente del tiempo**: una solicitud que ya ha esperado más que el timeout de quien la llama debe descartarse en lugar de procesarse. Los ingenieros de Facebook han descrito dos refinamientos: una cola de retardo controlado que solo permite una espera corta cuando una cola no se ha vaciado recientemente, y pasar a último en entrar, primero en salir (LIFO) bajo carga, para que se atiendan primero las solicitudes más recientes, cuyos usuarios todavía están esperando.

Un paso más es adaptar el propio límite. Los límites de concurrencia fijos son o demasiado bajos en un buen día o demasiado altos en uno malo; los limitadores de concurrencia adaptativos aumentan el número permitido de solicitudes en curso mientras la latencia se mantiene baja y lo reducen cuando la latencia sube, tomando la idea del control de congestión de TCP. En el extremo frontal de un sistema, el control de admisión puede adoptar la forma de una sala de espera virtual, como en la venta de entradas anterior: los usuarios se retienen en una cola en el borde y se admiten al ritmo que el núcleo puede atender, lo cual es descarte de carga presentado como equidad.

SI ESTO ES NUEVO PARA TI

Paso a paso: ocupado pero inútil

La idea más difícil de esta sección es que un servidor sobrecargado que acepta todas las solicitudes puede acabar sin hacer casi ningún trabajo útil, aunque nunca deje de trabajar. Rechazar algunas solicitudes a propósito, el descarte de carga, es lo que mantiene útil el resto.

- Supón que el proceso de pago de la plataforma de reparto de comida puede completar 100 solicitudes por segundo, y que la app móvil abandona cualquier solicitud que no reciba respuesta en 2 segundos.
- Durante una promoción llegan 150 solicitudes cada segundo. El servicio las acepta todas, así que su cola interna crece en 50 cada segundo.
- Al cabo de 4 segundos hay unas 200 esperando. A 100 por segundo, una nueva llegada espera unos 2 segundos antes de que empiece el trabajo con ella, así que para entonces su app ya la ha abandonado.
- La cola sigue creciendo, así que a partir de ese momento casi todas las solicitudes terminan demasiado tarde. El servicio sigue completando 100 por segundo, así que su rendimiento (throughput) parece sano, pero su goodput, las solicitudes completadas a tiempo para ser útiles, ha caído hacia cero.
- Ahora añade descarte en la puerta de entrada: cuando las solicitudes en curso alcanzan un límite elegido para que el trabajo aceptado termine a tiempo, las nuevas se rechazan de inmediato con un error 503 ('servicio no disponible'), lo que cuesta mucho menos que atenderlas. Se rechazan unas 50 por segundo; las 100 aceptadas reciben respuesta a tiempo.
- Un buen descarte también está priorizado: un servidor que gestiona varios tipos de solicitud rechaza las recomendaciones y los eventos de analítica antes que los pagos. También descarta cualquier solicitud que ya haya esperado más que el timeout de quien la llama.
- **La lección:** en situaciones de sobrecarga, la elección real no es entre atender a todos y atender a algunos, sino entre rechazar a algunos a propósito y no atender a casi nadie a tiempo. Rechaza pronto, a bajo costo y por prioridad.

Algoritmos de limitación de tasa

La limitación de tasa (rate limiting) restringe cuánto trabajo puede enviar en un periodo un cliente, una clave de API, un inquilino (tenant) o una dirección IP concretos. Se diferencia del descarte de carga en su propósito: el descarte protege al servidor de una sobrecarga total, la cause quien la cause, mientras que la limitación de tasa reparte la capacidad de forma

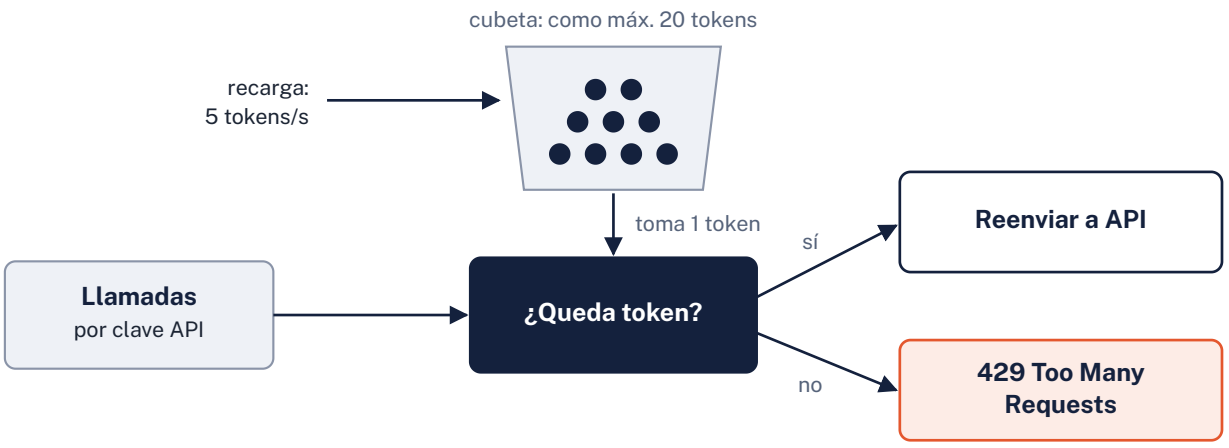
equitativa y protege frente a cualquier cliente concreto, ya tenga errores o sea malicioso. Ambos responden de la misma manera, normalmente con HTTP 429. Predominan cuatro algoritmos.

Comparación de los algoritmos de limitación de tasa

Algoritmo	Cómo funciona	Ráfagas	Estado por clave	Notas
Token bucket (cubeta de tokens)	Se añaden tokens a la tasa r hasta la capacidad b; cada solicitud gasta uno	Permite ráfagas de hasta b y después r de media	Dos números: los tokens y el momento de la última recarga	El más usado; la ráfaga y la tasa sostenida se ajustan por separado
Leaky bucket (cubeta con fugas)	Las solicitudes entran en una cubeta que se vacía a una tasa constante; cuando está llena, se rechazan las nuevas solicitudes	La salida es perfectamente uniforme; las ráfagas esperan o se rechazan	Una cola, o un contador y una marca de tiempo	Se usa para la conformación de tráfico (traffic shaping); la limitación de solicitudes de NGINX se basa en él
Contador de ventana fija	Cuenta las solicitudes por ventana de calendario (por minuto); se pone a cero al cambiar de ventana	Hasta el doble del límite en torno a un cambio de ventana	Un contador	El más sencillo; 100 solicitudes a las 12:00:59 y 100 a las 12:01:00 pasan todas
Registro de ventana deslizante	Guarda la marca de tiempo de cada solicitud de la última ventana; las cuenta	Exacto	Una entrada por solicitud	Preciso, pero consume mucha memoria con tasas altas
Contador de ventana deslizante	Pondera el recuento de la ventana anterior según cuánto se solapa todavía y suma el recuento actual	Aproximación cercana	Dos contadores	A los 30 s de un minuto, con 80 en el minuto anterior y 40 ahora: 80 × 0.5 + 40 = 80

El token bucket (cubeta de tokens) merece un examen más detenido porque es la opción predeterminada habitual. La cubeta contiene hasta b tokens y gana r tokens por segundo. Cada solicitud toma un token; si no queda ninguno, se rechaza (o, en la variante de conformación, se retrasa). Por tanto, un cliente que ha estado inactivo puede enviar una ráfaga de b solicitudes a la vez, lo que se adapta a clientes reales como una app móvil que abre una pantalla, mientras que en cualquier periodo más largo no puede superar r de media. El leaky bucket (cubeta con fugas) es su imagen especular: suaviza la salida hasta una tasa constante, lo que resulta útil para proteger un sistema posterior que no admite ráfagas en absoluto.

El token bucket



Cada solicitud debe tomar un token; la tasa de recarga fija la tasa sostenida y la capacidad de la cubeta fija la ráfaga más grande.

EJEMPLO PRÁCTICO

Ejemplo práctico: un token bucket para una API de socios

La clave de API de un restaurante asociado está limitada por un token bucket con capacidad b = 20 y tasa de recarga r = 5 tokens por segundo (300 por minuto de media). La cubeta está llena en t = 0.

t = 0: una ráfaga de 30 solicitudes	Las 20 primeras toman los 20 tokens y se reenvían; 10 se rechazan con 429
t = 0 a 1 s	Llegan 5 tokens nuevos (5 por segundo)
t = 1 a 10 s: 8 solicitudes por segundo constantes	Se aceptan unas 5 por segundo y se rechazan 3; la cubeta se mantiene casi vacía
t = 10 a 14 s: ninguna solicitud	La cubeta se recarga a 5 por segundo y vuelve a estar llena (20) en t = 14
Cota superior en cualquier periodo T	Como máximo b + r × T solicitudes: en cualquier periodo de 60 segundos, 20 + 5 × 60 = 320, así que la media a largo plazo nunca puede superar 5 por segundo
Retry-After para una solicitud rechazada	El siguiente token llega en 1 ÷ r = 0.2 s, pero la cabecera cuenta segundos enteros, así que la respuesta dice Retry-After: 1

La capacidad fija la ráfaga que puede enviar un cliente que se comporta bien tras un periodo de inactividad; la tasa de recarga fija la tasa sostenida. Elígelas por separado.

Un cliente que respeta Retry-After, con un poco de jitter, se estabiliza en la tasa de recarga; uno que reintenta de inmediato solo acumula más 429.

```
import math, time

class TokenBucket:
    def __init__(self, rate, capacity):
        self.rate = rate              # tokens added per second
        self.capacity = capacity      # largest burst allowed
        self.tokens = capacity        # start full
        self.updated = time.monotonic()

    def allow(self, cost=1):
        now = time.monotonic()
        elapsed = now - self.updated
        self.tokens = min(self.capacity, self.tokens + elapsed * self.rate)
        self.updated = now
        if self.tokens ⩾ cost:
            self.tokens -= cost
            return True, 0
        wait = (cost - self.tokens) / self.rate   # seconds until enough tokens
        return False, max(1, math.ceil(wait))      # for the Retry-After header

bucket = TokenBucket(rate=5, capacity=20)        # one bucket per API key
allowed, retry_after = bucket.allow()
```

Un token bucket con recarga diferida: los tokens se reponen según el tiempo transcurrido desde la última llamada, así que no hace falta ningún temporizador. En producción, el estado vive en un almacén compartido como Redis y se actualiza de forma atómica.

HTTP 429, Retry-After y dónde aplicar los límites

HTTP tiene señales estándar para ambas situaciones. `429 Too Many Requests`, definido en el RFC 6585 (2012), indica a un cliente que ha enviado demasiadas solicitudes en un tiempo determinado. `503 Service Unavailable` dice que el servidor en su conjunto está sobrecargado temporalmente o en mantenimiento. Cualquiera de los dos puede llevar una cabecera `Retry-After` (RFC 9110), que da un número de segundos o una fecha a partir de la cual tiene sentido reintentar. Muchas API también devuelven cabeceras que anuncian el límite, la cuota restante y el momento del reinicio, tradicionalmente `X-RateLimit-*`; un borrador del IETF está estandarizando los campos `RateLimit` y `RateLimit-Policy`. Los clientes que se comportan bien respetan `Retry-After`, añaden jitter para que los clientes rechazados no vuelvan todos en el mismo instante, y tratan los 429 repetidos como un motivo para ir más despacio y no para reintentar con más fuerza (Sección 9.4).

Los límites se pueden aplicar en varias capas, y los sistemas maduros usan más de una: en el borde o en el API gateway, por dirección IP y clave de API, para detener el abuso y los clientes descontrolados antes de que cuesten nada; por inquilino, para que un cliente grande no deje sin recursos a los demás; y dentro de cada servicio, para proteger su propia capacidad y sus dependencias. Un límite global entre muchas instancias del gateway necesita estado

compartido, normalmente contadores atómicos en Redis, lo que añade una llamada de red por solicitud; un compromiso habitual es dar a cada una de las N instancias un límite local de aproximadamente 1 ÷ N del total y aceptar cierta imprecisión. Los límites deben fijarse a partir de la capacidad medida y publicarse, para que los socios puedan diseñar teniéndolos en cuenta, y el número de rechazos debe monitorizarse, porque un aumento repentino es o un ataque, o un error en un cliente, o un límite que ya no se ajusta a la realidad.

Colas de mensajes fallidos

Algunos mensajes fallan siempre: un payload mal formado, un pedido para un restaurante que ya no existe, un bug que desencadena una entrada concreta. Si se reintenta indefinidamente, un mensaje así, un mensaje venenoso (poison message), bloquea una partición o desperdicia un worker para siempre. Una cola de mensajes fallidos (dead-letter queue) (presentada en la Sección 6.4) recibe los mensajes que han fallado un número determinado de veces (SQS lo llama número máximo de recepciones, maximum receive count, en una redrive policy), para que el resto del trabajo pueda continuar.

Una cola de mensajes fallidos es una herramienta para los defectos, no para la sobrecarga: si los mensajes acaban en ella porque los consumidores son demasiado lentos, la solución es la capacidad o el back pressure, no una segunda cola. Cada mensaje de una DLQ significa que algo que el negocio esperaba no ocurrió, así que la DLQ necesita un responsable, una alerta sobre su profundidad o antigüedad, herramientas para inspeccionar los mensajes con su error y su número de intentos, y una forma de volver a procesarlos una vez corregida la causa. Decide de antemano qué significa reprocesar con retraso para el orden y para los mensajes cuyo momento de negocio ya ha pasado: un reembolso puede reprocesarse con un día de retraso; una asignación de repartidor, no.

ATENCIÓN

Errores comunes

- Colas sin límite, en memoria o en disco, que convierten una sobrecarga temporal en una caída o en horas de trabajo obsoleto.
- Monitorizar la longitud de la cola pero no la antigüedad del mensaje más antiguo, de modo que los consumidores atascados pasan desapercibidos mientras la longitud parece normal.
- Descartar carga después de hacer la parte costosa del trabajo, de modo que rechazar cuesta casi tanto como tener éxito.
- Límites de tasa de ventana fija que permiten a los clientes enviar el doble del límite en torno a un cambio de ventana.
- Clientes que ignoran el 429 y Retry-After y reintentan de inmediato, con lo que convierten un límite en una avalancha.
- Timeouts de visibilidad más cortos que el tiempo de procesamiento, de modo que dos workers procesan un mismo mensaje a la vez.
- Usar una cola de mensajes fallidos como depósito de desbordamiento en lugar de corregir la capacidad, y dejarla sin responsable y sin monitorizar.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ Los workers procesan 200 mensajes por segundo y los mensajes pueden esperar como máximo 30 segundos. ¿Cuál es la mayor longitud de cola aceptable?

Según la ley de Little, 200 × 30 = 6000 mensajes; una cola más larga significa que algunos mensajes esperarán más de 30 segundos.

▼ ¿Por qué la antigüedad del mensaje más antiguo es una alarma mejor que la longitud de la cola?

La antigüedad mide el retraso que experimentan los usuarios y sube de inmediato si los consumidores se atascan, mientras que una longitud que parece normal puede ocultar una cola atascada o un vaciado lento.

▼ Un token bucket tiene capacidad 50 y se recarga a 10 por segundo. ¿Cuántas solicitudes puede hacer un cliente en el primer segundo tras un largo periodo de inactividad?

Unas 60: los 50 tokens que ya hay en la cubeta más los 10 que se añaden durante ese segundo.

▼ **¿Cuál es la diferencia entre el descarte de carga y la limitación de tasa?**

El descarte de carga protege al servidor de una sobrecarga total, la cause quien la cause; la limitación de tasa pone un tope a lo que puede enviar cada cliente para que la capacidad se reparta de forma equitativa.

FUENTES

- Microsoft Azure Architecture Center: Queue-Based Load Leveling pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling
- Google SRE book: Handling Overload: https://sre.google/sre-book/handling-overload/
- RFC 6585: Additional HTTP Status Codes (429 Too Many Requests): https://www.rfc-editor.org/rfc/rfc6585
- Ben Maurer: Fail at Scale (ACM Queue, 2015): https://queue.acm.org/detail.cfm?id=2839461

SECCIÓN 9.4 DE 9.8 · AVANZADO · 32 MIN DE LECTURA

Timeouts, reintentos, circuit breakers, bulkheads y degradación elegante

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.4 Rendimiento y escalabilidad
- 6.1 Por qué distribuir
- 6.5 Comunicación y API

Términos anteriores que utiliza: Fallo parcial, Percentil (p95, p99), Latencia de cola (tail latency), Ley de Little, Clave de idempotencia

LA IDEA EN BREVE

Imagina que llamas por teléfono a un proveedor que nunca contesta. Si te quedas al teléfono todo el día, no haces nada más, y si todos tus compañeros hacen lo mismo, toda la oficina se para. Las oficinas sensatas cuelgan tras unos cuantos tonos (un timeout), vuelven a intentarlo un poco más tarde, pero no sin fin (un reintento con pausa), dejan de llamar a un proveedor que claramente está cerrado y vuelven a probar al cabo de una hora (un circuit breaker), se aseguran de que un solo proveedor no pueda ocupar todos los teléfonos (un bulkhead) y tienen un plan B, como la lista de precios de ayer (una alternativa o fallback). Los servicios de software necesitan exactamente los mismos hábitos.

LO QUE NECESITAS SABER

- Los fallos en cascada se producen cuando quienes llaman esperan a una dependencia lenta hasta agotar sus propios hilos y conexiones; cada punto de integración necesita una protección deliberada (Nygard, Release It!).
- Fija timeouts explícitos de conexión y de solicitud a partir de percentiles de latencia medidos, cada vez más cortos a lo largo de la cadena de llamadas, y propaga el plazo restante para que ningún servicio trabaje en una solicitud abandonada.
- Reintenta solo los errores transitorios en operaciones idempotentes, unas pocas veces, con backoff exponencial limitado y jitter completo (Brooker, 2015).
- Los reintentos se multiplican entre capas (tres intentos en tres capas dan 27 llamadas); reintenta en una sola capa y limita los reintentos con un presupuesto, por ejemplo el 10% de las solicitudes.

- Un circuit breaker falla rápido mientras una dependencia está caída: pasa de cerrado a abierto cuando hay fallos y, a través de llamadas de prueba en estado semiabierto, vuelve a cerrado.
- Los bulkheads dan a cada dependencia o a cada llamante su propio pool acotado, y las alternativas mantienen en funcionamiento el recorrido principal, pero deben ser sencillas, independientes de la parte que ha fallado, estar probadas y, en materia de seguridad, fallar de forma cerrada (fail-closed).
- La idempotencia hace seguros los reintentos, las solicitudes de cobertura y los reprocesamientos; las solicitudes de cobertura (hedged requests) reducen la latencia de cola en las lecturas idempotentes a cambio de un pequeño costo, presupuestado, en carga adicional.

El fallo es normal; propagarlo es opcional

En un sistema distribuido siempre hay alguna dependencia lenta o fallando: un proveedor de pagos tiene un mal minuto, una base de datos hace failover, un enlace de red pierde paquetes, un despliegue sale mal. La Sección 6.1 mostró por qué estos fallos parciales no se pueden evitar y por qué a menudo quien llama ni siquiera puede saber qué ha ocurrido. Por tanto, la pregunta arquitectónica no es cómo evitar el fallo, sino cómo impedir que un fallo local se convierta en uno global. Un fallo en cascada es la forma típica en que mueren los sistemas: una dependencia lenta vuelve lentos a quienes la llaman, sus recursos se llenan de solicitudes en espera, ellos a su vez se vuelven lentos para quienes los llaman, y en cuestión de segundos un problema en un componente menor ha tumbado toda la plataforma.

Michael Nygard catalogó los patrones que interrumpen estas cadenas en *Release It!* (2007; segunda edición, 2018), a partir de las caídas que había investigado. Su observación central es que los puntos de integración, todos los lugares en que un sistema llama a otro, son la principal fuente de inestabilidad, y que cada uno necesita una protección deliberada. Los patrones de esta sección (timeouts, plazos, reintentos con backoff, circuit breakers (disyuntores), bulkheads (mamparos), alternativas, idempotencia y solicitudes de cobertura) son el conjunto de herramientas estándar. Son tácticas en el sentido de la Sección 2.5: cada una es una única decisión orientada a la disponibilidad, y cada una tiene un costo.

Timeouts: el primer patrón y el más importante

Un timeout limita cuánto tiempo espera quien llama. Sin él, una llamada a una dependencia que nunca responde retiene un hilo, una conexión y memoria mientras el sistema operativo mantenga abierta la conexión, lo que pueden ser muchos minutos. Muchas bibliotecas cliente no tienen timeout por defecto (la popular biblioteca `requests` de Python espera

indefinidamente salvo que se le indique lo contrario), así que la ausencia de un timeout es una decisión de diseño tomada por omisión. El ejemplo práctico de más abajo muestra la consecuencia.

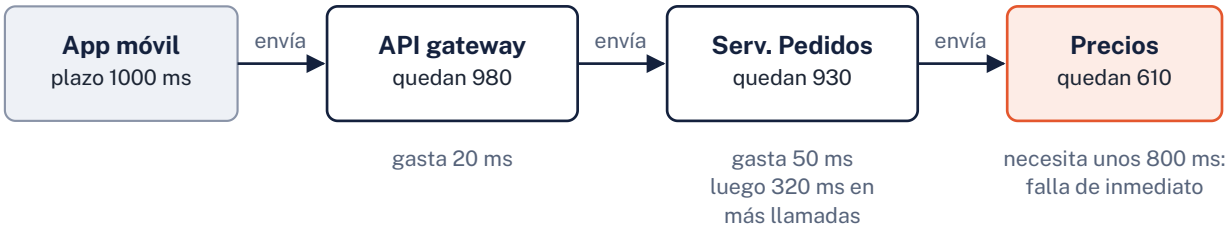
Normalmente importan dos timeouts. El **timeout de conexión** limita el establecimiento de una conexión y puede ser corto, unos cientos de milisegundos dentro de un centro de datos, porque un servidor sano acepta conexiones casi de inmediato. El **timeout de solicitud** limita todo el intercambio y debe salir de la medición: un poco por encima de la latencia de percentil alto de la dependencia (p99 o p99.9) con carga normal, para que corte las esperas patológicas sin hacer fallar las solicitudes lentas corrientes. También debe caber dentro del presupuesto de quien llama. Si el gateway se rinde a los 2 segundos pero el servicio de pedidos espera 10 segundos al de precios, el servicio de pedidos sigue trabajando durante 8 segundos en una solicitud que nadie está esperando. Por tanto, los timeouts deben acortarse a medida que se baja por una cadena de llamadas, que es la idea en la que se basan los plazos.

Los plazos y su propagación

Un plazo (deadline) es un momento absoluto en el tiempo para el que debe haber terminado toda la solicitud. La propagación de plazos (deadline propagation) pasa el tiempo restante a cada llamada posterior, para que cada servicio sepa cuánto presupuesto queda en lugar de aplicar un timeout fijo propio. gRPC lo incorpora: un cliente fija un plazo, el tiempo restante viaja con la llamada, y un servidor puede comprobar si el plazo ha vencido y detener el trabajo. Sobre HTTP, los equipos propagan una cabecera con los milisegundos restantes o con el plazo absoluto, idealmente mediante el mismo middleware que propaga el contexto de trazado (Sección 9.6).

La propagación aporta dos beneficios. Un servicio puede fallar rápido cuando el tiempo que queda es claramente demasiado corto para el trabajo, en lugar de iniciar una consulta a la base de datos que no puede terminar a tiempo. Y cuando el usuario se rinde, el trabajo de los niveles inferiores también se detiene, en lugar de seguir consumiendo capacidad justo durante la sobrecarga que causó el retraso. Esto es descarte de carga de solicitudes muertas, una de las protecciones más eficaces que puede tener un sistema sometido a estrés.

Un plazo propagado a lo largo de una cadena de llamadas



gasta 20 ms

gasta 50 ms
luego 320 ms en
más llamadas

necesita unos 800 ms:
falla de inmediato

Cada salto envía el tiempo restante, no un timeout nuevo propio

Cada salto pasa el tiempo restante; un servicio que no puede terminar en el tiempo que queda falla de inmediato en lugar de hacer un trabajo que nadie va a esperar.

EJEMPLO PRÁCTICO

Ejemplo práctico: cómo un solo timeout que falta se convierte en una caída total

El servicio de pedidos de la plataforma de reparto ejecuta 6 instancias con 200 hilos de trabajo cada una: 1200 hilos. A la hora del almuerzo recibe 400 solicitudes por segundo: 100 de pago, que llaman al proveedor de pagos (normalmente 300 ms, así que una solicitud de pago retiene su hilo unos 350 ms en total), y 300 solicitudes de otro tipo (historial de pedidos, carrito, seguimiento) que tardan 50 ms. El cliente de pagos no tiene ningún timeout configurado. A las 12:31 el proveedor de pagos deja de responder, pero mantiene abiertas las conexiones.

Hilos ocupados en condiciones normales (ley de Little)	Pago 100 × 0.35 s = 35; resto del tráfico 300 × 0.05 s = 15; total 50 de 1200 (alrededor del 4%)
Después de las 12:31	Cada solicitud de pago retiene ahora su hilo indefinidamente: cada segundo se quedan atascados 100 hilos más
Tiempo hasta el agotamiento	(1200 − 15 − 35) ÷ 100 ≈ 11.5 segundos: en unos 12 segundos todos los hilos están esperando al proveedor de pagos
Consecuencia	El historial de pedidos, el carrito y el seguimiento, que nunca llaman a pagos, ahora hacen cola para conseguir hilos y agotan su timeout en el gateway. Un fallo en el 25% del tráfico ha tumbado el 100%
Y aguas arriba	Las conexiones del gateway al servicio de pedidos se acumulan de la misma manera; las comprobaciones de estado agotan su timeout, las instancias se marcan como no sanas y, si la sonda de actividad (liveness probe) se ve afectada, se reinician, en frío, en las mismas condiciones
Con un timeout de 2 segundos	Hilos de pago en uso: 100 × 2 s = 200 en toda la flota (ley de Little). Quedan unos 985 hilos para el resto del tráfico, así que solo falla el pago, y falla con un error claro al cabo de 2 s
Añadir un circuit breaker	En cuanto, por ejemplo, la mitad de las 20 últimas llamadas han fallado, el breaker se abre: los pagos fallan en milisegundos y casi no usan hilos; una llamada de prueba cada 30 s detecta la recuperación
Añadir un bulkhead	Limitar las llamadas de pago a 40 hilos por instancia (240 en total) garantiza al menos 960 hilos para todo lo demás, sean cuales sean los timeouts configurados
Añadir una alternativa	El proceso de pago ofrece al cliente otro método de pago, o explica que los pagos con tarjeta se están retrasando, en lugar de un error genérico

La aritmética es de nuevo la ley de Little: hilos en uso = tasa de solicitudes × tiempo que cada solicitud retiene un hilo. Sin timeout, el tiempo de retención no tiene límite, así que los hilos necesarios tampoco lo tienen.

Los timeouts limitan cuánto tiempo; los bulkheads limitan cuántos; los circuit breakers dejan de llamar por completo. Cada uno cierra una brecha distinta, y por eso se usan juntos.

Release It!, de Nygard, describe caídas reales que siguieron exactamente este patrón, a menudo desencadenadas por un punto de integración que nadie consideraba crítico.

Agotamiento del pool de hilos

Sin timeout, un proveedor de pagos lento retiene todos los hilos del servicio de pedidos, así que fallan incluso las solicitudes que nunca tocan los pagos.

Reintentos con backoff exponencial y jitter

Muchos fallos son transitorios: un paquete perdido, una conexión reiniciada durante un failover, un breve 503 de una instancia sobrecargada. Reintentar los convierte en éxitos que el usuario nunca ve. Pero los reintentos también son carga, y los reintentos mal diseñados están entre las causas más comunes de las caídas. Cuatro reglas los hacen seguros. Reintenta solo los errores **transitorios** (timeouts, fallos de conexión, 502, 503, 504, y 429 después del retraso indicado), nunca errores que se repetirán, como 400 o 404. Reintenta solo operaciones **idempotentes** o que lleven una clave de idempotencia, porque una solicitud que agotó su timeout puede haber tenido éxito. Reintenta un **número pequeño** de veces, normalmente dos o tres. Y **espera** entre intentos, con backoff exponencial: un retraso que se duplica en cada intento, hasta un tope.

El backoff por sí solo no basta. Si mil clientes fallan en el mismo momento, como ocurre cuando una dependencia tiene un fallo momentáneo, todos reintentan al cabo de 100 ms, después todos al cabo de 200 ms, y llegan en oleadas sincronizadas que pueden mantener caída la dependencia. El artículo de Marc Brooker *Exponential Backoff And Jitter* (AWS Architecture Blog, 2015) lo simuló y mostró que añadir aleatoriedad, jitter (fluctuación aleatoria), reduce mucho tanto el trabajo total como el tiempo hasta completarlo. Su "jitter completo" (full jitter) recomendado espera un tiempo aleatorio entre cero y la cota exponencial: `sleep = random(0, min(cap, base × 2^attempt))`. El jitter debe aplicarse a todo lo que muchos clientes hacen a la vez: reintentos, reconexiones, tiempos de caducidad de la caché y trabajos programados.

```
import random, time

class TransientError(Exception):
    """Timeouts, connection resets, 502, 503 and 504 responses."""

def call_with_retries(fn, attempts=3, base=0.1, cap=2.0):
    for attempt in range(attempts):
        try:
            return fn()
        except TransientError:
            if attempt == attempts - 1:
                raise                       # give up: let the caller decide
            ceiling = min(cap, base * 2 ** attempt)
            time.sleep(random.uniform(0, ceiling))   # full jitter
```

Reintentar un fallo transitorio con backoff exponencial limitado y jitter completo (Brooker, 2015). Solo deben reintentarse las operaciones idempotentes o las que llevan una clave de idempotencia.

SI ESTO ES NUEVO PARA TI

Paso a paso: por qué los reintentos necesitan aleatoriedad

La idea más difícil de esta sección es que el backoff exponencial por sí solo no reparte los reintentos: cuando muchos llamantes fallan en el mismo momento, también esperan los mismos tiempos y vuelven juntos. Añadir aleatoriedad, el jitter, es lo que dispersa la multitud.

- Supón que hay 1000 llamadas de pago en curso, cada una con una clave de idempotencia para que los reintentos sean seguros, cuando el proveedor tiene un fallo momentáneo. Las 1000 fallan en el mismo instante.
- Cada llamante sigue un backoff exponencial simple: espera 100 ms y reintenta; si falla, espera 200 ms, y después 400 ms. Como todos fallaron a la vez, los 1000 reintentos llegan juntos, 100 ms después.
- Un proveedor que se está recuperando no puede aceptar 1000 llamadas en un instante, así que muchas vuelven a fallar, y la siguiente oleada llega junta 200 ms después. Entre oleadas el proveedor está tranquilo; durante ellas está desbordado, y las oleadas pueden mantenerlo caído.
- Con jitter completo, cada llamante espera un tiempo aleatorio entre cero y el retraso del backoff. Los primeros reintentos se reparten a lo largo de los 100 ms: unos 10 por milisegundo en lugar de 1000 a la vez.
- Los llamantes que vuelven a fallar esperan un tiempo aleatorio de hasta 200 ms, y después de hasta 400 ms, así que la carga llega repartida, algo que un proveedor en recuperación puede absorber con mucha más facilidad. Las simulaciones de Brooker concluyeron que esto reduce tanto el trabajo total como el tiempo hasta terminar.
- La misma aglomeración se produce con cualquier cosa que muchas máquinas hagan a la vez, como reconectarse, la caducidad de las entradas de la caché o los trabajos programados, así que el jitter también debe aplicarse ahí.
- **La lección:** el backoff hace que los llamantes esperen más cada vez; el jitter garantiza que no esperen todos el mismo tiempo. Esperar educadamente no basta si todos esperan al compás.

Tormentas de reintentos y presupuestos de reintentos

Los reintentos se multiplican entre capas. Supón que un toque en la app móvil pasa por el API gateway, el servicio de pedidos y el cliente de pagos, y que cada uno de los tres hace hasta tres intentos. Si el proveedor de pagos está caído, una sola acción del usuario produce 3 × 3 × 3 = 27 llamadas a él, y una caída que debería haber supuesto una tasa de errores constante se

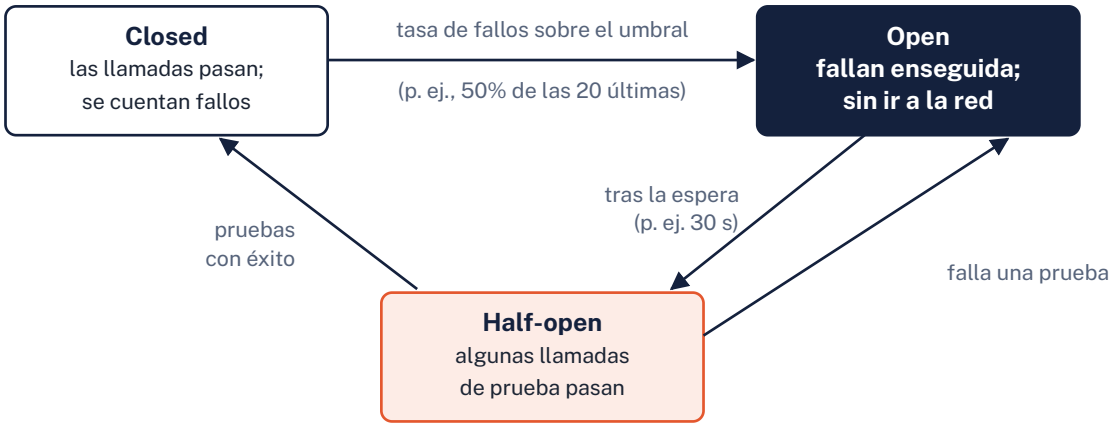
convierte en una tormenta de reintentos de 27 veces la carga normal, que mantiene caído al proveedor incluso después de corregida la causa original. Los usuarios que vuelven a tocar añaden una cuarta capa.

Tres defensas funcionan juntas. Primera, **reintenta en una sola capa**, normalmente la más cercana a la dependencia que falla, y haz que las demás capas pasen el error hacia arriba; el libro de SRE de Google recomienda que un servidor sobrecargado lo diga explícitamente, para que los llamantes de niveles superiores no reintenten. Segunda, usa un presupuesto de reintentos: el libro de SRE describe permitir reintentos solo mientras estén por debajo del 10% de las solicitudes de un cliente, lo que limita la carga adicional de los reintentos a unas 1.1 veces la normal en lugar de hasta tres veces. Los SDK de AWS implementan una cuota similar de tokens de reintento, y la política de reintentos de gRPC incluye una limitación (throttling) con el mismo fin. Tercera, combina los reintentos con un circuit breaker, para que no se reintente en absoluto una dependencia de la que se sabe que está caída.

Circuit breakers

Un circuit breaker, llamado así por Nygard en referencia al dispositivo eléctrico (el disyuntor), envuelve las llamadas a una dependencia y vigila sus resultados. Mientras la dependencia está sana, el breaker está **cerrado** (closed) y las llamadas pasan, mientras se cuentan los fallos y los timeouts. Cuando los fallos superan un umbral, por ejemplo la mitad de las 20 últimas llamadas, el breaker **se abre** (open): durante un periodo determinado todas las llamadas fallan de inmediato sin tocar la red, lo que protege los hilos de quien llama y da margen a la dependencia para recuperarse. Cuando vence el periodo, el breaker pasa a **semiabierto** (half-open) y deja pasar unas pocas llamadas de prueba. Si tienen éxito, se cierra; si una falla, vuelve a abrirse durante otro periodo. La descripción de Martin Fowler de 2014 dio a conocer ampliamente el patrón, y la biblioteca Hystrix de Netflix (ahora en modo de mantenimiento) lo popularizó en los microservicios; Resilience4j para Java y Polly para .NET son hoy las bibliotecas habituales, y las service meshes ofrecen protecciones relacionadas, como límites de concurrencia y detección de valores atípicos (outlier detection) (en Envoy se llaman circuit breaking, aunque funcionan de otra manera).

La máquina de estados del circuit breaker



Cerrado cuenta los fallos, abierto falla rápido y semiabierto comprueba la recuperación con unas pocas llamadas de prueba.

Los ajustes merecen reflexión. ¿Qué cuenta como fallo? Los timeouts y las respuestas 5xx, sí; las respuestas 4xx normalmente no, porque reflejan la solicitud de quien llama y no la salud de la dependencia. Los éxitos lentos también pueden contar, para que una dependencia que responde en 5 segundos se trate como fallida. La ventana debe ser lo bastante grande para no dispararse por un par de llamadas desafortunadas (las bibliotecas exigen un número mínimo de llamadas), y el periodo abierto lo bastante largo para la recuperación pero lo bastante corto para no prolongar una caída. Los breakers deben ser por dependencia, y a veces por endpoint, para que un endpoint de informes que falla no corte los pedidos. Los reintentos deben tratar un breaker abierto como una respuesta definitiva y no como algo que reintentar. Y cada cambio de estado debe registrarse y representarse en gráficos, porque un breaker abierto es un síntoma que alguien necesita ver.

```
import time

class CircuitOpen(Exception):
    pass

class CircuitBreaker:
    def __init__(self, failure_threshold=5, reset_after=30.0):
        self.failure_threshold = failure_threshold  # failures in a row that trip it
        self.reset_after = reset_after              # seconds to stay open
        self.failures, self.state, self.opened_at = 0, "closed", 0.0

    def call(self, fn, *args, **kwargs):
        if self.state == "open":
            if time.monotonic() - self.opened_at < self.reset_after:
                raise CircuitOpen("failing fast")   # no remote call at all
            self.state = "half-open"                # let a trial call through
        try:
            result = fn(*args, **kwargs)
        except Exception:
            self.failures += 1
            if self.state == "half-open" or self.failures ⩾ self.failure_threshold:
                self.state, self.opened_at = "open", time.monotonic()
            raise
        self.failures, self.state = 0, "closed"    # success closes the circuit
        return result

payments = CircuitBreaker()
# payments.call(provider.authorise, order_id, amount, timeout=2.0)
```

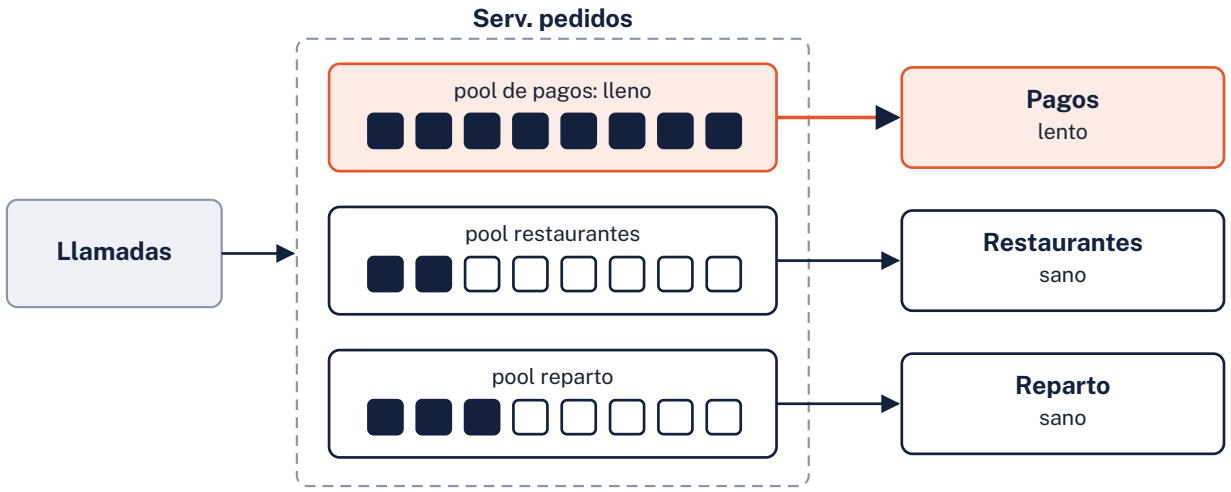
Un circuit breaker mínimo. Las bibliotecas de producción como Resilience4j (Java) y Polly (.NET) añaden ventanas de tasa de fallos, detección de llamadas lentas, límites de llamadas de prueba concurrentes, seguridad entre hilos y métricas.

Bulkheads

El casco de un barco está dividido por mamparos (bulkheads) en compartimentos estancos, para que una vía de agua inunde un compartimento en lugar de hundir el barco. En software, un bulkhead reparte los recursos para que una parte que falla o está sobrecargada no pueda consumirlo todo. La forma más común da a cada dependencia su propio pool de hilos, pool de conexiones o semáforo acotado: los pagos pueden usar como máximo 40 llamadas concurrentes por instancia, las consultas de restaurantes 40 y la asignación de repartidores 40. Cuando los pagos se ralentizan, su pool se llena y las siguientes llamadas de pago se rechazan de inmediato, pero las llamadas de restaurantes y repartidores continúan. Un timeout limita cuánto tiempo puede retener recursos cada llamada; un bulkhead limita cuántas pueden retenerlos a la vez, así que protege incluso cuando los timeouts son demasiado largos.

Los bulkheads también funcionan a mayor escala. Despliegues separados del mismo servicio para distintos llamantes (un grupo de instancias del servicio de pedidos para la app de clientes y otro para la API de restaurantes asociados) impiden que una integración de socio que se comporta mal deje sin recursos a los clientes. Las arquitecturas basadas en células, el eje Z del cubo de escalado, convierten cada célula en un bulkhead para una parte de los clientes, y el shuffle sharding, descrito por ingenieros de AWS, asigna a cada cliente una pequeña combinación aleatoria de instancias para que el tráfico venenoso de un cliente afecte a muy pocos otros. El costo de todo bulkhead es la utilización: la capacidad reservada para un compartimento no se puede prestar a otro, así que los pools deben dimensionarse a partir de la demanda medida.

Bulkheads dentro de un servicio



Con un pool separado por dependencia, un proveedor de pagos lento solo llena el pool de pagos; las llamadas a restaurantes y repartidores continúan con normalidad.

Alternativas (fallbacks) y degradación elegante

Cuando una llamada falla o un breaker está abierto, quien llama debe responder de todos modos. Una alternativa (fallback) decide qué devuelve. Las opciones, de mejor a peor para el usuario, son: un **valor en caché u obsoleto** (la última carta conocida, con los precios confirmados en el proceso de pago); un **valor predeterminado razonable** (un tiempo de entrega estimado de 30 a 45 minutos en lugar de una estimación en vivo); una **función reducida** (platos populares en lugar de recomendaciones personalizadas); **aplazar el trabajo** (aceptar una reseña y publicarla cuando se recupere el servicio de reseñas); y un **error claro y rápido** que indique al usuario qué hacer. Diseñarlas función por función, con el responsable de producto, es lo que significa en la práctica la degradación elegante: el recorrido principal (explorar, pedir,

pagar, seguir el pedido) sigue funcionando mientras las funciones no esenciales se hacen a un lado. El mismo razonamiento guía el descarte de carga (Sección 9.3): saber de antemano qué funciones desactivar primero.

Las decisiones de seguridad son la excepción: si el servicio de autorización no está accesible, la alternativa segura es denegar (fallar de forma cerrada, fail closed), no permitir. Los profesionales también discrepan sobre las alternativas en general. Nygard y los ingenieros de Netflix las defienden ampliamente. Los ingenieros de Amazon han argumentado, en el artículo de la Builders' Library *Avoiding fallback in distributed systems*, que las rutas alternativas rara vez se ejercitan, son difíciles de probar y pueden provocar ellas mismas caídas al pasar una flota grande a un modo no probado en el peor momento; prefieren hacer más fiable la ruta principal y fallar rápido. Ambas posturas coinciden en la regla práctica: una alternativa debe ser más sencilla que aquello a lo que sustituye, no debe depender del componente que ha fallado y debe ejercitarse con regularidad, por ejemplo en game days (Sección 9.8), o no debería existir.

Idempotencia: lo que hace seguros los reintentos y las solicitudes de cobertura

La idempotencia es la propiedad de que hacer algo dos veces tenga el mismo efecto que hacerlo una vez. Todos los patrones de esta sección que repiten trabajo dependen de ella: los reintentos tras un timeout, las solicitudes de cobertura, la nueva entrega de mensajes tras la caída de un consumidor y los reprocesamientos desde una cola de mensajes fallidos. La Sección 6.5 mostró cómo una clave de idempotencia convierte `POST /orders` en un reintento seguro, y la Sección 6.4, cómo los consumidores idempotentes eliminan los mensajes duplicados. La regla arquitectónica es sencilla: antes de añadir un reintento, comprueba que la operación es idempotente y, si no lo es, añade primero una clave o una comprobación de duplicados. Una política de reintentos sobre una llamada de pago no idempotente no es un patrón de resiliencia; es un generador de cobros dobles.

Solicitudes de cobertura

Los reintentos se ocupan de los fallos; las solicitudes de cobertura (hedged requests) se ocupan de la lentitud. La Sección 2.4 las presentó a partir de *The Tail at Scale* (2013), de Dean y Barroso: envía una solicitud a una réplica y, si no ha respondido dentro de, por ejemplo, la latencia del percentil 95, envía una copia a una segunda réplica y usa la respuesta que llegue primero, cancelando la otra. Como solo se duplica el pequeño porcentaje de solicitudes más lentas, la carga adicional es pequeña, mientras que la cola se reduce de forma espectacular, como mostró la prueba de rendimiento citada en la Sección 2.4.

Las solicitudes de cobertura tienen condiciones previas. La operación debe ser idempotente, normalmente una lectura. Debe haber réplicas independientes, para que sea improbable que la segunda copia sea lenta por el mismo motivo. Las solicitudes adicionales deben estar presupuestadas y desactivarse en situaciones de sobrecarga, porque cuando todas las réplicas son lentas, las solicitudes de cobertura duplican la carga de un sistema que ya tiene dificultades, exactamente igual que un reintento sin presupuesto. Usadas con cuidado en rutas con muchas lecturas y fan-out, como la búsqueda y las consultas de la carta, son una de las herramientas más eficaces contra la latencia de cola.

Los patrones de un vistazo

Patrón	Protege frente a	Principal costo o riesgo
Timeout	Esperar indefinidamente; recursos retenidos por llamadas muertas	Demasiado corto hace fallar llamadas lentas sanas; demasiado largo no protege nada
Propagación de plazos (deadline propagation)	Trabajo que continúa después de que quien llama se ha rendido	Requiere soporte en cada servicio y biblioteca de la ruta
Reintento con backoff y jitter	Fallos transitorios	Carga adicional; duplicados salvo que sea idempotente
Presupuesto de reintentos	Tormentas de reintentos que multiplican la carga	Algunos fallos transitorios no se reintentan en los periodos malos
Circuit breaker (disyuntor)	Llamar repetidamente a una dependencia que está caída	Ajuste; puede cortar una dependencia que solo falla en parte
Bulkhead (mamparo)	Que una dependencia o un llamante consuma todos los recursos	La capacidad reservada no se puede compartir
Alternativa y degradación	Fallo total de una función	Rutas de código sin probar; resultados obsoletos o aproximados
Solicitud de cobertura (hedged request)	Latencia de cola por réplicas lentas	Carga adicional; solo lecturas

Lista de comprobación: cada llamada remota

Revisa cada punto de integración, incluidas las bases de datos, las cachés, los brokers y las API de terceros, con esta lista.

- [ ] **Los timeouts de conexión y de solicitud se fijan explícitamente**
  Elegidos a partir de los percentiles altos medidos de la dependencia y del plazo propio de quien llama, nunca dejados en el valor predeterminado de una biblioteca.
- [ ] **Se propaga el plazo restante**
  Las llamadas posteriores reciben el tiempo que queda, y el trabajo se detiene cuando se agota.
- [ ] **Los reintentos son limitados y seguros**
  Solo errores transitorios, solo operaciones idempotentes o con clave de idempotencia, en una sola capa, con backoff exponencial limitado y jitter.
- [ ] **Un presupuesto de reintentos limita la carga adicional**
  Los reintentos siguen siendo una pequeña fracción del tráfico (por ejemplo, el 10%), así que una caída no puede multiplicar la carga.
- [ ] **Un circuit breaker protege la dependencia**
  Los umbrales, el tiempo de espera y lo que cuenta como fallo están ajustados, y su estado se exporta como métrica.
- [ ] **La concurrencia hacia la dependencia está acotada**
  Un bulkhead (pool dedicado o semáforo) impide que esta dependencia consuma todos los hilos o conexiones.
- [ ] **El comportamiento ante fallos está decidido**
  Una alternativa, una respuesta degradada o un error claro, acordados con el responsable de producto, y fallar de forma cerrada en las decisiones de seguridad.
- [ ] **Los timeouts, los reintentos, los disparos del breaker y las alternativas son observables**
  Se cuentan, se representan en gráficos y generan alertas, para que se detecte la degradación silenciosa (Sección 9.6).
- [ ] **La ruta de fallo está probada**
  La inyección de fallos o un game day demuestran que los timeouts, el breaker y la alternativa funcionan de verdad (Sección 9.8).

ATENCIÓN

Errores comunes

- Dejar los timeouts en los valores predeterminados de la biblioteca, que a menudo son infinitos o de varios minutos.
- Timeouts posteriores más largos que los anteriores, de modo que el trabajo continúa después de que quien llama se ha ido.
- Reintentar en todas las capas, reintentar operaciones no idempotentes o reintentar sin jitter.
- Contar las respuestas 4xx como fallos en un circuit breaker, de modo que un solo cliente con errores lo dispara para todos.
- Un único pool de hilos o de conexiones compartido para todas las dependencias, de modo que la más lenta decide por todas.
- Alternativas que nunca se ejercitan y que fallan, o sobrecargan otra cosa, la primera vez que se necesitan.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ Un servicio recibe 50 solicitudes por segundo, y cada una llama a una dependencia con un timeout de 4 segundos. Si la dependencia se cuelga, ¿cuántos hilos pueden retener estas llamadas?

Según la ley de Little, unos 50 × 4 = 200 hilos, y por eso el timeout y el límite de un bulkhead deben ajustarse al tamaño del pool.

▼ ¿Por qué el jitter completo es mejor que el backoff exponencial simple cuando muchos clientes fallan a la vez?

El backoff simple hace que todos los clientes reintenten en los mismos momentos, en oleadas sincronizadas; los retrasos aleatorios reparten los reintentos, de modo que la dependencia ve una carga uniforme y más baja.

▼ ¿En qué estado no hace un circuit breaker ninguna llamada remota, y qué pone fin a ese estado?

En el estado abierto; cuando vence el periodo de espera pasa a semiabierto y deja pasar llamadas de prueba.

▼ **¿Por qué una comprobación de autorización no debe recurrir a permitir el acceso cuando el servicio de autorización está caído?**

Las decisiones de seguridad deben fallar de forma cerrada: permitir el acceso durante una caída dejaría entrar a cualquiera siempre que el servicio fallara o fuera atacado.

FUENTES

- Marc Brooker: Exponential Backoff And Jitter (AWS Architecture Blog, 2015): https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/
- Martin Fowler: CircuitBreaker: https://martinfowler.com/bliki/CircuitBreaker.html
- Google SRE book: Addressing Cascading Failures: https://sre.google/sre-book/addressing-cascading-failures/
- Microsoft Azure Architecture Center: Bulkhead pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead
- Dean and Barroso: The Tail at Scale (Google Research): https://research.google/pubs/pub40801/

SECCIÓN 9.5 DE 9.8 · INTERMEDIO · 24 MIN DE LECTURA

SLI, SLO, SLA y presupuestos de errores

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.4 Rendimiento y escalabilidad
- 2.5 Disponibilidad y fiabilidad

Términos anteriores que utiliza: Disponibilidad, Fiabilidad, Percentil (p95, p99), Latencia

LA IDEA EN BREVE

Imagina una empresa de autobuses que promete "casi todos los autobuses a su hora". Para que eso sea útil, debe decir qué cuenta como a su hora (con un margen de dos minutos), cómo lo mide (en cada parada) y qué significa "casi todos" (99 de cada 100 a lo largo de un mes). Si no lo cumple, el equipo de horarios deja de probar rutas nuevas y primero arregla la puntualidad. Como la perfección es imposible y muy cara, ese autobús de cada cien que llega tarde es un margen que la empresa puede gastar en obras y en rutas nuevas. La fiabilidad del software funciona exactamente así.

LO QUE NECESITAS SABER

- Un SLI es una medición centrada en el usuario, que se expresa mejor como eventos buenos divididos entre eventos válidos; un SLO es un objetivo para ese SLI durante una ventana de tiempo; un SLA es un contrato con consecuencias, fijado de forma menos exigente que el SLO.
- Los SLI útiles cubren disponibilidad, latencia y calidad en los servicios basados en solicitudes; frescura, corrección y cobertura en los pipelines; y durabilidad en el almacenamiento, medidos tan cerca del usuario como sea práctico.
- El presupuesto de errores es uno menos el SLO: un 99.9% en 30 días permite 43.2 minutos de caída total o el 0.1% de las solicitudes, y una política de presupuesto de errores por escrito establece qué ocurre cuando se agota.
- La tasa de consumo (burn rate) es la proporción de errores observada dividida entre la proporción permitida; el presupuesto gastado es igual a la tasa de consumo por la ventana de alerta dividida entre la ventana del SLO, de modo que una tasa de consumo de 14.4 durante una hora gasta el 2% de un presupuesto de 30 días.
- Las alertas multiventana y de múltiples tasas de consumo del SRE Workbook avisan a la persona de guardia cuando se gasta el 2% del presupuesto en una hora o el 5% en seis horas, y abren un ticket cuando se gasta el 10% en tres días; cada condición se confirma

con una ventana corta que dura una duodécima parte.

- Los errores habituales son los objetivos del 100%, tener demasiados SLO, SLI que los usuarios no perciben, medir en el lugar equivocado y SLO sin ninguna política ni alertas que los respalden.

Tres términos, una idea

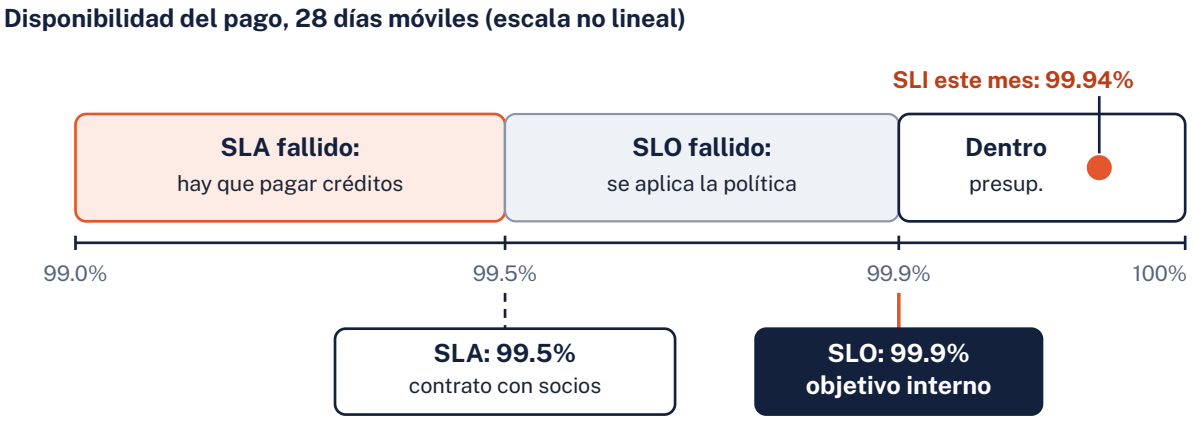
El Capítulo 2 presentó los objetivos de disponibilidad y los nueves (Sección 2.5) y mostró que un 99.9% en 30 días deja 43.2 minutos. Esta sección convierte esa aritmética en una disciplina operativa. El vocabulario procede del libro de Google *Site Reliability Engineering* (Beyer, Jones, Petoff y Murphy, O'Reilly, 2016) y de su complemento, *The Site Reliability Workbook* (2018), y hoy se usa mucho más allá de Google.

Un indicador de nivel de servicio (SLI) es una medida cuantitativa, definida con cuidado, de algún aspecto del servicio que reciben los usuarios. La forma más útil es una proporción: eventos buenos divididos entre eventos válidos. “La proporción de solicitudes del proceso de pago que devuelven una respuesta correcta en menos de 800 ms, medida en el balanceador de carga” es un SLI. Va del 0% (nada funciona) al 100% (nada está roto), y la forma de proporción permite leer, comparar y usar para alertas cualquier SLI de la misma manera.

Un objetivo de nivel de servicio (SLO) es una meta para un SLI durante una ventana establecida: “el 99.9% de las solicitudes del proceso de pago son buenas, medido durante 28 días móviles.” Es una promesa interna que acuerdan el equipo, el product owner y las personas que dependen del servicio. Un acuerdo de nivel de servicio (SLA) es un contrato con los clientes que establece consecuencias, normalmente créditos de servicio, si no se alcanza un nivel. Muchos servicios tienen SLO pero no SLA; las plataformas internas casi nunca tienen SLA.

La idea que une los tres es el presupuesto de errores. Si el objetivo es el 99.9%, el 0.1% de los eventos puede ser malo sin que nadie haya fallado. Ese 0.1% no es un desperdicio; es el margen que tiene el equipo para asumir riesgos: lanzar funcionalidades, ejecutar migraciones, experimentar con la configuración. El presupuesto convierte una discusión sobre sensaciones (“¿es el servicio lo bastante fiable?”) en una discusión sobre números (“¿hemos gastado nuestro presupuesto?”), y da a producto y a ingeniería una regla compartida para saber cuándo avanzar rápido y cuándo frenar.

SLI, SLO y SLA en una misma escala



El SLI es la medición, el SLO la meta interna y el SLA un mínimo contractual más laxo; el hueco entre el SLO y el SLA es la zona de aviso.

Elegir buenos SLI

Un SLI es útil en la medida en que se acerca a lo que experimentan los usuarios. El Workbook ofrece un menú breve que cubre la mayoría de los servicios. Los servicios basados en solicitudes (API, páginas web) necesitan **disponibilidad** (la proporción de solicitudes válidas atendidas correctamente), **latencia** (la proporción atendida más rápido que un umbral) y, a veces, **calidad** (la proporción atendida sin degradación, como una carta mostrada sin recomendaciones personalizadas porque esa dependencia estaba caída). Los pipelines de datos necesitan **frescura** (la proporción de datos actualizados más recientemente que un umbral), **corrección** (la proporción de la salida que es correcta) y **cobertura** (la proporción de la entrada que se procesa). Los sistemas de almacenamiento añaden **durabilidad** (la proporción de los datos escritos que aún pueden leerse).

Los SLI de latencia se escriben como proporciones, no como promedios: “el 99% de las solicitudes de la carta, en menos de 300 ms”. Esto conecta directamente con los percentiles de la Sección 2.4: decir que el 99% de las solicitudes debe tardar menos de 300 ms equivale a decir que el p99 debe ser inferior a 300 ms, pero la forma de proporción se suma limpiamente a lo largo del tiempo y entre servidores (se cuentan las buenas y las válidas), mientras que los percentiles no. Es habitual fijar dos umbrales de latencia, por ejemplo el 90% por debajo de 200 ms y el 99% por debajo de 800 ms, para captar tanto la experiencia típica como la de la cola.

Dónde mides importa tanto como qué mides. El Workbook separa la *especificación del SLI* (el resultado relevante para el usuario, como “pedidos realizados correctamente”) de la *implementación del SLI* (cómo lo cuentas: logs del servidor, métricas del balanceador de carga, sondas sintéticas o instrumentación en la app móvil). Medir dentro del servidor de aplicaciones

pasa por alto las solicitudes que nunca llegaron a él porque falló el DNS, la red de distribución de contenidos o el balanceador de carga. Medir en el cliente lo ve todo, pero incluye las malas conexiones móviles de los usuarios y llega con retraso. La mayoría de los equipos empieza con las métricas del balanceador de carga, que son baratas y están cerca del borde, y añade sondas sintéticas para las rutas con poco tráfico.

Define con precisión qué cuenta como *válido* y qué cuenta como *bueno*, porque los casos límite deciden si el número significa algo. ¿Es malo un 404 por una URL mal escrita? Normalmente no: es una respuesta correcta. ¿Es mala una respuesta 429 por limitación de tasa? Depende de si el cliente estaba dentro de su cuota acordada. ¿Son eventos válidos las comprobaciones de estado y las pruebas de carga? No; exclúyelas, o diluirán la proporción. Escribe estas reglas junto al SLO.

SLI para los ejemplos recurrentes

Recorrido	Tipo de SLI	Especificación del SLI (buenos / válidos)	SLO de ejemplo
Reparto de comida: explorar cartas	Latencia	Solicitudes de la carta respondidas en menos de 300 ms / todas las solicitudes válidas de la carta	99% en 28 días
Reparto de comida: hacer un pedido	Disponibilidad	Solicitudes de pedido respondidas con éxito o con un rechazo de negocio correcto / todas las solicitudes válidas de pedido	99.9% en 28 días
Reparto de comida: ubicación del repartidor	Frescura	Vistas del mapa cuya posición del repartidor tiene menos de 30 s / todas las vistas del mapa	99% en 28 días
Reparto de comida: pagos a restaurantes	Corrección	Lotes de pagos conciliados sin discrepancias / todos los lotes de pagos	99.99% en un trimestre
Venta de entradas: proceso de pago en la puesta a la venta	Latencia y disponibilidad	Reservas de asientos confirmadas en 2 s / todos los intentos de reserva de fans admitidos	99.5% durante cada puesta a la venta
Venta de entradas: entrega de la entrada electrónica	Frescura	Entradas en la app del fan en los 5 minutos siguientes al pago / todos los pedidos pagados	99.9% en 28 días

Fijar el objetivo

Un objetivo es una decisión de negocio, no técnica. El costo de cada nueve adicional sube abruptamente, mientras que los usuarios dejan de notar las mejoras a partir de cierto punto, porque su propia red telefónica, su wifi y sus dispositivos fallan con más frecuencia que eso. El

libro de SRE de Google lo dice sin rodeos: aspirar al 100% es un error para casi todo, porque no deja presupuesto para el cambio, cuesta una fortuna y, de todos modos, no se puede conseguir. Un objetivo del 100% también convierte cada fallo en un incumplimiento de la política, lo que enseña a la gente a ignorar el SLO.

Buenos puntos de partida son el rendimiento pasado (¿qué logró el servicio durante el último trimestre, y estaban contentos los usuarios?), los objetivos de los servicios de los que dependes (no puedes prometer un 99.99% sobre una base de datos del 99.9%, como mostró la Sección 6.1 para las cadenas) y el costo de negocio de la falta de fiabilidad. En la plataforma de reparto de comida, los pedidos fallidos a la hora de comer cuestan ingresos y confianza de los clientes, así que un 99.9% en la realización de pedidos es defendible; las actualizaciones de la posición del repartidor son menos críticas y basta un 99%. Empieza un poco por debajo del rendimiento actual, para que el equipo no incumpla el objetivo de inmediato, y endurécelo más adelante con datos.

Elige la ventana a conciencia. Una ventana móvil, como los últimos 28 días, se ajusta a cómo recuerdan los usuarios las caídas (no las olvidan el día uno del mes), y 28 días contienen exactamente cuatro veces cada día de la semana, lo que mantiene justos los patrones semanales. Los meses naturales son adecuados para los informes y los contratos. Los servicios basados en eventos, como una puesta a la venta de entradas, pueden necesitar un SLO por evento, porque una hora de fallos durante la puesta a la venta importa mucho más que una semana tranquila.

Los presupuestos de errores en números

El presupuesto de errores puede contarse en tiempo o en eventos. Los presupuestos basados en tiempo son adecuados para servicios que o están disponibles o están caídos; los basados en solicitudes son más justos para servicios con tráfico irregular, porque una caída a las 03:00 afecta a menos usuarios que una a mediodía.

EJEMPLO PRÁCTICO

Ejemplo práctico: el presupuesto de la realización de pedidos

La API de pedidos de la plataforma de reparto de comida tiene un SLO del 99.9% de solicitudes buenas en 30 días. Gestiona unos 3 millones de solicitudes de pedido válidas en ese periodo, muy concentradas a la hora de la comida y de la cena.

Presupuesto como fracción	1 - 0.999 = 0.001, es decir, 0.1%
Presupuesto en tiempo	0.001 × 30 días × 1440 minutos = 43.2 minutos de caída total
Presupuesto en solicitudes	0.001 × 3 000 000 = 3000 solicitudes malas
Incidente 1: despliegue defectuoso a las 15:00	12 minutos × 800 solicitudes por minuto = 9600 solicitudes; falla el 40% = 3840 solicitudes malas
Resultado	3840 es más que todo el presupuesto de 3000 solicitudes: un solo incidente agota el mes
El mismo incidente a las 03:00	12 × 50 solicitudes por minuto × 0.4 = 240 solicitudes malas, el 8% del presupuesto

Los presupuestos basados en solicitudes ponderan los fallos según a cuántos usuarios afectan, y por eso los mismos 12 minutos pueden costar el 8% o más del 100% del presupuesto.

La visión basada en tiempo (43.2 minutos) habría puntuado ambos incidentes igual, con 12 × 0.4 = 4.8 minutos de caída equivalente, alrededor del 11% del presupuesto.

Políticas de presupuesto de errores

Un presupuesto no significa nada si no ocurre algo cuando se agota. Una política de presupuesto de errores es un documento breve, acordado de antemano por ingeniería, producto y dirección, que dice qué hará el equipo. La política de ejemplo del Workbook mide el presupuesto durante las cuatro semanas anteriores; si el servicio supera su presupuesto, se detienen todos los cambios y lanzamientos salvo las correcciones urgentes (incidencias P0) y las de seguridad hasta que el servicio vuelva a estar dentro de su SLO. También exige un postmortem con al menos una acción de máxima prioridad siempre que un solo incidente consuma más del 20% del presupuesto de cuatro semanas, y eleva los desacuerdos al director de tecnología.

El valor de una política está en que se acuerda antes de que haga efecto. En plena carrera por un lanzamiento, nadie quiere congelar los cambios; con una política firmada, la congelación es una regla acordada de antemano y no una pelea entre un SRE y un product manager. Las buenas políticas también dicen qué *no* cuenta en contra del equipo: una caída de red de toda la empresa, una dependencia que también está congelada, o tráfico fuera del alcance del SLO, como una prueba de carga.

Los profesionales no se ponen de acuerdo sobre lo mecánica que debe ser la política. Algunos equipos la aplican de forma estricta, porque las excepciones erosionan la confianza en todo el sistema. Otros tratan el presupuesto como información para una conversación y sostienen que una congelación rígida puede retrasar justo las correcciones que restaurarían la fiabilidad. Es habitual un camino intermedio: congelar automáticamente los lanzamientos de funcionalidades arriesgadas, seguir entregando las correcciones de fiabilidad y permitir que un directivo designado conceda excepciones documentadas. El caso contrario también importa: un equipo que nunca gasta su presupuesto probablemente es demasiado prudente y podría entregar más rápido, hacer más experimentos o reducir sus costos.

La tasa de consumo y las alertas basadas en SLO

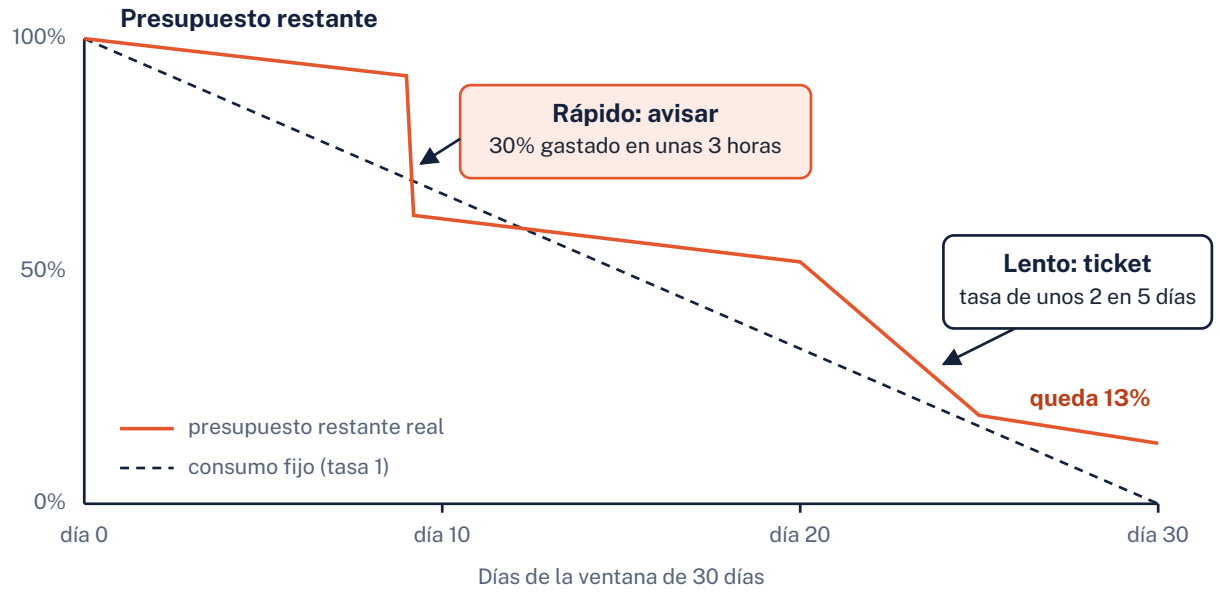
Alertar sobre tasas de error brutas (“avisar si falla más del 1% de las solicitudes durante 5 minutos”) genera mucho ruido: los picos breves despiertan a la gente por la noche por problemas que nunca amenazarían el SLO, mientras que una tasa de fallos constante del 0.5% puede agotar el mes en silencio sin cruzar nunca el umbral. La respuesta del Workbook es alertar sobre la tasa de consumo (burn rate): la rapidez con que se gasta el presupuesto comparada con el ritmo que lo agotaría justo al final de la ventana.

Tasa de consumo = proporción de errores observada / proporción de errores permitida. Con un SLO del 99.9%, la proporción permitida es el 0.1%, así que una proporción de errores del 0.1% es una tasa de consumo de 1 (el presupuesto dura exactamente la ventana), el 1% es una tasa de consumo de 10 y el 100% (una caída total) es una tasa de consumo de 1000. La fracción del presupuesto de la ventana que se gasta durante una ventana de alerta es tasa de consumo x ventana de alerta / ventana del SLO. Para una ventana de 30 días (720 horas), una tasa de consumo de 14.4 mantenida durante una hora gasta 14.4 × 1 / 720 = 0.02, es decir, el 2% del presupuesto del mes; a ese ritmo, todo el presupuesto se agotaría en 720 / 14.4 = 50 horas.

El Workbook recomienda una alerta multiventana y de múltiples tasas de consumo como opción razonable por defecto para un SLO del 99.9%: avisar a la persona de guardia (page) si se gasta el 2% del presupuesto en una hora (tasa de consumo 14.4) o el 5% en seis horas (tasa de consumo 6), y abrir un ticket si se gasta el 10% en tres días (tasa de consumo 1). Cada condición debe cumplirse en una ventana larga y en una corta, siendo la corta una duodécima parte de la larga (5 minutos, 30 minutos y 6 horas, respectivamente). La ventana larga

garantiza que el problema es significativo; la corta, que sigue ocurriendo, de modo que la alerta deja de dispararse a los pocos minutos de una corrección en lugar de seguir en rojo durante una hora.

Consumos rápidos y lentos



Un consumo rápido gasta una gran parte del presupuesto en horas y debería despertar a alguien; un consumo lento lo gasta a lo largo de días y corresponde a un ticket.

```
SLO = 0.999                # 99.9% of requests good
PERIOD_H = 30 * 24         # 720-hour SLO window

def burn_rate(bad, total, slo=SLO):
    """Observed error ratio divided by the ratio the SLO allows."""
    if total == 0:
        return 0.0
    return (bad / total) / (1 - slo)

def budget_spent(rate, window_h, period_h=PERIOD_H):
    """Fraction of the whole window's budget used in window_h hours."""
    return rate * window_h / period_h

def page(w1h, w5m, w6h, w30m):
    """Each argument is (bad, total) for that window."""
    fast = burn_rate(*w1h) >= 14.4 and burn_rate(*w5m) >= 14.4
    slow = burn_rate(*w6h) >= 6 and burn_rate(*w30m) >= 6
    return fast or slow

print(round(burn_rate(bad=288, total=20_000), 1))   # 14.4
print(budget_spent(14.4, window_h=1))               # 0.02, i.e. 2%
```

Tasa de consumo, presupuesto gastado y una regla de aviso con dos ventanas. En producción, la misma lógica se escribe como reglas de alerta en el sistema de monitorización (por ejemplo, reglas de grabación y de alerta de Prometheus).

Los umbrales también fijan el tiempo de detección. Si la API de pedidos falla por completo (proporción de errores del 100%, tasa de consumo de 1000), la proporción de errores de una hora supera el 1.44% al cabo de 0.0144 × 60 = unos 0.9 minutos, así que el aviso salta en menos de un minuto. Si falla el 2% de las solicitudes (tasa de consumo 20), la proporción de una hora llega al 1.44% al cabo de 0.72 × 60 = unos 43 minutos. Los problemas graves avisan rápido; los leves solo avisan si persisten, que es exactamente el comportamiento que se quiere a las tres de la madrugada.

Los servicios con poco tráfico rompen esta aritmética: con 10 solicitudes por hora, un solo fallo es una proporción de errores del 10% y una tasa de consumo de 100. Los remedios del Workbook son el tráfico sintético para añadir señal, agrupar servicios relacionados de poco tráfico bajo un mismo SLO, hacer que los fallos aislados importen menos mediante reintentos y alternativas, o aceptar una ventana más larga. Para el sistema de venta de entradas, que está casi inactivo entre puestas a la venta y luego extremadamente ocupado, funcionan mejor los SLO por evento con alertas de tasa de consumo durante la ventana del evento que una cifra mensual.

Los SLA y los créditos de servicio

Un SLA traslada el objetivo de la ingeniería al derecho contractual. Define el servicio cubierto, cómo se mide la disponibilidad, las exclusiones (mantenimiento programado, uso indebido por parte del cliente, fuerza mayor), la compensación y cómo la reclama el cliente. La compensación habitual es un crédito de servicio: un porcentaje de la cuota mensual que se devuelve, escalonado según cuánto se haya incumplido el nivel. Los créditos suelen ser pequeños en comparación con la pérdida real del cliente, y por eso los SLA se leen mejor como una señal de la confianza del proveedor que como un seguro; los clientes cuyas pérdidas serían grandes necesitan su propia resiliencia, no un crédito mayor.

Considera un SLA ilustrativo que la plataforma de reparto de comida podría ofrecer a los restaurantes asociados para su API de recepción de pedidos: 99.5% de disponibilidad mensual, con un crédito del 10% sobre la cuota de la plataforma de ese mes por debajo del 99.5% y del 25% por debajo del 99.0%. El SLO interno es el 99.9%. El hueco entre ambos es deliberado: el equipo debería agotar el presupuesto de su SLO, aplicar su política y recuperarse mucho antes de que el contrato esté en riesgo. Nunca fijes el SLO igual al SLA; solo te enterarías de los problemas cuando ya debieras dinero.

De ahí se derivan dos consecuencias arquitectónicas. Primera: un SLA solo puede ser honesto si conoces los SLA de tus propias dependencias (proveedor de nube, procesador de pagos, proveedor de mapas) y cómo se combinan. Segunda: los SLA necesitan pruebas: el método de medición del contrato debe ser uno que puedas obtener de tu propia telemetría, lo que constituye un requisito para el diseño de la observabilidad.

Errores comunes

Demasiados SLO. Un servicio con treinta SLO no tiene ninguno, porque nadie puede razonar sobre ellos ni actuar en consecuencia. Empieza con dos o tres por cada recorrido crítico del usuario: normalmente un SLO de disponibilidad y uno o dos de latencia. **Medir lo que no es.** La utilización de CPU, la profundidad de las colas y el retraso de las réplicas son causas útiles que vigilar, pero no son SLI, porque los usuarios no las experimentan directamente. **Medir en el lugar equivocado.** Las tasas de éxito del lado del servidor parecen perfectas mientras la CDN devuelve errores. **Promediar.** Un SLO de latencia media oculta la cola lenta de la que se quejan los usuarios.

Objetivos sin consecuencias. Un SLO sin política de presupuesto ni alertas de tasa de consumo es un adorno del panel. **Objetivos fijados solo por ingenieros.** El product owner debe estar de acuerdo, porque el SLO intercambia funcionalidades por fiabilidad. **No revisarlos nunca.** Revisa los SLO cada uno o dos trimestres: si los usuarios se quejan mientras el SLO está en verde, el SLI o el objetivo están mal; si el presupuesto nunca se toca, puede que el objetivo sea

demasiado laxo o el equipo demasiado prudente. **Un solo SLO para recorridos muy distintos.** Meter la exploración de la carta y el pago en una sola proporción permite que las abundantes solicitudes baratas oculten los fallos de las escasas y valiosas.

Los SLO conectan con el resto de este capítulo. Las alertas de tasa de consumo son las alertas basadas en síntomas que se recomiendan en observabilidad; los incidentes que consumen presupuesto alimentan las revisiones posteriores al incidente; y la medida de la respuesta de un SLO es una función de aptitud natural para la arquitectura.

SI ESTO ES NUEVO PARA TI

Si esto es nuevo para ti

Piensa en el presupuesto de errores como una paga mensual para equivocarte. El SLI es el extracto bancario, el SLO es la paga que acordaste con tu familia y el SLA es el contrato de préstamo con el banco, que te cobra si te quedas muy en descubierto. Gastar la paga está bien; para eso está. Quedarte sin ella significa que dejas de comprar caprichos (funcionalidades nuevas) hasta el mes siguiente.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué un SLI suele escribirse como eventos buenos divididos entre eventos válidos?

Porque la proporción siempre está entre el 0% y el 100%, se suma limpiamente entre servidores y a lo largo del tiempo con solo contar, y permite leer, comparar y convertir en presupuesto de errores cualquier SLI de la misma manera.

▼ Un SLO del 99.9% en 30 días registra una tasa de consumo de 6 durante seis horas. ¿Qué fracción del presupuesto se gasta?

6 × 6 / 720 = 0.05, es decir, el 5% del presupuesto de 30 días.

▼ ¿Por qué un SLA debe ser más laxo que el SLO interno?

Para que el equipo detecte un presupuesto de errores agotado y responda mucho antes de que se incumpla el contrato y haya que pagar créditos de servicio.

▼ ¿Qué aporta la ventana corta en una alerta multiventana?

Confirma que el problema sigue ocurriendo, de modo que la alerta se restablece a los pocos minutos de una corrección en lugar de seguir activa durante toda la ventana larga.

FUENTES

- Google SRE book: Service Level Objectives (2016): https://sre.google/sre-book/service-level-objectives/
- The Site Reliability Workbook: Implementing SLOs (2018): https://sre.google/workbook/implementing-slos/
- The Site Reliability Workbook: Alerting on SLOs: https://sre.google/workbook/alerting-on-slos/
- The Site Reliability Workbook: Example error budget policy: https://sre.google/workbook/error-budget-policy/

SECCIÓN 9.6 DE 9.8 · INTERMEDIO · 26 MIN DE LECTURA

Observabilidad: logs, métricas y trazas

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.4 Rendimiento y escalabilidad
- 6.1 Por qué distribuir
- 9.5 SLO y presupuestos de errores

Términos anteriores que utiliza: Percentil (p95, p99), Objetivo de nivel de servicio (SLO), Indicador de nivel de servicio (SLI), Utilización, Datos personales

LA IDEA EN BREVE

El panel de instrumentos de un automóvil muestra unos pocos indicadores y luces de advertencia: velocidad, combustible, temperatura del motor. Eso es monitorización: te informa de problemas que alguien previó. Cuando el vehículo hace un ruido extraño que ninguna luz explica, un mecánico conecta una computadora de diagnóstico y lee registros detallados de lo que hacía cada pieza, para poder averiguar una avería que nadie había anticipado. La observabilidad consiste en construir el software para que este tipo de investigación sea siempre posible: cada parte guarda registros claros, y los registros de un mismo trayecto pueden seguirse de una parte a otra.

LO QUE NECESITAS SABER

- La monitorización comprueba condiciones predefinidas; la observabilidad es la capacidad de explicar comportamientos nuevos e imprevistos a partir de la telemetría existente, y debe incorporarse desde el diseño mediante una instrumentación coherente e identificadores compartidos.
- Las métricas son agregados baratos para tendencias y alertas, los logs estructurados registran eventos individuales, las trazas muestran el recorrido de una solicitud por los servicios y los perfiles muestran dónde gasta recursos el código.
- Los contadores, los gauges y los histogramas son los tipos de métricas básicos; los percentiles no pueden promediarse entre servidores ni a lo largo del tiempo, pero los buckets de los histogramas sí pueden sumarse y el percentil leerse del resultado combinado.
- RED (tasa, errores, duración) describe los servicios, USE (utilización, saturación, errores) describe los recursos, y las cuatro señales doradas son latencia, tráfico, errores y saturación.

- El trazado distribuido (Dapper, 2010) enlaza los spans mediante la propagación de contexto, estandarizada por la cabecera traceparent del W3C; OpenTelemetry, un proyecto de la CNCF graduado en 2026, es el estándar neutral respecto al proveedor para producir trazas, métricas y logs.
- Los valores de etiqueta no acotados disparan la cardinalidad y el costo de las métricas, así que el detalle por usuario o por pedido corresponde a las trazas y los logs, y el muestreo al final (tail-based) conserva las trazas que importan.
- Avisa a la persona de guardia por síntomas que notan los usuarios, como las tasas de consumo del SLO, no por causas como la CPU, y construye cada panel para responder a una pregunta para un público concreto.

Monitorización y observabilidad

La monitorización consiste en recoger mediciones y comprobaciones predefinidas y dar la alarma cuando cruzan un umbral: disco lleno al 90%, tasa de errores por encima del 1%, comprobación de estado que falla. Responde a preguntas que sabías de antemano que había que hacer. La observabilidad es una propiedad del sistema: lo bien que puedes entender su estado interno, incluidos estados que nadie predijo, a partir de la telemetría que ya emite, sin desplegar código nuevo para averiguarlo. La palabra viene de la teoría de control, donde Rudolf Kálmán la definió en 1960 como la capacidad de inferir el estado interno de un sistema a partir de sus salidas.

La distinción importa porque los sistemas distribuidos fallan sobre todo de formas nuevas. Un pedido de reparto de comida puede ir lento porque la app de tableta de un restaurante reintenta de forma agresiva, porque el endpoint de un proveedor de pagos en una región está degradado, o porque una nueva funcionalidad de la carta hace que una consulta sea lenta solo para los restaurantes con más de 500 platos. Nadie creó un panel para ninguno de estos casos. La monitorización te dice *que* el proceso de pago va lento; la observabilidad te permite preguntar *para quién, dónde y por qué*, segmentando la telemetría por cualquier atributo hasta que aparece el patrón.

Los profesionales no se ponen de acuerdo sobre hasta dónde llega la distinción. Una visión, común entre los proveedores de herramientas, trata la observabilidad como un paraguas sobre tres pilares: logs, métricas y trazas. Charity Majors, Liz Fong-Jones y George Miranda (*Observability Engineering*, O'Reilly, 2022) sostienen que el enfoque de los pilares pierde de vista lo esencial, y que la observabilidad requiere eventos estructurados, ricos y de alta cardinalidad que puedan consultarse en combinaciones arbitrarias. Ambos bandos coinciden en la consecuencia arquitectónica: la observabilidad es un atributo de calidad que debe

incorporarse desde el diseño, mediante una instrumentación coherente, identificadores compartidos y propagación de contexto, porque no se puede añadir a posteriori en mitad de un incidente.

Tipos de telemetría

Señal	Qué es	Lo que mejor hace	Principal costo o límite
Métricas	Series temporales numéricas, agregadas por intervalo y etiquetadas con unas pocas dimensiones	Tendencias a largo plazo baratas, paneles, alertas sobre SLI	Preagregadas: no pueden responder preguntas sobre solicitudes individuales; las etiquetas deben ser de baja cardinalidad
Logs	Registros con marca de tiempo de eventos discretos, idealmente estructurados	Detalle sobre un evento: un error, una decisión, una entrada de auditoría	El volumen y el costo crecen con el tráfico; el texto libre es difícil de consultar
Trazas	Árboles de spans cronometrados que muestran el recorrido de una solicitud por los servicios	Encontrar en qué se va el tiempo y qué salto falló en una llamada distribuida	Deben muestrearse a escala; son inútiles si cualquier salto pierde el contexto
Perfiles	Muestras de dónde gasta el código tiempo de CPU o asigna memoria	Explicar por qué un servicio es lento o caro a nivel de código	Herramientas más recientes; el profiling continuo añade una pequeña sobrecarga
Eventos amplios (wide events)	Un registro con muchos atributos por solicitud y por servicio	Segmentación ad hoc por cualquier atributo (cliente, versión, región)	Necesitan un almacén diseñado para consultas de alta cardinalidad

Logging estructurado (structured logging)

Una línea de log como `Payment failed for order 81F3 after 3001ms` es fácil de escribir y difícil de usar: cada consulta necesita una expresión regular frágil, y cada desarrollador redacta las cosas de forma distinta. El logging estructurado escribe cada entrada como campos clave-valor, normalmente en JSON, con nombres acordados: marca de tiempo, nivel, servicio, nombre del evento, identificadores de negocio (pedido, restaurante, repartidor), resultado, duración y los identificadores de traza y de span que enlazan la línea con su traza. Las consultas se vuelven precisas (“todos los eventos card_authorisation_failed del proveedor cardco en la última hora, agrupados por http_status”), y los logs de distintos servicios pueden combinarse.

Unas pocas reglas mantienen los logs útiles y asequibles. Registra eventos, no prosa; usa un nombre de evento fijo para cada tipo de evento. Usa los niveles de forma coherente, y no registres con nivel de error condiciones sobre las que nadie necesita actuar. Nunca registres secretos, tokens, números de tarjeta completos ni datos personales innecesarios; los pipelines de telemetría se copian, se envían a terceros y se conservan durante meses, así que son una fuente habitual de fugas de datos, y los datos personales en los logs están sujetos al RGPD (GDPR) como cualquier otra copia (Sección 8.7). El logging y la detección de seguridad, un propósito relacionado pero distinto, con sus propias necesidades de retención e integridad, se tratan en la Sección 8.8.

```
traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
             version-trace id (16 bytes)-parent span id (8 bytes)-flags (01 = sampled)

{"ts": "2026-10-22T12:31:07.412Z", "level": "error",
 "service": "payment-service", "event": "card_authorisation_failed",
 "order_id": "ord_81F3", "provider": "cardco", "http_status": 504,
 "duration_ms": 3001, "attempt": 2,
 "trace_id": "4bf92f3577b34da6a3ce929d0e0e4736", "span_id": "00f067aa0ba902b7"}
```

Una cabecera traceparent del W3C y una entrada de log estructurado del servicio de pagos. El trace_id compartido permite que una consulta salte de esta línea de log a la traza completa.

Las métricas y por qué los percentiles no pueden promediarse

La mayoría de los sistemas de métricas, como Prometheus (un proyecto graduado de la CNCF) y el modelo de métricas de OpenTelemetry, ofrecen los mismos pocos tipos. Un contador (counter) solo aumenta, como el total de solicitudes atendidas; se lee como una tasa (solicitudes por segundo durante los últimos cinco minutos), lo que además sobrevive a los reinicios. Un gauge (indicador) es un valor actual que sube y baja: longitud de la cola, conexiones en uso, memoria. Un histograma cuenta las observaciones en buckets (por ejemplo, solicitudes de menos de 50 ms, de menos de 100 ms, de menos de 250 ms, etc.) más un recuento total y una suma, a partir de los cuales se estiman los percentiles. Prometheus también tiene un tipo summary (resumen) que calcula cuantiles dentro de cada proceso.

La distinción entre histogramas y summaries no es un detalle. Como señaló la Sección 2.4, un percentil es una propiedad de toda una distribución, y el percentil de una distribución combinada no puede calcularse a partir de los percentiles de sus partes. Promediar valores p99 entre servidores, o entre minutos, produce un número que no es el percentil de nada. Los recuentos de los buckets, en cambio, pueden simplemente sumarse: se suma cada bucket en todos los servidores y luego se estima el percentil a partir del histograma combinado. Por eso los SLI de latencia de toda la flota deben construirse sobre histogramas (o sobre recuentos de solicitudes por debajo de un umbral, como en la Sección 9.5), nunca sobre summaries por instancia.

EJEMPLO PRÁCTICO

Ejemplo práctico: promediar dos p99

El servicio de la carta se ejecuta en dos instancias, cada una de las cuales gestiona 10 000 solicitudes en un intervalo de cinco minutos. La instancia A está sana: todas sus solicitudes tardan entre 20 y 300 ms, y su p99 es de 180 ms. La instancia B tiene un vecino ruidoso (noisy neighbour): 7000 solicitudes tardan entre 20 y 300 ms, pero 3000 se reparten de manera uniforme entre 500 y 1100 ms.

p99 de B (sus 100 más lentas de 10 000)	1100 - 600 × (100 / 3000) = 1080 ms
Promedio de los dos p99	(180 + 1080) / 2 = 630 ms
p99 real de las 20 000 solicitudes	Las 200 más lentas están todas en B: 1100 - 600 × (200 / 3000) = 1060 ms
Error por promediar	Se informan 630 ms frente a 1060 ms experimentados: el panel subestima la cola en 430 ms
Con histogramas	Sumar los recuentos de los buckets de A y B y leer el percentil 99 de los buckets combinados: unos 1060 ms, limitado solo por la anchura de los buckets

Los percentiles promediados pueden equivocarse en cualquier dirección, y el error crece cuando las instancias soportan tráfico distinto.

Lo mismo se aplica en el tiempo: el p99 de una hora no es el promedio de doce p99 de cinco minutos.

Qué medir: RED, USE y las señales doradas

Tres listas de comprobación breves cubren la mayor parte de lo que un servicio debe exponer. El método RED, propuesto por Tom Wilkie en 2015, se aplica a todo lo que atiende solicitudes: **Rate** (tasa: solicitudes por segundo), **Errors** (errores: solicitudes fallidas por segundo) y **Duration** (duración: la distribución de los tiempos de respuesta). El método USE, de Brendan Gregg, se aplica a los recursos: para cada CPU, disco, enlace de red, pool de conexiones y cola, comprueba la **Utilisation** (utilización: lo ocupado que está), la **Saturation** (saturación: cuánto trabajo espera) y los **Errors** (errores). Las cuatro señales doradas del capítulo del libro de SRE de Google sobre la monitorización de sistemas distribuidos son la **latencia**, el **tráfico**, los **errores** y la **saturación**: en la práctica, RED más saturación.

En la práctica, RED describe lo que experimentan los usuarios y USE explica por qué. Si sube la duración del servicio de pagos (RED), USE aplicado a sus recursos podría mostrar un pool de conexiones al 100% de utilización con 40 solicitudes esperando (saturación), lo que a su vez apunta al lento proveedor de tarjetas que mantiene las conexiones abiertas. Mide la latencia por separado para los éxitos y los fallos: un error rápido no es buena latencia, y un timeout es a la vez un error y una respuesta lenta.

Tres listas de comprobación comparadas

Lista de comprobación	Se aplica a	Señales	Pregunta típica que responde
RED (Wilkie)	Servicios basados en solicitudes	Tasa, errores, duración	¿Está el servicio de pedidos atendiendo bien a los usuarios ahora mismo?
USE (Gregg)	Recursos: CPU, memoria, discos, pools, colas	Utilización, saturación, errores	¿Qué recurso es el cuello de botella?
Cuatro señales doradas (libro de SRE)	Sistemas orientados al usuario	Latencia, tráfico, errores, saturación	¿Está sano el sistema y cuánto le falta para llegar a sus límites?

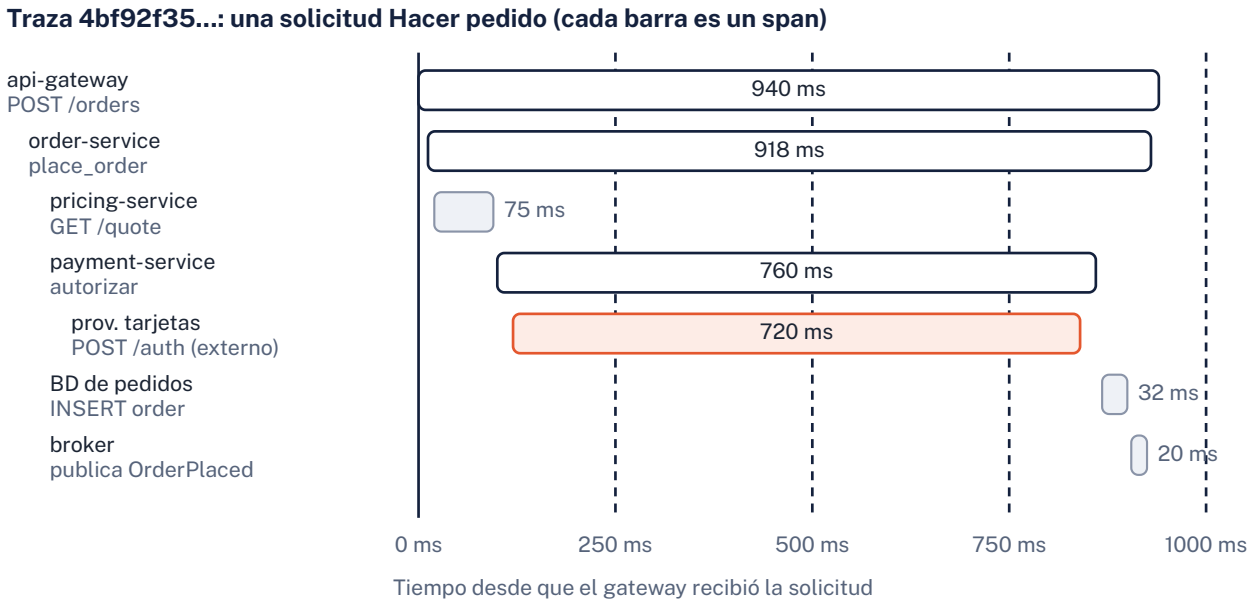
Trazado distribuido

En un monolito, una traza de pila (stack trace) muestra dónde pasó su tiempo una solicitud lenta. Entre servicios no hay una única pila. El trazado distribuido reconstruye una: cada solicitud recibe un identificador de traza en el borde, cada unidad de trabajo se convierte en un span (con un nombre, una hora de inicio, una duración, un estado, atributos y el identificador de su span padre), y un backend de trazado ensambla los spans en un árbol. El artículo de Google sobre Dapper (Sigelman y colaboradores, 2010) describió el enfoque a escala: baja sobrecarga, instrumentación en bibliotecas compartidas en lugar de en el código de la aplicación, y muestreo para que solo se registre una fracción de las solicitudes.

El trazado depende por completo de la propagación de contexto: cada servicio debe pasar el contexto de traza al siguiente. La especificación W3C Trace Context (Recomendación del W3C desde 2020) lo estandariza para HTTP con una cabecera `traceparent` que contiene una versión, un ID de traza de 16 bytes, el ID de 8 bytes del span que llama y flags como si la traza está muestreada, más una cabecera opcional `tracestate` para datos del proveedor. En la mensajería, los mismos valores viajan en las cabeceras de los mensajes, y el span del consumidor hace referencia al del productor. Un solo servicio que pierda la cabecera (una biblioteca antigua, un cliente HTTP escrito a mano, un trabajo por lotes) rompe todas las trazas que pasan por él, y por eso la propagación corresponde a las bibliotecas compartidas de la plataforma, no al código de cada equipo.

Registrar cada span de cada solicitud es caro a escala, así que las trazas se muestrean. El muestreo al inicio (head-based sampling) decide al comienzo de la solicitud (conservar 1 de cada 100) y es barato, pero descarta la mayoría de las solicitudes lentas y fallidas, poco frecuentes, que te interesan. El muestreo al final (tail-based sampling) guarda los spans en un búfer hasta que la solicitud termina y luego conserva todos los errores, todas las solicitudes más lentas que un umbral y una pequeña fracción del resto; necesita más memoria en el collector, pero conserva las trazas interesantes.

Una traza a través de los servicios de reparto de comida



La cascada muestra de inmediato que 720 de los 940 ms se pasan esperando al proveedor externo de tarjetas, no en los servicios propios de la plataforma.

SI ESTO ES NUEVO PARA TI

Paso a paso: seguir un pedido a través de los servicios

La idea más difícil de esta sección es cómo se cose una traza a partir de fragmentos registrados por distintos servicios, ninguno de los cuales ve la solicitud completa. El pegamento son dos identificadores que se transmiten con cada llamada.

- Un cliente hace un pedido. El API gateway recibe POST /orders, asigna a la solicitud un nuevo ID de traza e inicia su propio span: un registro con un nombre, una hora de inicio y, cuando termina, una duración.
- Cuando el gateway llama al servicio de pedidos, añade una cabecera traceparent que lleva el ID de traza y el ID de su propio span. El servicio de pedidos inicia su span place_order y registra el span del gateway como su padre.
- El servicio de pedidos hace lo mismo en cada llamada que realiza, a precios y a pagos, y pone el contexto de traza en las cabeceras del evento OrderPlaced. Cada span nuevo registra el ID de traza y su padre.
- Cada servicio envía por separado sus spans terminados al backend de trazado. El ID de traza compartido permite al backend reunirlos, y los enlaces a los padres le permiten organizarlos en la cascada del diagrama.
- Supón ahora que el servicio de pedidos llamara a precios mediante un cliente HTTP escrito a mano que no añade la cabecera. Precios iniciaría una traza nueva propia, y sus spans faltarían en la cascada del pedido.
- Por eso la propagación de contexto corresponde a las bibliotecas compartidas que usan todos los equipos: un solo salto que la olvide rompe todas las trazas que pasan por él.
- **La lección:** cada servicio registra solo su propia parte de una solicitud. El ID de traza y los enlaces a los padres que viajan en cada llamada convierten esas partes en una sola historia, así que una traza solo es tan completa como su salto más débil.

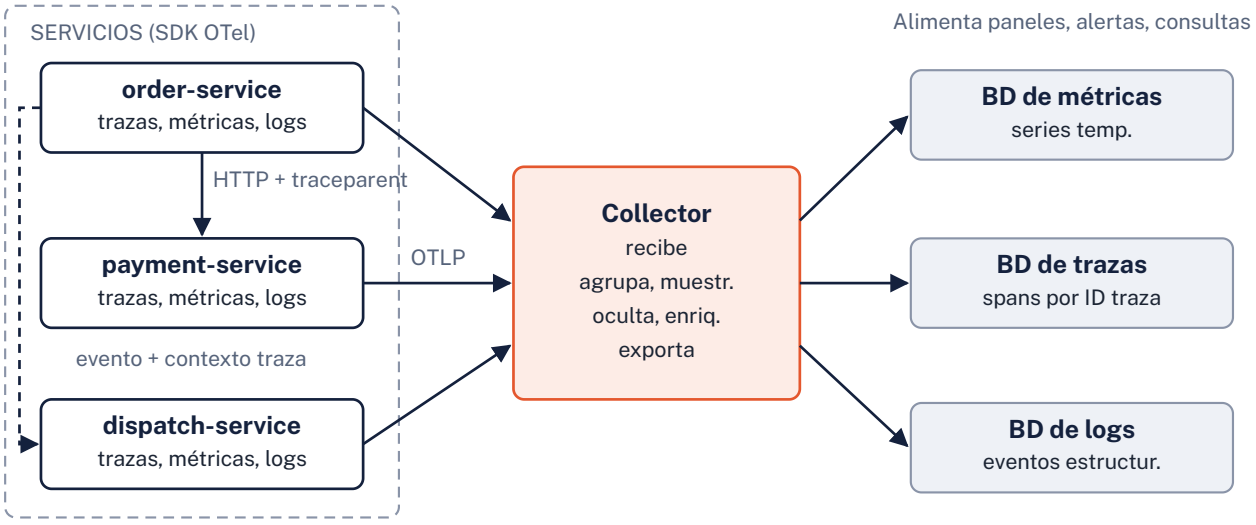
OpenTelemetry: un estándar para la instrumentación

Hasta aproximadamente 2019, cada proveedor de trazado y monitorización tenía sus propios agentes y bibliotecas, así que cambiar de proveedor significaba volver a instrumentar todos los servicios. OpenTelemetry (OTel) se formó en 2019 al fusionar dos proyectos de código abierto anteriores, OpenTracing y OpenCensus, bajo la Cloud Native Computing Foundation (CNCF), y se graduó como proyecto de la CNCF en mayo de 2026. Proporciona API y SDK para los distintos lenguajes, instrumentación automática para frameworks y clientes habituales, el protocolo de transmisión OTLP, el Collector y unas convenciones semánticas que fijan nombres

de atributos como `http.request.method` y `http.response.status_code` para que la telemetría de distintos equipos y lenguajes encaje. Las trazas, las métricas y los logs son estables; los perfiles se añadieron como cuarta señal y entraron en alfa pública en 2026.

El valor arquitectónico está en separar la instrumentación del backend. Los servicios emiten OTLP a un Collector, que normalmente se ejecuta junto a cada nodo o como gateway central; el Collector agrupa en lotes, muestrea, elimina atributos sensibles, añade metadatos como la versión desplegada y exporta a los almacenes que use la organización. Cambiar de proveedor, o enviar los mismos datos a dos backends durante una migración, pasa a ser un cambio de configuración del Collector en lugar de un cambio de código en doscientos servicios. Un equipo de plataforma suele proporcionar la configuración del SDK, la propagación y el Collector como un camino pavimentado (paved road), para que los equipos de producto solo añadan atributos de negocio.

Pipeline de telemetría



La instrumentación vive en los servicios, el procesamiento en el Collector y el almacenamiento en backends sustituibles.

```
from opentelemetry import trace
import structlog

tracer = trace.get_tracer("order-service")
log = structlog.get_logger()

def place_order(order):
    with tracer.start_as_current_span("place_order") as span:
        span.set_attribute("restaurant.id", order.restaurant_id)
        span.set_attribute("order.item_count", len(order.items))
        quote = pricing.quote(order)       # instrumented HTTP client adds
        payment.authorise(order, quote)    # the traceparent header itself
        log.info("order_placed", order_id=order.id,
                 total_pence=quote.total_pence)
        return order.id
```

Instrumentación manual con la API de Python de OpenTelemetry y un logger estructurado. Los clientes HTTP con instrumentación automática crean spans hijos y propagan el contexto sin código adicional.

Cardinalidad y costo

La telemetría puede costar fácilmente más que el sistema que observa. En las métricas, el factor principal es la cardinalidad: cada combinación distinta de valores de etiqueta es una serie temporal separada que hay que almacenar e indexar. Las etiquetas con pocos valores (método HTTP, clase de estado, región) no son problema. Las etiquetas con valores no acotados (ID de usuario, ID de pedido, URL completa con identificadores, texto del mensaje de error) multiplican las series sin límite y pueden tumbar un sistema de métricas.

EJEMPLO PRÁCTICO

Ejemplo práctico: contar series temporales

El servicio de pedidos exporta un histograma de duración de solicitudes con 12 buckets. Prometheus almacena cada bucket como una serie, más una suma y un recuento, así que son 14 series por combinación de etiquetas.

Pares de ruta y método	60
Clases de estado (1xx a 5xx)	5
Instancias	30
Series de esta sola métrica	60 × 5 × 30 × 14 = 126,000
Añadir una etiqueta de región (5 regiones)	126 000 × 5 = 630 000 series
Añadir restaurant_id (40 000 restaurantes)	126 000 × 40 000 = unos 5000 millones de series potenciales: inviable

Las preguntas por restaurante o por cliente corresponden a las trazas, los logs o los eventos amplios, donde se espera una alta cardinalidad, no a las etiquetas de las métricas.

Los exemplars, que adjuntan un ID de traza de muestra a un bucket del histograma, enlazan un pico de latencia en una métrica barata con una traza detallada y cara.

Otras palancas de costo son muestrear las trazas (el muestreo al final conserva las útiles), una retención más corta para los datos detallados y más larga para los agregados, descartar por defecto los logs de depuración en producción, e imputar el costo de la telemetría a los equipos que la generan para que sea visible. Trata el volumen de telemetría como un parámetro de diseño con un presupuesto, como cualquier otro recurso.

Alertar sobre síntomas, no sobre causas

La pauta del libro de SRE es que los avisos a la persona de guardia deben referirse a síntomas que notan los usuarios, no a las muchas causas posibles. Una CPU alta, un disco lleno en una réplica o un contenedor reiniciado son causas: a menudo inofensivas, a veces ya resueltas por la automatización, y nunca una lista completa de lo que puede fallar. Un SLO que se consume, procesos de pago que fallan o un pipeline de posiciones de repartidores detenido son síntomas:

importan sea cual sea su causa. Las alertas basadas en síntomas son menos, rara vez son falsas y cubren fallos que nadie predijo; las señales basadas en causas corresponden a paneles que ayudan a diagnosticar un síntoma una vez que se ha avisado de él.

Cada aviso debe ser urgente, accionable y real. La fatiga de alertas es en sí misma un riesgo para la fiabilidad: los ingenieros de guardia a los que despiertan alertas sin sentido aprenden a descartarlas y pasan por alto la que importa. Una regla práctica es avisar por las tasas de consumo del SLO (Sección 9.5), enviar los consumos más lentos y las advertencias sobre recursos a una cola de tickets, y enlazar cada aviso con un runbook (manual de operaciones) y un panel.

GUÍA DE DECISIÓN

Guía de decisión: ¿debe esta alerta avisar a una persona?

Aplica estas preguntas a cada regla de alerta antes de permitir que despierte a alguien, y de nuevo cada vez que se dispare sin que nadie necesite actuar.

Q1 **¿Significa la condición que los usuarios se ven perjudicados ahora, o lo serán pronto (un SLO que se consume rápido, un recorrido que falla)?**

- **Sí** Continúa con la siguiente pregunta.
- **No** No avises. Ponla en un panel para el diagnóstico, o abre un ticket si alguien debe revisarla en horario laboral.

Q2 **¿Necesita que actúe una persona, en lugar de una respuesta automática como un reinicio o un escalado horizontal?**

- **Sí** Continúa con la siguiente pregunta.
- **No** Automatiza la respuesta y alerta solo si la automatización falla.

Q3 **¿Debe producirse la acción antes del siguiente día laborable?**

- **Sí** Avisa, y enlaza la alerta con un runbook y el panel correspondiente.
- **No** Envíala a la cola de tickets del equipo en lugar de al sistema de avisos.

Q4 **¿Ha avisado esta alerta en el último mes sin que nadie necesitara hacer nada?**

- **Sí** Corrige el umbral o la señal, o elimina la alerta: un aviso sobre el que nadie actúa enseña a la gente a ignorar los avisos.
- **No** Mantenla, y revísala en el siguiente traspaso de guardia.

Paneles que responden preguntas

Un muro de cincuenta gráficos no responde rápido a ninguna pregunta. Los buenos paneles se diseñan como cualquier interfaz de usuario: para un público concreto y una pregunta específica. Una vista general del servicio responde a “¿está sano este servicio?” con sus SLO, el presupuesto de errores restante y las métricas RED arriba, y las métricas USE de sus recursos clave debajo. Un panel de recorrido para la puesta a la venta responde a “¿están consiguiendo entradas los fans?” con las admisiones, las reservas de asientos, los pagos y los fallos por minuto de extremo a extremo. A los paneles de diagnóstico se llega profundizando; no se muestran primero.

Guarda los paneles en el control de versiones como código, asigna a cada uno un propietario, muestra los despliegues y los cambios de feature flags como anotaciones en los gráficos (la mayoría de los incidentes siguen a un cambio) y elimina los paneles que nadie ha abierto en meses. Durante un incidente, la primera vista útil suele ser el recorrido orientado al usuario, la segunda la traza de una solicitud que falla, y solo después las métricas de recursos.

ATENCIÓN

Errores habituales de observabilidad

- Añadir trazado solo a unos pocos servicios: un salto sin propagación rompe todas las trazas que pasan por él.
- Poner ID de usuario, ID de pedido o URL sin procesar en las etiquetas de las métricas.
- Promediar percentiles entre instancias o ventanas de tiempo.
- Medir solo en el servidor, de modo que los fallos del DNS, la CDN o el balanceador de carga son invisibles.
- Registrar datos personales y secretos, lo que convierte el almacén de logs en la copia menos protegida de los datos de los clientes.
- Alertar sobre cada causa, de modo que los síntomas reales se pierden entre el ruido.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué el p99 de toda una flota debe calcularse a partir de histogramas combinados y no a partir del p99 de cada instancia?

Porque el percentil de unos datos combinados no es función de los percentiles de las partes; los recuentos de los buckets se suman con exactitud, así que el histograma combinado da una estimación fiel (limitada por los buckets), mientras que los p99 promediados pueden estar muy equivocados.

▼ ¿Qué se rompe si un servicio de una cadena de llamadas no reenvía la cabecera traceparent?

Se divide cada traza que pasa por él: los spans posteriores a ese salto inician una traza nueva, así que se pierden la visión de extremo a extremo y el vínculo entre causa y síntoma.

▼ ¿Por qué order_id es una mala etiqueta de métrica pero un buen atributo de span?

Como etiqueta, crea una serie temporal nueva por pedido, lo que dispara la cardinalidad y el costo; como atributo de span o de log, se almacena una vez por evento, donde se espera una alta cardinalidad y es útil para las consultas.

▼ ¿Cuál de estas situaciones debería avisar a alguien: la CPU al 95% en una instancia, o el SLO del proceso de pago consumiéndose a una tasa de 20?

El consumo del SLO, porque es un síntoma que notan los usuarios; la lectura de CPU es una causa posible que corresponde a un panel de diagnóstico.

FUENTES

- Google SRE book: Monitoring Distributed Systems: https://sre.google/sre-book/monitoring-distributed-systems/
- Sigelman et al.: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure (Google, 2010): https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/
- W3C Trace Context: https://www.w3.org/TR/trace-context/
- OpenTelemetry documentation: Signals: https://opentelemetry.io/docs/concepts/signals/
- Prometheus documentation: Histograms and summaries: https://prometheus.io/docs/practices/histograms/

SECCIÓN 9.7 DE 9.8 · AVANZADO · 25 MIN DE LECTURA

Planificación de capacidad y pruebas de rendimiento

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.4 Rendimiento y escalabilidad
- 9.1 Escalado
- 9.3 Colas y back pressure

Términos anteriores que utiliza: Margen de capacidad (headroom), Ley de Little, Percentil (p95, p99), Autoescalado, Función de aptitud (fitness function)

LA IDEA EN BREVE

Antes de una gran boda, un servicio de catering calcula cuántos invitados vendrán, averigua cuántos platos puede servir cada cocinero en una hora, añade algo de personal de reserva por si alguien se pone enfermo y encarga los hornos adicionales con semanas de antelación, porque no se pueden comprar el mismo día. Después hace un ensayo a tamaño real para comprobar que la cocina de verdad da abasto. Los equipos de software hacen lo mismo: prevén la demanda, miden cuánto puede soportar cada servidor, añaden capacidad de reserva, encargan lo que tarda en conseguirse y ensayan con clientes simulados realistas.

LO QUE NECESITAS SABER

- La planificación de capacidad prevé la demanda máxima, mide la capacidad por unidad en condiciones realistas, añade margen y redundancia, y encarga con antelación todo lo que tenga un plazo de entrega largo, incluidas las cuotas y los límites de terceros.
- Las pruebas de carga, de estrés, de resistencia (soak), de picos (spike) y de punto de ruptura responden a preguntas distintas: cumplir los objetivos en el pico, el comportamiento ante fallos por encima de él, la degradación con el tiempo, la reacción ante aumentos repentinos y el límite real.
- Los modelos de carga cerrados se ralentizan cuando lo hace el sistema, mientras que en los abiertos las llegadas siguen a un ritmo fijo; el tráfico público y las salidas a la venta (on-sales) son abiertos, por lo que las pruebas con modelo cerrado pueden ser peligrosamente optimistas (Schroeder, Wierman y Harchol-Balter, 2006).
- La omisión coordinada (Gil Tene) oculta las respuestas lentas cuando la herramienta de prueba espera las respuestas antes de enviar; mide desde el momento de envío previsto, con generadores de modelo abierto e histogramas completos.

- k6, Gatling, JMeter y Locust son las principales herramientas de código abierto; los escenarios realistas, los volúmenes de datos, las claves calientes, la latencia de las dependencias y un generador sin sobrecarga importan más que la herramienta.
- Los presupuestos de rendimiento, como los umbrales de Core Web Vitals o un reparto de la latencia por salto, se comprueban automáticamente en el pipeline de entrega como funciones de aptitud.
- El recurso compartido más escaso, a menudo un tercero, fija el ritmo de admisión seguro para un evento como la salida a la venta de entradas.

La capacidad es un plan, no una esperanza

La planificación de capacidad responde de antemano a una pregunta: ¿tendremos suficiente de todo cuando la demanda alcance su pico? “Todo” abarca más que los servidores. Incluye las conexiones a la base de datos y el rendimiento de escritura, la memoria de la caché, las particiones del intermediario de mensajes, las cuotas de la cuenta en la nube, los límites de tasa de terceros, las licencias y las personas de guardia. La elasticidad de la nube permite añadir rápidamente parte de la capacidad, pero no toda: el nodo primario de una base de datos no se puede duplicar en segundos, un proveedor de pagos puede necesitar semanas de aviso para elevar un límite, y el autoescalado (tratado en la Sección 9.1) reacciona en minutos a un salto brusco que llega en segundos.

El método tiene cuatro pasos, que se repiten en un ciclo regular (trimestral para un crecimiento constante, por evento para los lanzamientos y las salidas a la venta):

El ciclo de planificación de capacidad

1. **Prever la demanda**
   Combina el crecimiento orgánico (pedidos diarios que aumentan un 4% al mes), la estacionalidad (picos de comida y cena, las noches de los viernes, Navidad) y los eventos conocidos (una campaña de marketing, la salida a la venta de un concierto en un estadio). Expresa la demanda en las unidades con las que escala el sistema: solicitudes por segundo por recorrido, escrituras por segundo, mensajes por segundo, en el pico, no en la media diaria.

2. **Medir la capacidad por unidad**
   Averigua, mediante pruebas, cuánta carga puede soportar una instancia, una base de datos o una partición sin dejar de cumplir su SLO de latencia. El codo de la curva de tiempo de respuesta, donde el encolamiento empieza a dominar (Sección 2.4), es el límite práctico, no el punto en que se cae.

3 **Añadir margen y redundancia**
Dimensiona para el pico previsto más un margen para el error de previsión, las ráfagas y los reintentos, y de forma que la pérdida de una instancia o de una zona de disponibilidad entera siga dejando capacidad suficiente (redundancia N+1 o a nivel de zona).

4 **Planificar los plazos y revisar**
Haz una lista de lo que tarda en conseguirse (aumentos de cuota, capacidad reservada, actualizaciones de la base de datos, límites de terceros, hardware para sistemas locales) y encárgalo con suficiente antelación. Después compara la demanda real con la prevista y mejora el modelo.

Los cálculos de dimensionamiento con la ley de Little se trabajaron en la Sección 9.1; esta sección se centra en cómo obtener cifras fiables para introducir en ellos. El error más común en la planificación de capacidad no es una mala aritmética, sino una capacidad por instancia medida en condiciones poco realistas, y el segundo es olvidar una restricción compartida, como un límite de un tercero, que ningún margen de capacidad (headroom) en tus propios servidores puede resolver.

Tipos de prueba de rendimiento

Las pruebas de rendimiento son la única forma de medir la capacidad, encontrar los cuellos de botella antes que los clientes y verificar que un lanzamiento no ha empeorado las cosas. Las pruebas se diferencian en la forma de la carga que aplican y en la pregunta que responden. Una prueba de carga comprueba el pico esperado; una prueba de estrés va más allá para ver cómo falla el sistema; una prueba de resistencia (soak test) se ejecuta durante horas para sacar a la luz fugas lentas; una prueba de picos (spike test) comprueba la reacción ante saltos repentinos; una prueba de punto de ruptura aumenta la carga hasta que algo se rompe, para encontrar el límite real.

Tipos de pruebas de rendimiento

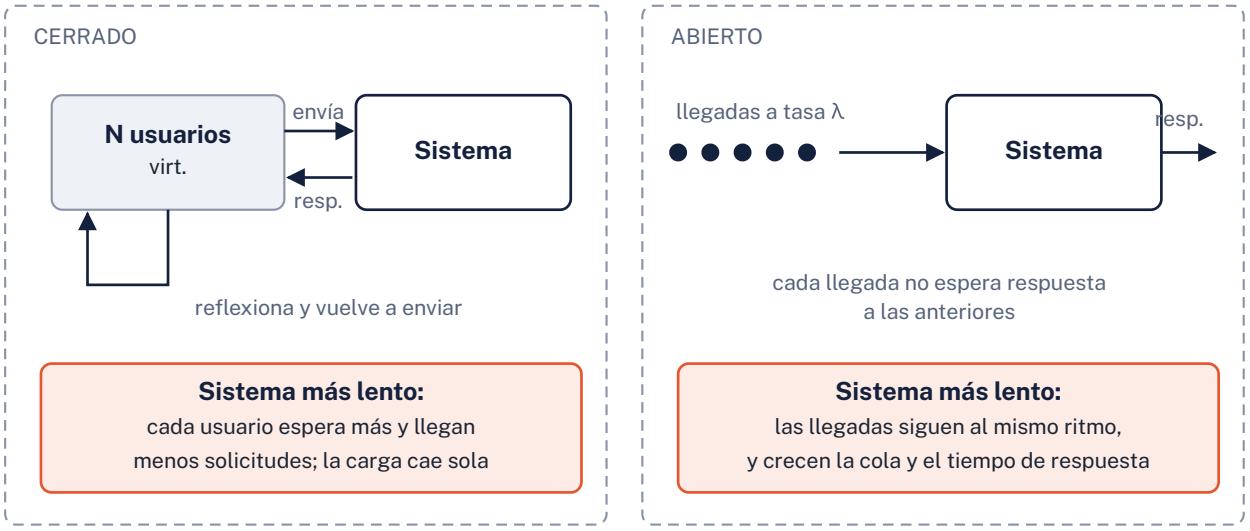
Prueba	Pregunta que responde	Forma de la carga	Duración típica	Qué vigilar
Humo (smoke)	¿Funcionan siquiera el script de prueba y el entorno?	Unos pocos usuarios	Minutos	Errores del script, entorno equivocado
Carga	¿Cumplimos nuestros objetivos de latencia y de errores en el pico esperado?	Subir hasta el pico previsto y mantener	30-60 minutos	Latencia p99, tasa de errores, saturación de recursos
Estrés	¿Cómo falla el sistema por encima del pico, y se recupera?	Subir hasta 1.5-3x el pico y luego bajar	30-60 minutos	Degradación elegante, descarte de carga, tiempo de recuperación
Resistencia (soak)	¿Se degrada algo con el tiempo?	Carga normal o de pico, mantenida	De horas a días	Fugas de memoria, colas que crecen, fugas de conexiones, disco que se llena
Spike (investigación acotada)	¿Podemos absorber un aumento repentino?	Salto de baja a muy alta en segundos	Minutos	Retardo del autoescalado, desbordamiento de colas, comportamiento del limitador de tasa
Punto de ruptura (capacidad)	¿Dónde está el límite y cuál es el primer cuello de botella?	Subida constante hasta el fallo	Hasta que se rompe	El codo de la latencia, el primer recurso saturado

Modelos de carga abiertos y cerrados

Los generadores de carga simulan a los usuarios de una de dos formas, y elegir la equivocada produce resultados que no solo son inexactos, sino peligrosamente optimistas. En un modelo de carga cerrado, la herramienta ejecuta un número fijo de usuarios virtuales; cada uno envía una solicitud, espera la respuesta, hace una pausa durante un tiempo de reflexión y repite. Si el sistema se ralentiza, cada usuario virtual envía menos solicitudes, así que la carga baja por sí sola. En un modelo de carga abierto, las solicitudes llegan a un ritmo fijo, con independencia de lo rápido que se respondan las anteriores, como hacen los visitantes reales de un sitio web público: miles de fans que pulsan Comprar a las 10:00 no se esperan unos a otros.

Bianca Schroeder, Adam Wierman y Mor Harchol-Balter demostraron (“Open Versus Closed: A Cautionary Tale”, NSDI 2006) que los dos modelos pueden dar tiempos de respuesta muy distintos para el mismo sistema y la misma carga media, y que las decisiones de planificación o de capacidad ajustadas con uno pueden ser erróneas con el otro. Los sistemas internos con una población fija de clientes, como una aplicación de centro de llamadas con 300 agentes, son razonablemente cerrados. El tráfico público de internet, los consumidores dirigidos por eventos que se alimentan de una cola y las salidas a la venta son abiertos, o casi. Para esos sistemas, la prueba realista es la de modelo abierto; los ejecutores (executors) `constant-arrival-rate` y `ramping-arrival-rate` de k6, por ejemplo, lo implementan, mientras que su ejecutor `constant-vus` es cerrado.

Cerrado frente a abierto



Una prueba con modelo cerrado retrocede educadamente cuando el sistema tiene dificultades; el tráfico público real no lo hace.

SI ESTO ES NUEVO PARA TI

Paso a paso: una prueba que retrocede educadamente

La idea más difícil de esta sección es que la forma en que un generador de carga envía las solicitudes puede ocultar una sobrecarga. En un modelo cerrado, los usuarios simulados esperan cada respuesta, así que un sistema con dificultades recibe automáticamente menos carga; los fans reales en una salida a la venta no se esperan unos a otros.

- Un equipo prueba el mapa de asientos de la salida a la venta con 300 usuarios virtuales. Cada uno obtiene el mapa, espera la respuesta, hace una pausa de 2.5 segundos y repite. Con respuestas en 0.5 segundos, cada ciclo dura 3 segundos: 300 ÷ 3 = 100 solicitudes por segundo.
- A mitad de la prueba, una consulta lenta a la base de datos hace que cada respuesta tarde 5 segundos. Cada ciclo dura ahora 7.5 segundos, así que los mismos 300 usuarios envían solo 300 ÷ 7.5 = 40 solicitudes por segundo.
- La carga se ha reducido a menos de la mitad justo cuando el sistema tiene dificultades, un respiro que el tráfico real nunca da, así que el informe parece mucho mejor de lo que será la salida a la venta.
- Los fans reales siguen un modelo abierto: siguen llegando 100 por segundo pase lo que pase con las solicitudes anteriores. A 5 segundos cada una, la ley de Little da 100 × 5 = 500 solicitudes en curso, diez veces las 50 normales, y si el sistema no puede terminar 100 por segundo, la cola no deja de crecer.
- Los ejecutores de tasa de llegada de k6 reproducen esto: inician nuevas solicitudes a un ritmo fijo por muy despacio que se respondan las anteriores, siempre que la prueba permita suficientes usuarios virtuales.
- Las pruebas cerradas son adecuadas para una población realmente fija, como una aplicación de centro de llamadas que usan 300 agentes que no envían nada mientras esperan.
- **La lección:** ajusta el modelo a quién envía el tráfico. Para sitios públicos, consumidores de colas y salidas a la venta, prueba a una tasa de llegada fija, o la prueba será más optimista justo donde más importa.

Omisión coordinada (coordinated omission)

Gil Tene, de Azul Systems y autor de la biblioteca HdrHistogram, puso nombre a un error de medición que afecta a muchas pruebas de latencia y herramientas de monitorización: la omisión coordinada (coordinated omission). Un generador de carga en bucle cerrado espera cada respuesta antes de enviar la siguiente solicitud. Cuando el sistema se detiene, el

generador se detiene con él, así que las solicitudes que los usuarios reales habrían enviado durante la parada, y que habrían sido todas lentas, nunca se envían ni se miden. La herramienta de prueba se “coordina” con el sistema que está probando y omite exactamente las muestras malas.

Imagina una prueba que pretende enviar una solicitud cada 10 ms durante 100 segundos contra un servicio que normalmente responde en 1 ms. A los 50 segundos, el servicio se congela durante 10 segundos (por ejemplo, una larga pausa de recolección de basura). La herramienta de prueba en bucle cerrado registra unas 9000 muestras rápidas y una muestra de 10 segundos, así que informa de un p99 de 1 ms e incluso de un p99.9 de 1 ms. Sin embargo, según el calendario previsto, durante la congelación habrían llegado unas 1000 solicitudes, que habrían esperado entre 10 segundos y 10 ms, unos 5 segundos de media. El diez por ciento de todas las solicitudes sufrió un retraso grave: el p95 real es de unos 5 segundos y el p99 real, de unos 9 segundos.

Lo que la herramienta de prueba no ve

La única muestra lenta de la herramienta oculta mil experiencias lentas de usuarios.

Las soluciones son usar generadores de modelo abierto (de tasa de llegada) para los sistemas abiertos, medir la latencia desde el momento en que se *pretendía* enviar la solicitud y no desde cuándo se envió realmente, registrar histogramas completos en lugar de medias y desconfiar de cualquier informe de latencia cuyo máximo esté muy por encima de su p99.9 con pocas muestras entre ambos. Para esto existen precisamente wrk2, la herramienta modificada de Tene para pruebas de rendimiento HTTP, y las opciones de corrección de HdrHistogram. El mismo error aparece en la monitorización de producción cuando una biblioteca cliente mide solo las solicitudes que consiguió enviar.

Herramientas

Cuatro herramientas de código abierto cubren la mayoría de las necesidades. **Apache JMeter** es la veterana: basada en Java, con un editor gráfico de planes de prueba, muchos plug-ins de protocolos y grupos de hilos (thread groups) que son cerrados por defecto. **Gatling** escribe las pruebas como código (Scala, Java y Kotlin, y más recientemente JavaScript y TypeScript), es eficiente con cargas altas y tiene perfiles de inyección abiertos y cerrados. **k6**, mantenida por Grafana Labs, ejecuta scripts de prueba en JavaScript sobre un motor escrito en Go, tiene ejecutores explícitos de modelo abierto y cerrado, y umbrales de aprobado o fallo adecuados para la integración continua. **Locust** describe el comportamiento de los usuarios en Python y distribuye la carga entre procesos trabajadores; sus usuarios son de bucle cerrado. Los servicios comerciales y alojados en la nube añaden generación de carga distribuida a gran escala e informes.

La herramienta importa menos que el escenario. Un buen script reproduce la combinación real de recorridos (según la analítica de producción: quizá un 70% de navegación, un 20% de cambios en el carrito y un 10% de proceso de pago), tiempos de reflexión realistas, datos realistas (usuarios, restaurantes y asientos distintos, no el mismo 10 000 veces) y el modelo de carga correcto, y comprueba que las respuestas sean correctas, porque una página de error rápida no es una solicitud con éxito.

```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  scenarios: {
    onsale: {
      executor: 'ramping-arrival-rate',   // open model: arrivals per second
      startRate: 100, timeUnit: '1s',
      preAllocatedVUs: 2000, maxVUs: 20000,
      stages: [
        { target: 4900, duration: '2m' },  // ramp to 1.5x design load
        { target: 4900, duration: '20m' }, // hold
      ],
    },
  },
  thresholds: {                           // fail the run if missed
    http_req_failed: ['rate<0.005'],
    http_req_duration: ['p(99)<500'],
  },
};

export default function () {
  const res = http.get('https://perf.tickets.example/api/events/42/seats');
  check(res, { 'status is 200': (r) => r.status === 200 });
}
```

Una prueba de modelo abierto con k6 para el mapa de asientos de la salida a la venta, que sube hasta 4900 solicitudes por segundo y da la ejecución por fallida si el p99 supera los 500 ms o si falla más del 0.5% de las solicitudes.

Entornos y datos realistas

Una prueba de rendimiento solo es tan fiable como su entorno. Una base de datos de prueba con 10 000 filas cuyos índices caben en memoria no dice nada sobre producción, con 200 millones de filas cuyos índices no caben. La lista de comprobación siguiente detecta las diferencias habituales.

Tamaño y forma de los datos: volúmenes a escala de producción y distribuciones realistas, incluidas las claves calientes (hot keys), como los restaurantes más populares o los mejores asientos, que crean una contención que una prueba uniforme nunca ve. **Los mismos bloques de construcción:** los mismos tipos de instancia, límites de contenedor, configuración de la base de datos, tamaños de los grupos de conexiones y topología de red (incluidos los saltos entre zonas). **Estado de la caché:** prueba tanto con la caché caliente como en frío, porque un despliegue o un failover la vacían (Sección 7.6). **Dependencias:** servicios de terceros sustituidos por stubs que reproduzcan su distribución real de latencia y sus límites de tasa, no

respuestas instantáneas. **Aislamiento:** ninguna prueba de otro equipo compartiendo el entorno, o los resultados son ruido. **El propio generador:** capacidad suficiente de generación de carga en las ubicaciones adecuadas; un generador que funciona al 100% de CPU se mide a sí mismo.

Algunas organizaciones prueban en producción, con cuidado, porque ninguna copia es nunca del todo igual: usuarios sintéticos y eventos de prueba que se excluyen de los informes de negocio, tráfico sombra (copiar solicitudes reales a una nueva versión y descartar sus respuestas) y subidas graduales con condiciones de cancelación automáticas. Esto necesita las mismas salvaguardas que los experimentos de caos: un radio de impacto pequeño, un interruptor de emergencia y una vigilancia estrecha de los SLO.

Presupuestos de rendimiento

Un presupuesto de rendimiento es un límite que un cambio no puede superar, acordado de antemano y comprobado automáticamente, para que el rendimiento no se erosione a base de pequeñas regresiones. Para los frontends web, los presupuestos abarcan el peso de la página, el tamaño de los scripts y las Core Web Vitals, cuyos umbrales de Google para una buena experiencia, medidos en el percentil 75 de las cargas de página, son un Largest Contentful Paint de 2.5 segundos o menos, un Interaction to Next Paint de 200 ms o menos y un Cumulative Layout Shift de 0.1 o menos.

Para los backends, un presupuesto de latencia reparte un SLO de extremo a extremo entre los saltos de un recorrido. El proceso de pago de la plataforma de reparto de comida tiene un objetivo de p99 de 800 ms; el equipo asigna 20 ms al gateway, 60 ms al servicio de pedidos, 80 ms a precios, 450 ms al pago (la mayor parte, el proveedor externo de tarjetas), 40 ms a la base de datos, 100 ms a la red y la serialización, y deja 50 ms de reserva. Ahora cada equipo conoce su propio objetivo, y un cambio que lleve precios a 150 ms se ve como un incumplimiento del presupuesto antes de convertirse en un incumplimiento del SLO. En sentido estricto, los percentiles de las partes no suman el percentil del todo, así que trata el reparto como una asignación de ingeniería y verifica por separado la cifra de extremo a extremo.

Los presupuestos se hacen cumplir ejecutando una prueba de rendimiento corta y estable en el pipeline de entrega y haciendo fallar la compilación cuando se cruza un umbral, como hacen los umbrales de k6 anteriores. Esto es una función de aptitud para el rendimiento. Las pruebas del pipeline tienen que ser más repetibles que realistas: el mismo entorno, los mismos datos, la misma carga, para que un cambio del 10% signifique una regresión real y no ruido.

EJEMPLO PRÁCTICO

Ejemplo práctico: un plan de capacidad para la salida a la venta de entradas

Este es un evento distinto y más grande que la salida a la venta de la Sección 9.3, con sus propias cifras. Una gira por estadios tiene tres noches de 60 000 asientos, 180 000 entradas en total, que salen a la venta a las 10:00 de un viernes. Los fans entran en una sala de espera virtual y se les admite en el flujo de compra a un ritmo controlado. Las pruebas de carga han medido el codo de cada instancia de la API en 520 solicitudes por segundo; el equipo planifica con 400 para mantener el p99 por debajo de 500 ms.

Previsión de la demanda	520 000 inscripciones en la preventa, de las que se espera que acuda un 75%: planificar para 400 000 fans en la sala de espera. Con una media de 2.5 entradas por pedido, 72 000 pedidos agotan la gira.
Restricción determinante	El límite acordado del proveedor de pagos es de 120 pagos por segundo. Si compra el 80% de los fans admitidos y se reserva un tercio del límite para reintentos y errores, se admiten 100 fans por segundo (6000 por minuto): 80 pagos por segundo.
Fans activos (ley de Little)	100 fans/s × 240 s en el flujo de compra = 24 000 fans activos a la vez
Carga de la API	100 fans/s × 25 solicitudes por recorrido = 2500 solicitudes/s; × 1.3 por la agrupación de las llamadas al mapa de asientos = 3250 solicitudes/s de carga de diseño
Instancias	3250 / 400 = 8.1, así que 9 instancias; para sobrevivir a la pérdida de una de tres zonas, las 9 deben caber en dos zonas: 5 por zona × 3 = 15 instancias, arrancadas antes de las 09:30
Base de datos de reservas de asientos	1.5 intentos de reserva por fan = 150 escrituras/s frente a 600 escrituras/s medidas con p99 por debajo de 50 ms: 25% de utilización, pero hay que probar la contención en los asientos más populares
Sala de espera	400 000 fans consultando cada 20 s = 20 000 solicitudes/s, servidas en el borde de la CDN y sin llegar nunca al nivel de la API
Resultado	72 000 pedidos / 80 por segundo = 900 s: entradas agotadas en unos 15 minutos, tras admitir a unos 90 000 fans; la sala de espera debe avisar enseguida a todos los demás
Plazos	Límite de pagos confirmado con 3 semanas de antelación; cuota de la nube ampliada con 1 semana de antelación; prueba de carga a 1.5x (unas 4900 solicitudes/s), prueba de picos y prueba de resistencia de 2 horas con 2 semanas de antelación en un entorno del tamaño de producción

El ritmo de admisión lo fija el recurso compartido más escaso, aquí el proveedor de pagos, no cuántos servidores puede permitirse el equipo; duplicar los servidores no vendería las entradas más rápido.

Aquí el autoescalado es una red de seguridad, no el plan: un salto de cero a 3250 solicitudes por segundo a las 10:00:00 llega más rápido de lo que tardan en arrancar las instancias.

ATENCIÓN

Trampas que hacen mentir a las pruebas de rendimiento

- Generadores de modelo cerrado para tráfico abierto, y latencia medida desde el momento de envío real en lugar del previsto (omisión coordinada).
- Informar de medias en lugar de percentiles e histogramas.
- Datos de prueba diminutos y uniformes que caben en memoria y no tienen claves calientes.
- Stubs que responden al instante cuando la dependencia real tarda 400 ms y tiene un límite de tasa.
- Contar las respuestas de error rápidas como solicitudes con éxito.
- Un generador de carga que es en sí mismo el cuello de botella.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué una prueba de carga con modelo cerrado tiende a subestimar los tiempos de respuesta de un sitio web público?

Porque sus usuarios virtuales esperan cada respuesta antes de enviar más, así que cuando el sistema se ralentiza la carga ofrecida baja, mientras que los visitantes reales siguen llegando y las colas crecen.

▼ Una herramienta de prueba muestra un p99 de 2 ms y un máximo de 8 s, casi sin muestras entre ambos. ¿Qué deberías sospechar?

Omisión coordinada: probablemente la herramienta se detuvo junto con el sistema y no registró las muchas solicitudes que se habrían retrasado durante la pausa.

▼ **En el ejemplo de la salida a la venta, ¿por qué añadir instancias de la API no permite agotar las entradas más rápido?**

Porque el límite de 120 pagos por segundo del proveedor de pagos, y no el nivel de la API, limita cuántos fans pueden comprar por segundo.

▼ **¿Qué pretende encontrar una prueba de resistencia (soak)?**

Problemas que solo aparecen con el tiempo bajo una carga constante, como fugas de memoria o de conexiones, colas que crecen y discos que se llenan.

FUENTES

- Schroeder, Wierman and Harchol-Balter: Open Versus Closed: A Cautionary Tale (NSDI 2006): https://www.usenix.org/conference/nsdi-06/open-versus-closed-cautionary-tale
- Grafana k6 documentation: Open and closed models: https://grafana.com/docs/k6/latest/using-k6/scenarios/concepts/open-vs-closed/
- web.dev: Web Vitals: https://web.dev/articles/vitals
- Google SRE book: Handling Overload: https://sre.google/sre-book/handling-overload/

SECCIÓN 9.8 DE 9.8 · INTERMEDIO · 25 MIN DE LECTURA

Ingeniería del caos, incidentes y aprendizaje sin culpas

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.5 Disponibilidad y fiabilidad
- 9.4 Patrones de resiliencia
- 9.5 SLO y presupuestos de errores

Términos anteriores que utiliza: Tiempo medio de reparación (MTTR), Presupuesto de errores, Objetivo de nivel de servicio (SLO), Runbook (manual de operaciones), Failover (conmutación por error)

LA IDEA EN BREVE

Los bomberos no esperan a un incendio real para saber si las mangueras funcionan. Hacen simulacros, provocan pequeños incendios controlados para practicar y, después de cada incendio real, se sientan juntos a averiguar qué pasó, sin buscar a quién culpar, para que el siguiente incendio vaya mejor. Los equipos de software hacen lo mismo. Rompen deliberadamente pequeñas partes de sus sistemas para comprobar que las redes de seguridad funcionan, siguen un plan claro con roles definidos cuando algo falla de verdad y, después, estudian cada incidente con honestidad para que mejore el sistema, y no solo las personas.

LO QUE NECESITAS SABER

- La ingeniería del caos, que surgió del Chaos Monkey de Netflix (descrito en 2011), experimenta con un sistema para ganar confianza en que resiste condiciones turbulentas; es una prueba controlada de una hipótesis, no una rotura aleatoria.
- Los Principios de la Ingeniería del Caos (Principles of Chaos Engineering) son: una hipótesis sobre el comportamiento en estado estable, eventos del mundo real, experimentos en producción, automatización continua y un radio de impacto (blast radius) mínimo con condiciones de cancelación.
- Los game days ensayan fallos importantes con todo el equipo y ponen a prueba a las personas, los accesos, los runbooks, las copias de seguridad y la comunicación, además del software.
- La gestión de incidentes separa los roles: un coordinador del incidente que coordina en lugar de depurar, personal de operaciones que interviene, un responsable de comunicaciones que se encarga de la página de estado y un escriba; los niveles de

gravedad se definen por el impacto en los usuarios, y la mitigación va antes que el diagnóstico.

- Las revisiones posteriores al incidente sin culpas (libro de SRE de Google; John Allspaw, 2012) buscan segundas historias y varios factores contribuyentes en lugar de una única causa raíz o un error humano.
- El MTTR calculado sobre muchos incidentes es una métrica engañosa porque las duraciones están muy sesgadas; el impacto en los SLO, los temas recurrentes y los cambios del sistema completados son mejores guías.
- Las acciones sólidas eliminan peligros o añaden salvaguardas, tienen responsables y fechas límite, y se priorizan mediante la política de presupuesto de errores; los recordatorios de tener cuidado no cambian nada.

Practicar el fallo a propósito

Todo lo visto hasta ahora en este capítulo es una promesa sobre cómo se comportará el sistema cuando algo vaya mal: los timeouts saltarán, los circuit breakers se abrirán, se descartará carga, las réplicas tomarán el relevo, las alertas llegarán a alguien que sabe qué hacer. El Capítulo 2 advirtió de que un failover que nunca se ha ejercitado suele fallar cuando por fin se necesita (Sección 2.5). Esta sección trata de convertir esas promesas en evidencia, y de cómo las organizaciones responden a los fallos que ocurren de todos modos y aprenden de ellos.

Dos prácticas van juntas. Antes de los fallos, la *ingeniería del caos* y los *game days* provocan deliberadamente fallos pequeños y controlados para encontrar debilidades y formar a las personas. Después de los fallos, la *gestión de incidentes* limita los daños y las *revisiones posteriores al incidente sin culpas* convierten cada incidente en una mejora duradera. Ambas se basan en la misma convicción: los sistemas complejos siempre están parcialmente rotos, y la resiliencia viene de descubrir cómo, pronto y de forma segura.

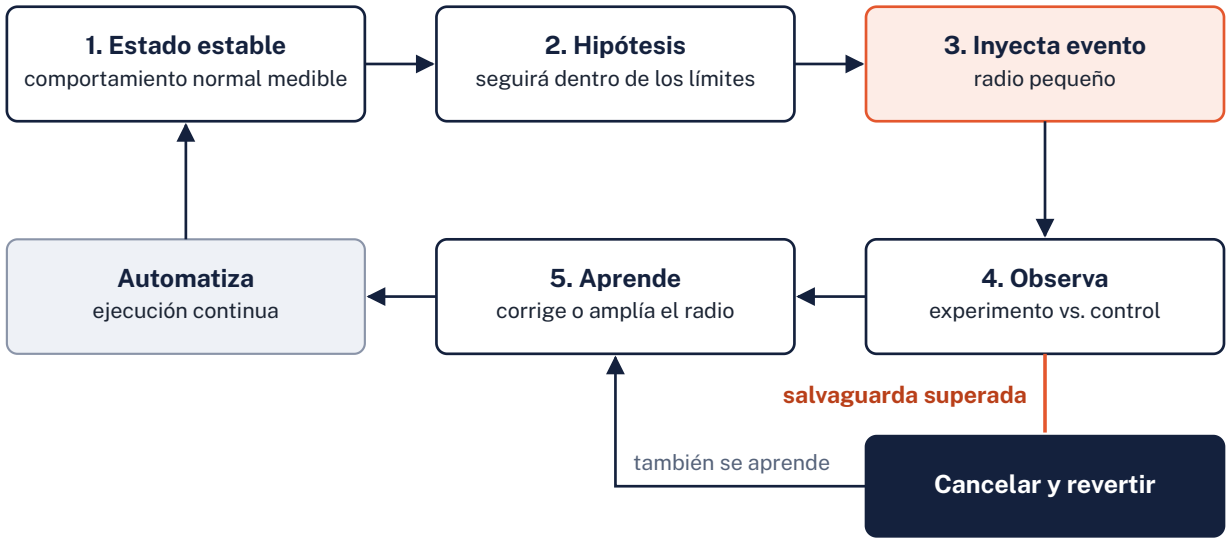
Ingeniería del caos

La ingeniería del caos nació de la migración de Netflix desde sus propios centros de datos a Amazon Web Services, que empezó tras una grave corrupción de una base de datos en 2008. Las instancias individuales en la nube podían desaparecer en cualquier momento, así que Netflix creó Chaos Monkey, una herramienta que terminaba aleatoriamente instancias de producción en horario laboral, cuando los ingenieros estaban en sus puestos para detectar y corregir cualquier problema resultante. Netflix la describió, junto con una familia de herramientas relacionadas llamada Simian Army, en una entrada de blog de julio de 2011. Herramientas posteriores fueron más allá: Chaos Kong simulaba la pérdida de una región entera de AWS evacuando su tráfico.

Ingenieros de Netflix y otros pusieron después por escrito la disciplina como los *Principios de la Ingeniería del Caos* (principlesofchaos.org, revisados por última vez en 2019), que la definen como experimentar con un sistema para ganar confianza en su capacidad de resistir condiciones turbulentas en producción. La palabra clave es *experimentar*: la ingeniería del caos no consiste en romper cosas al azar para ver qué pasa, sino en aplicar el método científico a la resiliencia. Los principios enumeran cinco prácticas:

Formula una hipótesis en torno al comportamiento en estado estable. Elige un resultado medible que indique un funcionamiento normal, como los pedidos realizados por minuto o el SLI del proceso de pago, en lugar de métricas internas como la CPU. La hipótesis de estado estable predice que ese resultado se mantendrá dentro de unos límites mientras se inyecta el fallo. **Varía los eventos del mundo real.** Inyecta los fallos que ocurren de verdad: instancias que mueren, una dependencia que se ralentiza, pérdida de paquetes, un disco lleno, un certificado caducado, desfase de relojes, un pico de tráfico. **Ejecuta los experimentos en producción.** Solo producción tiene el tráfico, los datos y la configuración reales. **Automatiza los experimentos para que se ejecuten de forma continua.** Un experimento puntual demuestra la resiliencia una vez; los sistemas cambian a diario. **Minimiza el radio de impacto (blast radius).** Empieza por el experimento más pequeño que pueda enseñar algo (una instancia, un pequeño porcentaje del tráfico) y ten preparado un interruptor para cancelarlo.

El ciclo del experimento de caos



Un experimento de caos es una prueba controlada de una hipótesis, con salvaguardas que lo detienen antes de que los usuarios sufran daños graves.

Ejecutar experimentos en producción es el principio más debatido. Sus defensores argumentan que los entornos de staging difieren en tráfico, datos, escala y configuración, así que solo los experimentos en producción revelan las debilidades reales, y que un experimento pequeño y controlado a las 15:00 es mucho más amable con los usuarios que el mismo fallo llegando sin control a las 03:00. Los críticos señalan que muchas organizaciones carecen de la observabilidad, la automatización y la confianza organizativa necesarias para hacerlo con seguridad. El consenso práctico es una progresión: empezar en un entorno de staging, después ejecutar en producción con un radio de impacto diminuto y condiciones de cancelación automáticas ligadas a los SLO, y ampliar solo a medida que crece la confianza. Nunca empieces con un sistema que ya sabes que es frágil; corrige primero las debilidades conocidas, porque un experimento cuyo resultado puedes predecir no enseña nada.

Hoy las herramientas están ampliamente disponibles: los proveedores de nube ofrecen inyección de fallos gestionada (AWS Fault Injection Service, Azure Chaos Studio), Chaos Mesh y LitmusChaos son proyectos de código abierto de la CNCF para Kubernetes, y plataformas comerciales como Gremlin ofrecen bibliotecas de ataques con controles de seguridad. Los service meshes y los proxies pueden inyectar latencia y errores en un porcentaje de las llamadas sin tocar el código de la aplicación. Las herramientas son la parte fácil; lo difícil es elegir hipótesis significativas y actuar según los resultados.

EJEMPLO PRÁCTICO

Ejemplo práctico: un experimento de latencia en el servicio de pedidos

El servicio de pedidos de la plataforma de reparto de comida llama al servicio de carta para confirmar los precios de los platos. El diseño (según la Sección 9.4) dice que la llamada tiene un timeout de 300 ms y recurre a los precios en caché como alternativa. Cada instancia del servicio de pedidos tiene 50 hilos de trabajo y recibe unos 40 pedidos por segundo, cada uno de los cuales ocupa normalmente un hilo durante 0.2 s.

Estado estable	Tasa de pedidos con éxito de al menos el 99.9%; p99 del proceso de pago por debajo de 800 ms; pedidos por minuto dentro de un 5% del mismo minuto de la semana anterior
Hipótesis	Si las llamadas al servicio de carta desde una instancia del servicio de pedidos tardan 2000 ms más, las métricas de estado estable de esa instancia se mantienen dentro de los límites, porque el timeout y la alternativa absorben el retraso
Radio de impacto (blast radius)	1 de 20 instancias (5% del tráfico de pedidos) en una región, a las 15:00 de un día laborable, 15 minutos, con las otras 19 instancias como grupo de control
Condiciones de cancelación	Tasa de éxito de la instancia del experimento por debajo del 99% durante 2 minutos, o tasa de consumo del SLO regional del proceso de pago por encima de 14.4: parar automáticamente
Uso de hilos esperado	Con el timeout de 300 ms: 40/s × (0.2 + 0.3) s = 20 hilos ocupados de 50
Qué ocurrió	La tasa de éxito cayó al 97% en 3 minutos y saltó la cancelación. Las trazas mostraron llamadas que esperaban 2.0 s: el timeout era en realidad de 3000 ms, un valor por defecto de una biblioteca que anulaba la configuración prevista
Por qué falló	40/s × (0.2 + 2.0) s = 88 hilos necesarios, pero solo hay 50: las solicitudes se encolaron y agotaron su timeout en el gateway
Acciones	Fijar el timeout en la configuración compartida de los clientes, añadir una prueba automatizada que haga fallar la compilación si alguna llamada saliente no tiene un timeout explícito, repetir el experimento y después ampliarlo a una zona entera

El experimento encontró, con un 5% del tráfico de una región y en 3 minutos, una debilidad que una ralentización real del servicio de carta a la hora de comer habría convertido en una caída regional.

El diseño era correcto; la configuración, no. Los experimentos prueban lo que está desplegado, no lo que está documentado.

Game days

Un game day (día de juego) es un ejercicio planificado en el que un equipo provoca o simula deliberadamente un fallo importante y practica la respuesta. Jesse Robbins popularizó esta práctica en Amazon, donde, según se cuenta, su cargo era Master of Disaster. Google organiza grandes ejercicios de pruebas de recuperación ante desastres (Disaster Recovery Testing, DiRT), y su libro de SRE describe la Wheel of Misfortune (rueda de la desgracia), un juego de rol en el que los ingenieros de guardia trabajan sobre un incidente pasado mientras un director de juego revela lo que habría mostrado la monitorización.

Los game days ponen a prueba algo más que el software. Comprueban si las alertas llegan a las personas adecuadas, si las personas tienen el acceso que necesitan a las 03:00, si los runbooks están al día, si el procedimiento de failover funciona de verdad, si las copias de seguridad se restauran y si el plan de comunicación llega a los clientes. Un game day típico de la plataforma de reparto de comida podría hacer failover de la base de datos de pedidos a su réplica en espera durante una tarde tranquila, con todo el turno de guardia observando, un facilitador que introduce sorpresas (“la persona que conoce el runbook está de vacaciones”) y un observador que anota cada retraso y cada confusión para la revisión posterior. Antes de una gran salida a la venta de entradas, el mismo equipo podría ensayar la pérdida del proveedor de pagos y el cambio al proveedor de respaldo en plena venta.

Gestión de incidentes

Un incidente es un evento no planificado que interrumpe un servicio, o amenaza con interrumpirlo, y necesita una respuesta coordinada. La gestión de incidentes es la práctica de llevar esa respuesta de forma que sea rápida, tranquila y bien comunicada. La mayor parte de la práctica moderna toma prestado del Incident Command System (sistema de mando de incidentes), desarrollado en California en la década de 1970 para la respuesta a los incendios forestales y adaptado al software en el capítulo sobre gestión de incidentes del libro de SRE de Google.

La idea central es la separación de roles. El coordinador del incidente (incident commander) es el dueño del incidente: fija prioridades, asigna roles, toma decisiones y mantiene la visión de conjunto, y deliberadamente *no* depura, porque un coordinador enterrado en un archivo de log no está coordinando. El responsable de operaciones y los intervinientes diagnostican y mitigan. El responsable de comunicaciones es la única voz hacia todos los que están fuera de la respuesta: los equipos de soporte, los directivos y los clientes, a través de la página de estado,

con actualizaciones a la frecuencia prometida incluso cuando no hay nada nuevo. Un escriba registra la cronología, las decisiones y las hipótesis. En una empresa pequeña una persona puede ocupar varios roles, pero los roles deberían nombrarse igualmente. Los roles se traspasan de forma explícita cuando cambian los turnos.

Roles en un incidente

Un solo canal del incidente; el coordinador no depura

Separar la coordinación, el trabajo técnico y la comunicación mantiene cada rol centrado.

Varios hábitos marcan la diferencia. Declara los incidentes pronto y sin complicaciones: es mejor desactivar una falsa alarma que descubrir al cabo de una hora que tres equipos han estado depurando el mismo problema por separado. Clasifica cada incidente con un nivel de gravedad definido por el impacto en los usuarios, no por la causa, para que todo el mundo sepa a quién llamar y con qué amplitud comunicar. Mitiga antes de diagnosticar: primero revierte el último despliegue, haz failover, desactiva el feature flag o descarta carga, y averigua el porqué después. Usa un solo canal del incidente. Y recuerda las obligaciones externas: puede que haya que notificar una violación de datos personales a la autoridad de control en un plazo de 72 horas según el RGPD (GDPR), por lo que los incidentes de seguridad siguen el proceso relacionado de la Sección 8.8.

Niveles de gravedad ilustrativos para la plataforma de reparto de comida

Nivel	Impacto de ejemplo	Respuesta	Comunicación
SEV1 (crítico)	Pedidos o pagos no disponibles para la mayoría de los usuarios de una región; sospecha de violación de datos	Avisar a la guardia de inmediato; coordinador del incidente y responsable de comunicaciones asignados al momento	Página de estado en 15 minutos; actualizaciones cada 30 minutos; directivos informados
SEV2 (grave)	Un recorrido clave muy degradado para muchos usuarios, como un p99 del proceso de pago por encima de 4 s en una ciudad; consumo rápido del SLO	Avisar al ingeniero de guardia; coordinador asignado	Página de estado si los clientes pueden notarlo; actualizaciones internas cada hora
SEV3 (menor)	Degradación parcial con una solución alternativa; una herramienta interna caída	Lo gestiona en horario laboral el equipo responsable	Canal del equipo; ticket
SEV4 (bajo)	Problema estético o un cuasi incidente sin impacto en los usuarios	Elemento del backlog	Revisión opcional si enseña algo

Revisiones posteriores al incidente sin culpas

Después del incidente llega la parte más valiosa: el aprendizaje. Una revisión posterior al incidente (a menudo llamada postmortem) reconstruye lo que ocurrió, por qué tuvo sentido para las personas implicadas en ese momento y qué va a cambiar. El capítulo sobre la cultura de los postmortems del libro de SRE de Google establece las expectativas: un registro escrito de cada incidente significativo, que cubra el impacto, la cronología, las causas contribuyentes y las acciones de seguimiento, que se revise y se comparta, y que se escriba sin culpas.

La entrada de John Allspaw de 2012 “Blameless PostMortems and a Just Culture”, escrita cuando dirigía las operaciones técnicas de Etsy, explica por qué una cultura sin culpas (blameless) es práctica y no simplemente amable. Los ingenieros que esperan un castigo dejan de dar cuentas completas de lo que vieron e hicieron, así que la organización pierde exactamente la información que necesita para evitar el siguiente incidente. Allspaw se apoya en la ciencia de la seguridad, incluido el trabajo de Sidney Dekker sobre la cultura justa, y en la idea de las *segundas historias*: la primera historia dice “un ingeniero ejecutó el comando equivocado”; la segunda pregunta por qué era tan fácil ejecutar ese comando, por qué la herramienta no avisó, por qué el procedimiento parecía razonable bajo la presión del tiempo, y

encuentra condiciones que también atraparán a la siguiente persona. Sin culpas no significa sin responsabilidad: se sigue esperando que las personas den cuentas completas, y la organización es responsable de corregir las condiciones.

Las buenas revisiones evitan el sesgo retrospectivo (el resultado no era evidente en ese momento), los contrafácticos (“debería haber”, “no logró”) y la búsqueda de una única causa raíz. El breve artículo de Richard Cook “How Complex Systems Fail” (1998) sostiene que los fallos graves en los sistemas complejos necesitan que se alineen varios defectos, así que no hay una única causa raíz, solo factores contribuyentes. Una cadena de cinco porqués que termina en “error humano” se ha detenido demasiado pronto. Anota también lo que salió bien: la pericia y la improvisación que limitaron los daños forman parte de cómo el sistema se mantiene realmente en pie, y son invisibles a menos que alguien las registre.

SI ESTO ES NUEVO PARA TI

Paso a paso: del quién al cómo

La idea más difícil de esta sección es que una buena revisión posterior al incidente mira más allá de la persona que cometió el último error, hacia las condiciones que lo hicieron fácil, porque esas condiciones también atraparán a la siguiente persona. Imagina que el timeout del ejemplo práctico no se hubiera detectado con un experimento.

- Un mediodía, el servicio de carta se ralentiza. Las llamadas del servicio de pedidos a ese servicio esperan hasta 3 segundos en lugar de los 300 ms del diseño, sus hilos se agotan y los pagos fallan en toda la región.
- La primera historia se escribe rápido: un desarrollador lanzó el cliente de la carta con un timeout de 3 segundos. Su acción se escribe sola: tener más cuidado con la configuración.
- La segunda historia pregunta por qué el error era fácil de cometer y difícil de notar. El desarrollador había escrito 300 ms, pero un valor por defecto de 3000 ms de una biblioteca lo anuló sin avisar; nada en la compilación comprobaba que hubiera timeouts explícitos; y el diseño decía 300 ms, así que los revisores no vieron nada raro.
- Estos son factores contribuyentes: varias condiciones que se alinearon. Cualquier desarrollador podría caer en la misma trampa, así que culpar a este no cambia nada.
- Como la revisión es sin culpas, el desarrollador puede contar abiertamente qué configuró y por qué, lo que lleva al equipo hasta el valor por defecto de la biblioteca.
- Las acciones, como en el ejemplo práctico, cambian el sistema: fijar el timeout en la configuración compartida de los clientes y añadir una comprobación en la compilación que falle si alguna llamada saliente no tiene un timeout explícito, cada una con un responsable y una fecha límite.
- **La lección:** el 'error humano' es donde empieza una revisión, no donde termina. Pregunta cómo tuvo sentido cada acción en su momento y corrige las condiciones; un recordatorio de tener cuidado deja la trampa puesta.

Aprender de los incidentes, y los límites del MTTR

Las organizaciones quieren, como es natural, una cifra que muestre si la respuesta a incidentes está mejorando, y el tiempo medio de recuperación (MTTR) es el candidato habitual. El Capítulo 2 usó el MTTR en A = MTBF / (MTBF + MTTR), donde tiene sentido para un único componente con fallos repetibles. Como métrica organizativa sobre muchos incidentes distintos es mucho más débil. Las duraciones de los incidentes están muy sesgadas: la mayoría son cortos y unos pocos son muy largos, así que la media está dominada por un puñado de valores atípicos. Los análisis de Courtney Nash de miles de informes públicos de incidentes en la Verica Open

Incident Database (informes VOID desde 2021) encontraron exactamente este sesgo, y un informe de Google de 2021 de Štěpán Davidovič, *Incident Metrics in SRE*, usó simulaciones para mostrar que, con un número realista de incidentes, los cambios en el MTTR son sobre todo ruido y no pueden mostrar de forma fiable si una mejora funcionó.

Mejores señales son el impacto en los SLO y en los presupuestos de errores (que ponderan los incidentes según lo que notaron los usuarios), los temas recurrentes en las revisiones (el mismo tipo de cambio de configuración causando problemas tres veces), los cuasi incidentes, la rapidez con que se detectan los incidentes, la frecuencia con que las revisiones dan lugar a cambios del sistema completados y la evidencia cualitativa, como si los ingenieros encuentran útiles las revisiones. Muchas organizaciones publican las revisiones internamente en una biblioteca con búsqueda y celebran reuniones periódicas de revisión de incidentes, para que la lección de un equipo llegue a los demás.

Acciones que cambian de verdad el sistema

Una revisión vale solo por los cambios que produce. Las acciones difieren mucho en solidez. Las más sólidas son las que eliminan el peligro (borrar el comando peligroso, hacer imposible el estado inseguro), después las salvaguardas que impiden que el error se propague (validación de la configuración, despliegues escalonados y canarios, límites de tasa a los cambios, rollback automático ante el consumo del SLO), y después una detección y una mitigación más rápidas (una nueva alerta de SLO, un runbook probado, un failover automatizado). Las más débiles son la formación, la documentación y los recordatorios. “Tener más cuidado” no es una acción; no cambia nada de las condiciones que encontrará la siguiente persona.

Cada acción necesita un responsable, una prioridad y una fecha límite, y debe vivir en el backlog normal del equipo y no en un documento olvidado. La política de presupuesto de errores de la Sección 9.5 aporta la palanca: un incidente que consumió una gran parte del presupuesto hace que sus correcciones tengan prioridad sobre las funcionalidades. Mantén la lista corta; cinco acciones completadas valen más que veinte abiertas. Cuando una acción cambia la arquitectura, como añadir un bulkhead o eliminar una dependencia síncrona, regístrala en un registro de decisiones de arquitectura que cite el incidente, y enlaza los runbooks (manuales de operaciones) correspondientes desde las alertas que saltarán la próxima vez.

Lista de comprobación: realizar una revisión posterior al incidente

Úsala para cada incidente por encima de una gravedad acordada, para cada incidente que consuma más de una parte fijada de un presupuesto de errores y para los cuasi incidentes interesantes.

- [ ] **Prográmala en pocos días laborables**
  Los recuerdos se desvanecen rápido; los logs y el historial del chat pueden caducar. Convoca a las personas implicadas, a un facilitador que no haya participado y a los propietarios de los servicios afectados.
- [ ] **Construye primero la cronología a partir de la evidencia**
  Reúne alertas, despliegues, mensajes de chat, paneles y trazas en una cronología antes de la reunión, para que el debate parta de los hechos y no de los recuerdos.
- [ ] **Expón el impacto en términos de usuarios**
  Quién se vio afectado, cuántos, durante cuánto tiempo, qué SLO y cuánto presupuesto de errores; incluye la exposición económica, de datos y regulatoria cuando proceda.
- [ ] **Establece explícitamente el marco sin culpas**
  Empieza diciendo que el objetivo es entender cómo ocurren realmente el sistema y el trabajo, y que nadie será castigado por describir lo que hizo y por qué.
- [ ] **Pregunta cómo tuvo sentido en su momento**
  Para cada decisión clave, pregunta qué sabía la persona, qué intentaba hacer y qué le mostraban las herramientas. Evita los contrafácticos como 'debería haber' y 'no logró'.
- [ ] **Busca varios factores contribuyentes**
  Condiciones técnicas, organizativas y de procedimiento a la vez; resiste la tentación de detenerte en la primera 'causa raíz' única o en el error humano.
- [ ] **Anota lo que salió bien**
  Registra las adaptaciones, la pericia y la suerte que limitaron los daños; forman parte de cómo el sistema se mantiene realmente en pie.
- [ ] **Acuerda unas pocas acciones que cambien el sistema**
  Prefiere eliminar peligros y añadir salvaguardas antes que la formación y los recordatorios; asigna a cada acción un responsable, una prioridad y una fecha límite, y haz su seguimiento en el backlog normal.

- [ ] **Publica y comparte**
  Redacta la revisión con una plantilla estándar, compártela ampliamente, añádela a una biblioteca con búsqueda y revisa las acciones abiertas en una reunión de revisión periódica.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué un experimento de caos necesita una hipótesis de estado estable en lugar de simplemente 'ver qué se rompe'?

Porque una predicción medible sobre el comportamiento normal de cara al usuario te dice si el sistema aguantó, proporciona condiciones de cancelación y convierte el ejercicio en evidencia y no en una anécdota.

▼ ¿Por qué debería el coordinador del incidente evitar depurar?

Porque su rol es la coordinación: prioridades, roles, decisiones y visión de conjunto. Un coordinador absorto en los detalles técnicos hace que nadie dirija la respuesta.

▼ ¿Qué es una 'segunda historia' en una revisión posterior al incidente?

El relato que va más allá de quién cometió un error, hacia las condiciones, las herramientas, las presiones y el diseño que hicieron que la acción pareciera razonable y que también atraparán a la siguiente persona.

▼ ¿Por qué el MTTR medio de un año de incidentes es una medida débil de la mejora?

Las duraciones de los incidentes están muy sesgadas, así que unos pocos incidentes largos dominan la media, y con un número realista de incidentes sus cambios son sobre todo ruido.

FUENTES

- Principles of Chaos Engineering: https://principlesofchaos.org/
- Netflix Technology Blog: The Netflix Simian Army (2011): https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116
- John Allspaw: Blameless PostMortems and a Just Culture (Etsy, 2012): https://www.etsy.com/codeascraft/blameless-postmortems
- Google SRE book: Postmortem Culture: Learning from Failure: https://sre.google/sre-book/postmortem-culture/
- Google SRE: Incident Metrics in SRE (Štěpán Davidovič): https://sre.google/resources/practices-and-processes/incident-metrics-in-sre/

Lecturas recomendadas

- *Release It! Design and Deploy Production-Ready Software (2nd edition)*, Michael T. Nygard (2018), Pragmatic Bookshelf. El libro que dio nombre a los patrones circuit breaker y bulkhead; sus patrones y antipatrones de estabilidad, ilustrados con caídas reales, son la base de la Sección 9.4.
- *The Art of Scalability (2nd edition)*, Martin L. Abbott and Michael T. Fisher (2015), Addison-Wesley. Presenta el cubo de escalado y relaciona las decisiones técnicas de escalado con la organización y los procesos.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (editors) (2016), O'Reilly Media. Gratuito en línea; el origen de los SLO y los presupuestos de errores tal como se practican hoy, con excelentes capítulos sobre sobrecarga, fallos en cascada y post-mortems.
  https://sre.google/sre-book/table-of-contents/
- *The Site Reliability Workbook: Practical Ways to Implement SRE*, Betsy Beyer, Niall Richard Murphy, David K. Rensin, Kent Kawahara and Stephen Thorne (editors) (2018), O'Reilly Media. El complemento práctico: cómo implementar los SLO, las alertas por tasa de consumo con varias ventanas y las guardias, con ejemplos prácticos.
  https://sre.google/workbook/table-of-contents/
- *Exponential Backoff And Jitter*, Marc Brooker (2015), AWS Architecture Blog. Un artículo breve basado en simulaciones que muestra por qué los reintentos necesitan esperas aleatorias; el origen de la fórmula de full jitter, muy utilizada.
  https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/
- *Systems Performance: Enterprise and the Cloud (2nd edition)*, Brendan Gregg (2020), Addison-Wesley. La referencia estándar sobre análisis de rendimiento, que incluye el método USE, las herramientas de observabilidad y los errores habituales en los benchmarks.
- *Chaos Engineering: System Resiliency in Practice*, Casey Rosenthal and Nora Jones (2020), O'Reilly Media. Escrito por profesionales de Netflix y de otras empresas; explica los principios de la ingeniería del caos y cómo la adoptan las organizaciones.
- *Fail at Scale: Reliability in the face of rapid change*, Ben Maurer (2015), ACM Queue, vol. 13, no. 8. Cómo gestiona Facebook la sobrecarga y las dependencias lentas con colas de retardo controlado, LIFO adaptativo y límites de concurrencia en el lado del cliente.
  https://queue.acm.org/detail.cfm?id=2839461

Glosario

Alerta multiventana y de múltiples tasas de consumo
Alerta de SLO que solo se dispara cuando la tasa de consumo (burn rate) supera un umbral tanto en una ventana larga como en una corta, lo que combina una detección rápida con un restablecimiento ágil y pocas falsas alarmas.

Anycast
Anunciar la misma dirección IP desde muchas ubicaciones, de modo que el enrutamiento de internet entregue los paquetes de cada usuario en la ubicación más cercana que la anuncia.

Autoescalado
Cambiar automáticamente el número de instancias a medida que cambia la carga, de forma reactiva a partir de una métrica, según un calendario o de forma predictiva a partir de una previsión.

Backoff exponencial
Esperar cada vez más entre reintentos sucesivos, normalmente duplicando el retraso cada vez hasta un límite, para que un servicio con dificultades tenga tiempo de recuperarse.

Balanceador de carga de capa 4
Balanceador de carga que trabaja en el nivel de transporte (TCP o UDP) y reenvía conexiones o paquetes sin leer la solicitud HTTP que contienen.

Balanceador de carga de capa 7
Balanceador de carga que entiende el protocolo de aplicación, normalmente HTTP, de modo que puede enrutar cada solicitud por ruta, cabecera o cookie, terminar TLS y reintentar.

Balanceo de carga global
Distribuir a los usuarios entre centros de datos o regiones, normalmente mediante respuestas DNS que varían según la ubicación y el estado, o mediante enrutamiento anycast.

Bulkhead (mamparo)
Dividir recursos como grupos de hilos, grupos de conexiones o instancias, para que un fallo o una sobrecarga en una parte no pueda consumir los recursos de las demás.

Cardinalidad
El número de valores distintos que puede tomar un campo; en las métricas, el número de combinaciones distintas de etiquetas, cada una de las cuales se convierte en una serie temporal independiente que hay que almacenar.

Circuit breaker (disyuntor)
Un envoltorio alrededor de las llamadas a una dependencia que, tras suficientes fallos, deja de llamarla durante un tiempo y falla rápidamente, y luego deja pasar llamadas de prueba para comprobar si se ha recuperado (Michael Nygard, 2007).

Comprobación de estado (health check)
Una prueba periódica, como una solicitud HTTP a un endpoint de estado, que un balanceador de carga o un orquestador usa para decidir si una instancia debe recibir tráfico o reiniciarse.

Consumidores competidores
Un patrón en el que varios workers leen de la misma cola y cada mensaje lo procesa solo uno de ellos, de modo que el rendimiento (throughput) aumenta al añadir workers.

Contador (counter)
Una métrica cuyo valor solo aumenta (o vuelve a cero al reiniciar), como el total de solicitudes atendidas; normalmente se observa su tasa de aumento.

Cooldown (ventana de estabilización)
Un periodo posterior a una acción de escalado durante el cual el autoescalador espera antes de volver a actuar, para que pueda medirse el efecto del último cambio.

Coordinador del incidente (incident commander)
La persona que coordina la respuesta a un incidente: fija prioridades, asigna funciones y toma decisiones, en lugar de arreglar las cosas personalmente.

Crédito de servicio
Reembolso o descuento que se debe a un cliente según un SLA cuando el proveedor no alcanza un nivel de servicio prometido.

Cuatro señales doradas
Latencia, tráfico, errores y saturación: las cuatro mediciones que el libro de SRE de Google recomienda supervisar en cualquier sistema orientado al usuario.

Cultura sin culpas (blameless)
Un enfoque en el que las personas que participaron en un incidente pueden describir lo que hicieron y por qué sin miedo a ser castigadas, para que la organización aprenda cómo funciona realmente el sistema.

Descarte de carga (load shedding)
Rechazar deliberadamente algunas solicitudes, de forma barata y temprana, cuando un sistema está sobrecargado, para que las solicitudes que acepta sigan completándose a tiempo.

Drenaje de conexiones (connection draining)
Dejar de enviar nuevas solicitudes a una instancia que se va a retirar mientras se deja terminar las solicitudes ya en curso, para que la reducción de escala y los despliegues no pierdan trabajo.

Factor contribuyente
Una de las varias condiciones que, juntas, hicieron posible o agravaron un incidente; las revisiones buscan muchos factores contribuyentes en lugar de una única causa raíz.

Flapping (oscilación)
Un autoescalador que añade y quita capacidad repetidamente porque reacciona a cambios de corta duración o a los efectos de sus propias acciones.

Game day (día de juego)
Un ejercicio planificado en el que un equipo provoca o simula deliberadamente un fallo y practica cómo detectarlo, responder y recuperarse de él.

Gauge (indicador)
Una métrica que registra un valor actual que puede subir o bajar, como la longitud de una cola, la memoria en uso o la temperatura.

Hipótesis de estado estable
En un experimento de caos, la predicción de que una señal medible de comportamiento normal, como los pedidos por minuto, se mantendrá dentro de unos límites mientras se inyecta un fallo.

Histograma
Una métrica que cuenta las observaciones en rangos de valores llamados buckets, como las duraciones de las solicitudes, para poder estimar percentiles y sumar los histogramas de muchos servidores.

Ingeniería del caos
Experimentar con un sistema, idealmente en producción y de forma controlada, inyectando fallos realistas para ganar confianza en que resiste condiciones turbulentas.

Jitter (fluctuación aleatoria)
Aleatoriedad que se añade a los retrasos de los reintentos o a las programaciones para que muchos clientes no actúen en el mismo momento y generen picos sincronizados.

Leaky bucket (cubeta con fugas)
Algoritmo de limitación de tasa o de conformación del tráfico en el que las solicitudes entran en una cubeta y salen a un ritmo constante; cuando la cubeta está llena, las solicitudes nuevas se rechazan.

Limitación de tasa (rate limiting)
Restringir cuántas solicitudes puede hacer un cliente, una clave o un inquilino (tenant) en un periodo, para proteger la capacidad y repartirla de forma justa; las solicitudes que exceden el límite suelen responderse con HTTP 429.

Logging estructurado (structured logging)
Escribir las entradas de log como registros clave-valor legibles por máquina (a menudo JSON) con nombres de campo coherentes, en lugar de frases de texto libre.

Margen de capacidad (headroom)
Capacidad sobrante que se mantiene deliberadamente por encima de la carga máxima prevista, para absorber picos, fallos y el tiempo que se tarda en añadir más capacidad.

Menos conexiones (least connections)
Algoritmo de balanceo de carga que envía cada solicitud nueva a la instancia con menos solicitudes o conexiones en curso en ese momento.

Método RED
Lista de comprobación de Tom Wilkie para monitorizar servicios basados en solicitudes: Rate (tasa), Errors (errores) y Duration (duración) de las solicitudes.

Método USE
Lista de comprobación de Brendan Gregg para analizar recursos: para cada recurso, comprueba la utilización (Utilisation), la saturación (Saturation) y los errores (Errors).

Modelo de carga abierto
Modelo de carga en el que las solicitudes nuevas llegan a un ritmo fijo con independencia de lo rápido que se respondan las anteriores, como los clientes que llegan a un sitio web.

Modelo de carga cerrado
Un modelo de carga con un número fijo de usuarios, cada uno de los cuales envía una nueva solicitud solo después de recibir respuesta a la anterior (más un tiempo de reflexión), de modo que la carga disminuye cuando el sistema se ralentiza.

Monitorización
Recopilar y vigilar mediciones y comprobaciones predefinidas para que los tipos de fallo conocidos se detecten y se notifiquen.

Muestreo de trazas
Conservar solo algunas trazas para controlar el costo: el muestreo en cabeza (head-based) decide cuando empieza una solicitud y el muestreo en cola (tail-based) decide cuando termina, por lo que puede conservar las solicitudes lentas y las fallidas.

Nivel de gravedad
Clasificación predefinida de lo grave que es un incidente, como de SEV1 a SEV4, que decide a quién se llama, con qué rapidez y con qué amplitud se comunica.

Omisión coordinada (coordinated omission)
Un error de medición, bautizado por Gil Tene, en el que un generador de carga espera las respuestas lentas antes de enviar más solicitudes y, por tanto, nunca registra los retrasos que habrían sufrido los usuarios reales.

OpenTelemetry
Proyecto de la CNCF, de código abierto e independiente de proveedores, que proporciona API, SDK, un protocolo (OTLP) y un recolector para producir y exportar trazas, métricas y logs.

Página de estado
Página pública u orientada a los clientes que informa de los incidentes en curso y del estado del servicio, actualizada por el responsable de comunicaciones del incidente.

Planificación de capacidad
Prever la demanda futura, medir cuánto puede soportar cada unidad de infraestructura y disponer de capacidad suficiente, con margen, antes de que se necesite.

Poder de dos elecciones aleatorias (power of two choices)
Método de balanceo de carga que elige dos instancias al azar y envía la solicitud a la menos cargada; Mitzenmacher demostró que equilibra casi tan bien como comprobar todas las instancias.

Política de presupuesto de errores
Una declaración escrita y acordada de lo que hará un equipo cuando su presupuesto de errores se esté gastando demasiado rápido o se haya agotado, como congelar los lanzamientos arriesgados.

Presupuesto de errores
La cantidad de falta de fiabilidad que permite un SLO: uno menos el objetivo, aplicado a la ventana. Un SLO de 99.9% sobre 1 000 000 de solicitudes permite 1000 solicitudes erróneas.

Presupuesto de reintentos
Límite de reintentos expresado como una proporción del tráfico normal (por ejemplo, el 10%), para que los reintentos ayuden con los fallos ocasionales pero no puedan multiplicar la carga durante una caída del servicio.

Presupuesto de rendimiento
Límite acordado sobre una medida de rendimiento, como el peso de la página o la latencia p99 por llamada a un servicio, que un cambio no debe superar, idealmente comprobado de forma automática.

Propagación de contexto
Pasar los identificadores de traza (y otro contexto de la solicitud) de un servicio al siguiente con cada llamada o mensaje, por ejemplo en la cabecera traceparent del W3C.

Propagación de plazos (deadline propagation)
Pasar a cada llamada descendente el tiempo que le queda a una solicitud completa, para que ningún servicio siga trabajando en una solicitud que quien la hizo ya ha abandonado.

Prueba de carga
Prueba de rendimiento que aplica la carga esperada (o el pico previsto) durante un periodo para comprobar que se cumplen los objetivos de latencia y de errores.

Prueba de estrés
Prueba de rendimiento que lleva la carga más allá del pico esperado para ver cómo falla el sistema y si se recupera.

Prueba de picos (spike test)
Prueba de rendimiento que aumenta la carga de golpe, por ejemplo diez veces en segundos, para comprobar el autoescalado, las colas y los mecanismos de protección.

Prueba de punto de ruptura
Una prueba de rendimiento que aumenta la carga de forma constante hasta que el sistema se rompe, para encontrar su capacidad máxima y su primer cuello de botella.

Prueba de resistencia (soak test)
Prueba de rendimiento larga con carga normal, ejecutada durante horas o días, para revelar problemas lentos como fugas de memoria o colas que crecen; también se llama prueba de durabilidad (endurance test).

Radio de impacto (blast radius)
Qué parte del sistema y cuántos usuarios pueden verse afectados por un fallo, un cambio o un experimento; los experimentos de caos lo mantienen deliberadamente pequeño.

Revisión posterior al incidente
Reunión estructurada e informe escrito tras un incidente que reconstruyen lo que ocurrió, por qué tuvo sentido para las personas implicadas y qué va a cambiar; también se llama postmortem.

Servicio sin estado
Instancia de servicio que no guarda datos entre solicitudes en su propia memoria ni en su disco, de modo que cualquier instancia puede atender cualquier solicitud y pueden añadirse o retirarse instancias libremente.

Sesión persistente (sticky session, afinidad de sesión)
Ajuste del balanceador de carga que envía todas las solicitudes de un cliente a la misma instancia, normalmente para que puedan encontrarse los datos de sesión guardados en la memoria de esa instancia.

Solicitud de cobertura (hedged request)
Enviar una segunda copia de una solicitud lenta a otra réplica tras un breve retraso y usar la respuesta que llegue primero, para reducir la latencia de cola.

Sonda de actividad (liveness probe)
Comprobación de salud que pregunta si un proceso está bloqueado sin posibilidad de recuperarse; si falla, el orquestador reinicia la instancia.

Sonda de preparación (readiness probe)
Comprobación de salud que pregunta si una instancia puede atender tráfico en este momento; si falla, la instancia se retira del balanceador de carga sin reiniciarla.

Span
Una operación cronometrada dentro de una traza, como una llamada HTTP o una consulta a la base de datos, con un nombre, una hora de inicio, una duración, atributos y un enlace a su span padre.

Tasa de consumo (burn rate)
La rapidez con la que se gasta un presupuesto de errores en relación con el ritmo que lo agotaría exactamente al final de la ventana; una tasa de consumo de 1 lo gasta por completo en exactamente una ventana.

Telemetría
Los datos que emite un sistema en ejecución sobre su propio comportamiento: logs, métricas, trazas, perfiles y eventos.

Timeout
El tiempo máximo que quien llama esperará a una operación antes de rendirse y tratarla como fallida; toda llamada remota necesita uno.

Token bucket (cubeta de tokens)
Algoritmo de limitación de tasa en el que se añaden tokens a una cubeta a un ritmo fijo hasta una capacidad máxima, y cada solicitud gasta un token; permite ráfagas de hasta esa capacidad y una tasa media constante.

Tormenta de reintentos
Aumento brusco de carga que se produce cuando muchos clientes, a menudo en varias capas, reintentan a la vez solicitudes fallidas, lo que mantiene sobrecargado un servicio que ya lo está.

Trazado distribuido
Registrar el recorrido y los tiempos de una solicitud a medida que atraviesa muchos servicios, para poder encontrar el paso lento o que falla.

Usuario virtual
Usuario simulado en una herramienta de pruebas de carga que ejecuta un script de solicitudes, normalmente en bucle.

Ventana móvil
Periodo de medición que avanza de forma continua, como los últimos 28 días, a diferencia de un mes natural que se reinicia el día uno.