BE SOFTWARE ARCHITECTURE WISE · CHAPTER 9

Performance, Scalability, Resilience and Operability

This chapter turns Chapter 2's performance and availability scenarios into working systems. It covers scaling up and out and the scale cube, load balancing and stateless services, queues, back pressure and rate limiting, the resilience patterns that stop failures spreading (timeouts, retries, circuit breakers, bulkheads), SLIs, SLOs and error budgets, observability with logs, metrics and traces, capacity planning and performance testing, and chaos engineering and incident learning. It is for developers and architects who know the distributed styles of Chapter 6 and must keep a system fast and running under real load.

bewiseapps.com/software-architecture-wise/resilience-and-operability/ · Version 2026-11-03.
Educational information only, not professional advice.

AT A GLANCE

You will learn to

- Choose between vertical and horizontal scaling and among the X, Y and Z axes of the scale cube, and size a fleet for a known peak with Little's law and headroom.
- Select autoscaling signals and settings that avoid lag and flapping, and explain why CPU is often the wrong metric.
- Compare layer 4 and layer 7 load balancing and the main algorithms, design health checks and connection draining, and externalise session state.
- Apply queues, bounded buffers, back pressure, load shedding and rate limiting (including the token bucket) to keep overload under control.
- Combine timeouts, deadlines, retries with backoff and jitter, retry budgets, circuit breakers, bulkheads and fallbacks to contain failures.
- Define SLIs and SLOs from the user's point of view, calculate error budgets and burn rates, and design alerts on them.
- Instrument a system with logs, metrics and traces using OpenTelemetry, and use the RED and USE methods to diagnose problems.
- Plan capacity and performance tests, run chaos experiments safely and lead blameless post-incident reviews.

Before you start

Chapter 2, especially the sections on performance and scalability and on availability and reliability; Chapter 6, especially why distribute, communication and APIs; and Chapter 7's sections on replication, partitioning and caching.

Time

About 3½ hours to read; 45 minutes for the test.

Key terms

scale cube · autoscaling · stateless service · layer 7 load balancer · power of two random choices · back pressure · load shedding · token bucket · circuit breaker · bulkhead · error budget · burn rate · distributed tracing · chaos engineering

From qualities on paper to qualities in production

Chapter 2 made performance and availability precise: latency percentiles, throughput, Little's law, utilisation and the nines, written as testable quality attribute scenarios. This chapter is about achieving those scenarios in a running system and keeping them true on the busiest day of the year. A design that meets its targets in a diagram can still fall over at 12:30 on a Friday when every office in the city orders lunch, or at 10:00 on the morning concert tickets go on sale. The difference between the two is rarely clever code. It is a set of architectural decisions about how capacity is added, how load is spread, how work waits, how failures are contained, and how the people running the system know what it is doing.

Two running examples carry through the chapter. The **food-delivery platform** has a sharp daily lunchtime and evening peak, dozens of services and third-party dependencies such as payment providers and mapping APIs. The **ticketing on-sale** is the extreme case: demand that rises from almost nothing to many times the system's capacity within seconds, for a fixed stock of seats, with the whole country watching. Between them they exercise every technique in the chapter.

Handling load: scaling, balancing and waiting

The first four sections are about load. Scaling strategies compares scaling up with scaling out, uses Abbott and Fisher's scale cube to separate cloning, splitting by function and splitting by data, and explains why autoscaling on CPU is often the wrong choice and how to size a fleet for a known peak with Little's law and headroom. Load balancing, statelessness and sessions covers what sits in front of the instances: layer 4 and layer 7 load balancers, the algorithms they use (including the power of two random choices), health checks that do not take a whole fleet out at once, connection draining, and why sticky sessions should give way to externalised session state.

Asynchrony, queues and back pressure is about what happens when work arrives faster than it can be done. Queues turn spikes into backlogs, which is useful only if the backlog is bounded and drains in an acceptable time. Back pressure, load shedding and rate limiting decide which work is accepted, which waits and which is refused politely with HTTP 429. Timeouts, retries, circuit breakers, bulkheads and graceful degradation then deals with failure between services: timeouts and deadlines, retries with backoff and jitter and the budgets that stop retry storms, circuit breakers, bulkheads, fallbacks, idempotency and hedged requests. Its worked example shows how one missing timeout lets a single slow dependency take down everything that calls it.

Running the system: objectives, observability, testing and learning

The second half turns from mechanisms to operation. SLIs, SLOs, SLAs and error budgets shows how to state reliability from the user's point of view, how an error budget turns an availability target into a quantity of allowed failure that teams can spend on change, and how burn-rate alerts page people only when that budget is genuinely at risk. Observability covers the logs, metrics and traces that let engineers ask new questions of a production system, the RED and USE methods and the four golden signals, distributed tracing with OpenTelemetry, and the cost of high-cardinality data.

Capacity planning and performance testing closes the loop between forecasts and evidence: load, stress, soak, spike and breakpoint tests, open and closed workload models, coordinated omission, and a capacity plan for a ticket on-sale. Chaos engineering, incidents and blameless

learning deals with the failures that happen anyway: experimenting deliberately in production to find weaknesses before customers do, running incidents with clear roles, and holding blameless reviews whose action items change the system rather than the people.

How the sections fit together

The sections form a chain. Scaling and load balancing add capacity; queues and back pressure decide what happens when capacity runs out anyway; resilience patterns stop a local problem becoming a global one. SLOs say how good is good enough, observability shows whether the system is meeting that standard and why not, testing and capacity planning predict when it will stop doing so, and chaos engineering and incident reviews find the gaps in all of the above. Most of the mechanisms are cheap to add at design time and expensive to retrofit, which is why they are architecture rather than operations detail: whether a service is stateless, whether a call is idempotent, whether a queue is bounded and whether a request carries a deadline are all decided long before the first incident.

The chapter assumes the distributed styles of Chapter 6 and the data architecture of Chapter 7. It links back to them for idempotency keys, delivery guarantees, replication and caching rather than repeating them, and forward to Chapter 10 for the cloud platforms, Kubernetes features and multi-region designs these systems usually run on.

The one idea to take away

Every resource is finite, so every system has a breaking point; the architect's job is to decide in advance what happens when it is reached. A well-designed system under excessive load slows down gracefully, sheds the least important work first, refuses clearly rather than timing out silently, isolates the failing part and recovers by itself when the pressure passes. A poorly designed one queues without limit, retries until it collapses and takes its neighbours with it. The difference is made of small, deliberate choices (a timeout here, a bounded queue there, a retry budget, a readiness check) and of the habit of measuring, testing and learning from the system as it actually runs.

SECTION 9.1 OF 9.8 · INTERMEDIATE · 30 MIN READ · LAST REVIEWED NOVEMBER 2026

Scaling strategies: vertical, horizontal and the scale cube

BEFORE YOU READ THIS

This section builds on:

- 0.7 Servers and the cloud
- 2.4 Performance and scalability
- 6.1 Why distribute

Terms it uses from earlier: Load balancer, Little's law, Amdahl's law, Utilisation, Read replica

THE IDEA IN BRIEF

Think of a busy café. You can make it serve more customers by buying a bigger, faster coffee machine (scaling up), or by opening more counters with more staff (scaling out). Scaling out goes further, but only works if any counter can serve any customer, so nobody's order can be kept in one barista's head. You can also split the work, with one counter for drinks and one for food, or one café per neighbourhood. And because extra staff take time to arrive, a sensible manager rosters them before the lunchtime rush rather than calling them when the queue is already out of the door.

WHAT YOU NEED TO KNOW

- Vertical scaling adds resources to one machine without changing the software, but hits a size and price ceiling and leaves one failure domain; horizontal scaling adds instances, scales further and tolerates failures, but needs stateless, divisible work.
- The scale cube (Abbott and Fisher) separates three ways to scale out: X clones identical copies, Y splits by function into services, and Z splits by data so each copy serves a slice of customers or records.
- Statelessness is the precondition for X-axis scaling: sessions, files, caches of record, scheduled jobs, locks and counters must move to shared stores or travel with the request.
- Autoscaling can be reactive, scheduled or predictive; reactive scaling lags load by minutes, so step changes need headroom, queues or capacity scheduled in advance.
- Scale on a signal tied to the resource that runs out, usually in-flight requests per instance or queue backlog per worker; CPU misleads for I/O-bound services and latency misleads when a shared dependency is slow.

- Prevent flapping by scaling out fast and in slowly, with stabilisation windows, tolerance bands and minimum and maximum counts, and size fleets for the peak with a zone lost, using Little's law.
- Scaling the application tier moves pressure to the data tier; check connections and hot rows, and reach for query tuning, caching, replicas, poolers and scaling up before partitioning.

What scaling means in practice

Section 2.4 defined scalability as the ability to handle more load by adding resources, and showed with Amdahl's law and the Universal Scalability Law why adding resources does not always add capacity. This section is about the practical choices: *which* resources to add, *where*, *when* and *how automatically*. The starting point is always the same question: what is the load parameter, and which resource runs out first? A food-delivery platform's order API at lunchtime is limited by requests per second; its dispatch service by the number of active couriers whose positions must be matched every few seconds; its database by write transactions per second and by connections. Each needs a different answer.

Scaling a system really means scaling its current bottleneck, then finding the next one. Doubling the web tier of a system whose database is 95% busy buys nothing except a larger bill and a database that is now pushed harder. So every scaling decision starts with measurement: utilisation and saturation for each resource (Brendan Gregg's USE method, covered in Observability), and a load test that finds where latency starts to climb (Capacity planning and performance testing).

Vertical scaling: a bigger machine

Vertical scaling (scaling up) means giving the existing machine more CPU cores, memory, faster disks or a faster network. Its great virtue is that nothing in the software changes. A single relational database on a larger server keeps its transactions, its joins and its simple consistency; an application keeps its in-memory caches and its local files. In the cloud, resizing is an API call followed by a restart. The largest cloud virtual machines now offer hundreds of virtual CPUs and terabytes of memory, which is far more than most business systems will ever need, and some very busy websites have publicly described serving enormous traffic from a small number of powerful database servers.

The limits are real, though. There is a largest machine, and the price per unit of capacity rises steeply near the top of the range. One machine is one failure domain: when it fails, or is restarted to be resized or patched, everything on it stops, so scaling up alone does nothing for availability. Software often cannot use a bigger machine fully: a single-threaded process, a

global lock or a hot database row limits the speed-up exactly as Amdahl's law predicts. And capacity comes in large, slow steps, so you pay for the peak around the clock. Vertical scaling is nevertheless the right first move far more often than fashion suggests, especially for databases, where scaling out is expensive in design effort.

Horizontal scaling: more machines

Horizontal scaling (scaling out) means running more instances and spreading the work between them, usually with a load balancer in front. It scales much further, in small and cheap steps, and the same redundancy that adds capacity also tolerates failure: with eight instances, losing one costs an eighth of capacity rather than all of it. Cloud platforms make adding an instance a matter of seconds or minutes, which is the basis of autoscaling.

The price is that the software must be designed for it. Any instance must be able to handle any request, which means instances must not keep anything that a later request depends on (the next section). Work must be divisible: a job that needs one global lock does not run faster on ten machines. Data must be reachable from every instance, which moves the pressure to the shared data tier. And the system becomes distributed, with the partial failures, timeouts and retries that Section 6.1 describes. Most real systems combine the two: stateless application tiers scale out, while databases scale up for as long as they reasonably can and then scale out with replicas and partitions.

Vertical and horizontal scaling compared

Concern	Vertical (scale up)	Horizontal (scale out)
Changes to the software	None	Instances must be stateless and work divisible
Upper limit	The largest machine available	Usually the shared data tier or coordination costs
Cost curve	Rises steeply for the largest sizes	Roughly linear, in small steps
Failure	One failure domain; resizing needs a restart	Losing one instance loses a fraction of capacity
Speed of change	Minutes plus a restart	Seconds to minutes, can be automatic
Consistency and transactions	Unchanged and simple	Unchanged for stateless tiers; hard for data
Typical use	Databases, legacy and stateful software, first response to growth	Web and API tiers, workers, anything stateless

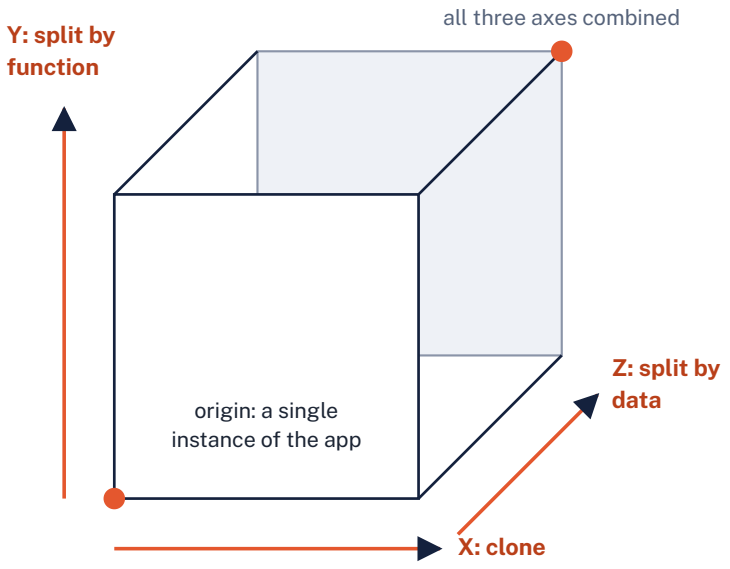
The scale cube: three different ways to scale out

Martin Abbott and Michael Fisher, who ran technology at eBay and PayPal, describe scaling out with the scale cube in *The Art of Scalability* (first edition 2009, second edition 2015). It separates three independent ways of splitting a system, each drawn as one axis of a cube whose origin is a single instance doing everything.

The **X axis** is horizontal duplication, or cloning: run N identical copies of the whole application or service and spread requests between them. It is the simplest and cheapest axis, and it works well for stateless request handling and read traffic. It does nothing for data size or write volume, because every clone still talks to the same data, and it does nothing for the size of the codebase or the number of teams working on it.

The **Y axis** splits the system by function: different services for menus, orders, dispatch and payments, each running only the code and data for its capability. Each service can then scale, fail and be deployed separately, and each is smaller and easier to understand. This is the axis behind microservices (Section 6.3), with all their costs. The **Z axis** splits by data: every copy runs the same code, but each is responsible for only a slice of the customers, tenants, regions or records, chosen by a key such as city or customer identifier. Database sharding is the Z axis applied to storage (Section 7.4); a *cell-based* architecture applies it to whole stacks, so that a fault or a bad deployment in one cell affects only that cell's customers.

The scale cube



X axis: clone
N identical copies behind a load balancer, each doing all the work: 14 order-API copies

Y axis: split by function
Separate services per capability: menus, orders, dispatch, payments

Z axis: split by data
Same code; each copy owns a slice of customers or data: one cell or shard per city

Abbott and Fisher's scale cube: each axis is a different way of splitting a system, and real systems usually combine all three.

The three axes in the food-delivery platform

Axis	What is split	Example	Solves	Costs and limits
X: clone	Requests, across identical copies	14 instances of the order API behind one load balancer	Request throughput, instance failures	Needs statelessness; all clones share one database
Y: function	Code and data, by capability	Separate menu, order, dispatch and payment services	Different scaling and failure needs; team size	Network calls, distributed data, operational overhead
Z: data	Customers or records, by key	One cell per group of cities, or orders sharded by city	Data size, write volume, blast radius	Routing layer, cross-slice queries, rebalancing, hot keys

The axes combine. The platform's order service (a Y-axis split) runs as many clones (X) in each regional cell (Z). Abbott and Fisher's practical advice is to use the cheapest axis that solves the actual problem: X for request volume, Y when parts of the system have genuinely different scaling, failure or team needs, and Z when one data set or one blast radius has become too large. Choosing Z because it sounds impressive brings routing, rebalancing and cross-partition queries long before they are needed.

Statelessness: the precondition for cloning

X-axis scaling works only if any clone can serve any request. A stateless service keeps nothing between requests in its own memory or on its own disk; everything a later request needs lives in a shared store or travels with the request. The state that typically hides in an instance is easy to list once you look: user sessions held in memory, uploaded files written to local disk, in-memory caches that are treated as the source of truth, scheduled jobs that must run on exactly one machine, in-process locks guarding a shared resource, and counters for rate limits.

Each has a standard remedy: sessions go to a shared store or a signed token (Section 9.2); files go to object storage; caches become a performance optimisation that can be lost at any time, or move to a shared cache (Section 7.6); scheduled jobs use a scheduler with leader election or a queue; locks move to the database or a coordination service; rate-limit counters move to a shared store. The twelve-factor app's rule that processes are stateless and share nothing (Section 10.3) is this same idea. Statelessness is cheap to design in and expensive to retrofit, because state leaks into code in many small places.

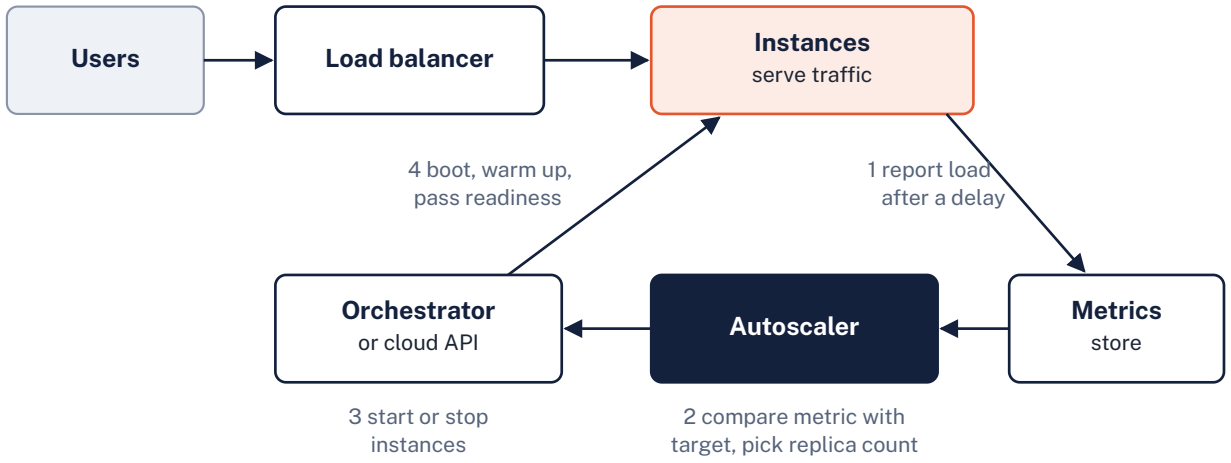
Autoscaling: reactive, scheduled and predictive

Autoscaling closes a feedback loop: measure load, compare it with a target, change the number of instances. **Reactive** autoscaling does this continuously from a metric. Target tracking is the common form: keep average CPU at 60%, or in-flight requests per instance at 30. The Kubernetes Horizontal Pod Autoscaler computes `desired = ceil(current replicas × current metric ÷ target)`: six pods averaging 45 in-flight requests against a target of 30 become ceil(6 × 45 ÷ 30) = 9. By default it re-evaluates every 15 seconds and ignores deviations within 10% of the target.

Scheduled autoscaling changes capacity at known times: raise the minimum to 18 instances at 11:15 every weekday before the lunchtime peak, lower it at 14:00. It is crude and very effective for predictable patterns, and it is the only way to have capacity in place *before* a step change such as a ticket on-sale at 10:00 sharp. **Predictive** autoscaling, offered by the major cloud providers, forecasts load from history (daily and weekly cycles) and scales ahead of it, with reactive scaling as a safety net for surprises.

The crucial property of any reactive scheme is lag. Metrics are collected and averaged over tens of seconds; the autoscaler acts on its next evaluation; a new virtual machine takes a minute or more to boot, a container image must be pulled, a JVM or Python service must start, warm its caches and connection pools and pass its readiness check before the load balancer sends it traffic. Under a cluster, the nodes themselves may have to be added first by a second, slower autoscaler. From the moment load rises to the moment new capacity is useful is commonly two to ten minutes. Reactive autoscaling therefore handles gradual change; sudden change must be absorbed by headroom, by queues (Section 9.3) or by capacity scheduled in advance.

The autoscaling feedback loop and where the lag comes from



Each step of the loop adds delay, so reactive autoscaling follows load minutes behind; headroom and scheduled capacity cover the gap.

Which metric to scale on, and why CPU is often wrong

CPU utilisation is the default signal in most platforms because it is always available. It is the right signal only when the service is CPU-bound: image resizing, encryption, heavy serialisation. Most business services are not. An order API spends most of each request waiting for the database, the payment provider and other services; its CPU may sit at 25% while every worker thread is busy waiting and new requests queue. Scaling on CPU then does nothing until it is far too late. The opposite failure also happens: a garbage-collection storm or a JIT warm-up raises CPU without any rise in load, and the autoscaler adds instances that are not needed.

A better signal is one that is directly tied to the resource that runs out. For request-response services that is usually **concurrency**: the number of requests in flight per instance, compared with the number of worker threads or connections it has. Little's law makes the link exact (in flight = arrival rate × response time), and it reacts both to more traffic and to slower responses. For queue workers it is the **backlog per worker** or the age of the oldest message, which expresses how long the queue will take to drain. Latency is tempting but dangerous as a scaling signal: it rises for reasons more instances cannot fix, such as a slow database, and scaling out a service whose dependency is struggling sends that dependency even more load. Use latency for alerting, and scale on a measure of work.

Scaling signals and when they mislead

Signal	Fits	Misleads when
CPU utilisation	CPU-bound work: image processing, encryption, heavy serialisation	Work waits on I/O, threads or connections; garbage collection or warm-up inflates CPU
Requests per second per instance	Services whose requests cost roughly the same	The request mix changes, for example more searches and fewer menu reads
In-flight requests per instance	Most request-response services; maps to thread and connection limits	A downstream dependency slows down; scaling out then adds load to it, so cap the maximum
Queue backlog per worker, or oldest message age	Asynchronous workers	Workers are limited by a downstream rate limit rather than by their number
Latency	Alerting and SLOs (Section 9.5)	The cause is a shared dependency that more instances cannot fix
Business metric (orders per minute, active sessions)	Load with a stable, measured relationship to capacity	The relationship drifts as the product changes; recalibrate with load tests

IF YOU ARE NEW TO THIS

Step by step: busy threads, idle processor

The step most people trip over here is that a server can be full while its processor looks almost idle, so CPU utilisation can be the wrong signal to scale on.

- One instance of the order-history API has 48 worker threads, so it can work on at most 48 requests at once. Each request spends most of its time waiting: for database queries and for replies from the menu service.
- A waiting thread uses almost no processor time, but it is still taken: it cannot start another request until the answer comes back.
- Each request holds a thread for about 0.14 seconds. By Little's law, 250 requests a second keep about 35 threads busy (250 × 0.14). At 450 a second, 63 would be needed, more than the 48 there are, so requests queue and response times climb.
- Yet the processor may sit at about 28%. An autoscaler told to keep CPU at 65% sees nothing wrong and adds no instances while customers wait.
- Counting in-flight requests per instance shows the problem directly: the count climbs towards 48. Scaling on that count, with a target of about 36 (three-quarters of the threads), starts adding instances while some threads are still free.
- CPU is still right for work that really is computing, such as resizing images. And the in-flight count also climbs when the database itself is struggling; then more instances only send it more load, which is why the section also sets a maximum.
- **The lesson:** scale on the resource that actually runs out. For most business services that is threads or connections, so measure the work in progress, not how hard the processor is working.

Cooldowns, stabilisation and flapping

An autoscaler that reacts to every wobble will flap: add instances when a burst arrives, remove them a minute later when the burst passes, then add them again, paying the start-up cost each time and disturbing caches and connection pools. It can also chase its own tail: new instances are cold and slow, so latency or CPU per request rises briefly after scaling out, which looks like more load.

The standard defences are asymmetric. Scale out quickly, because under-capacity hurts users; scale in slowly, because over-capacity only costs money. A cooldown or stabilisation window makes the autoscaler wait after an action, or take the highest recommendation over a recent window before scaling in: the Kubernetes autoscaler, by default, uses a five-minute (300-second) window for scaling down and none for scaling up. A tolerance band ignores small

deviations. Minimum and maximum instance counts bound the loop: the minimum keeps enough capacity to survive the loss of a zone and the first minutes of a surge, and the maximum protects the database, downstream services and the budget from an autoscaler that misreads a problem as load. Scaling in must also be graceful: an instance chosen for removal must stop receiving new requests and finish the ones in progress (connection draining, Section 9.2).

WORKED EXAMPLE

Worked example: sizing the order API for the lunchtime peak

The food-delivery platform's order API must handle weekday lunch. Over the last eight weeks the busiest minute reached 2,050 requests per second (rps) at about 12:30; marketing expects 15% growth this quarter. A load test of one instance (4 vCPU, 32 worker threads) found a mean response time of 120 ms and latency starting to climb at about 260 rps. The service runs in three availability zones and must survive the loss of one zone at peak.

Forecast peak λ	2,050 × 1.15 ≈ 2,360, rounded up to 2,400 rps
Little's law for the fleet	L = λ × W = 2,400 × 0.12 s = 288 requests in flight at peak
Capacity of one instance	32 threads ÷ 0.12 s ≈ 267 rps, which agrees with the knee the load test found (about 260 rps)
Target utilisation	70% of 32 threads = 22.4 requests in flight per instance (about 187 rps), leaving 30% headroom for bursts
Instances for the peak	288 ÷ 22.4 = 12.9, so 13 instances (2,400 ÷ 13 ≈ 185 rps each)
Surviving a zone loss	13 spread 5/4/4: losing 5 leaves 8 instances with 288 ÷ 8 = 36 in flight each, more than 32 threads: overload. Allowing up to 85% (27.2 in flight) during a zone loss needs 288 ÷ 27.2 = 10.6, so 11 in two zones: 6 per zone, 18 in total
Ramp and scale-out lag	Demand rises from 600 rps at 11:30 to 2,400 at 12:15: 40 rps per minute. Reactive scale-out takes about 5 minutes (1 minute to detect, 4 to boot and warm up), during which demand rises 200 rps; spare capacity at 70% is about 80 rps per instance, so reactive scaling alone could keep up with this gradual ramp; what it cannot provide is protection against a zone loss
Decision	Scheduled scaling raises the minimum to 18 (6 per zone) at 11:15 and lowers it to 6 (2 per zone) at 14:00; reactive scaling on in-flight requests per instance (target 22) remains as a safety net, with a maximum of 30
Database check	18 instances × a pool of 10 connections = 180; at the maximum of 30 instances, 300. The database allows 200, so the team adds a connection pooler before raising the maximum

Little's law turns a forecast rate and a measured response time into a concurrency figure that can be compared directly with thread and connection limits.

The zone-loss requirement, not the average load, sets the peak fleet size: 18 instances rather than 13. Reactive autoscaling cannot provide this protection because replacing a lost zone takes minutes.

Scaling out the application tier multiplied the load on the shared database's connections, which is the next bottleneck. Every sizing exercise should end by checking the tier below.

Scaling the data tier

Stateless application tiers are easy to scale; that is the point of making them stateless. The pressure then lands on the databases, caches and brokers they share, which is why the data tier is usually the real limit of a system. The tools were covered in Chapter 7 and are summarised here in the order professionals usually reach for them. First, reduce the work: indexes, better queries, fewer round trips per request. Second, absorb reads with caching (Section 7.6) and read replicas, accepting replication lag where the use case allows. Third, protect connections: each application instance holds a pool, so the total grows with the fleet, and a connection pooler such as PgBouncer lets thousands of client connections share a few hundred database connections (PostgreSQL's default `max_connections` is only 100). Fourth, scale up the primary, which is often cheaper than any redesign. Only then partition writes along the Z axis by a key such as city or customer (Section 7.4), or move to a distributed SQL database.

Two warnings apply. First, autoscaling the application tier can harm the data tier: every new instance opens connections and warms its cache from the database, so a large scale-out event can itself overload the database. Second, some limits are logical, not physical: a single hot row (the stock count for the last 500 tickets of a concert, or a global sequence number) serialises writes however large the server is. Those limits are removed by design, for example by splitting the stock into several counters or by reserving seats in batches, not by hardware.

DECISION GUIDE

Decision guide: how to add capacity to a component

Work through the questions in order for the component that limits the system today.

Q1 **Has the bottleneck been measured (utilisation, saturation and queueing for CPU, memory, connections, locks and downstream calls) rather than guessed?**

- **Yes** Continue with the component that saturates first.
- **No** Load-test and measure first. Scaling the wrong tier adds cost and no capacity, and can push more load onto the real bottleneck.

Q2 **Can the work itself be made cheaper (an index, a cache, fewer round trips, a smaller payload, less logging)?**

- **Yes** Do that first. It lowers cost at every scale and often removes the need to scale at all.
- **No** Continue: capacity has to be added.

Q3 **Is the component stateless, or can its state (sessions, files, in-memory locks, schedules) be moved to shared stores?**

- **Yes** Scale out on the X axis behind a load balancer, with autoscaling on a load signal and enough headroom for the scale-out lag.
- **No** Scale up for now, and plan the work to externalise the state; a stateful single instance is also a single point of failure.

Q4 **Is the limit in the data tier: one database's write rate, storage size or connection count?**

- **Yes** Use caching and read replicas for reads, a connection pooler for connections, and Z-axis partitioning (shards or cells) for writes and storage.
- **No** Continue.

Q5 **Do parts of the component have very different load, scaling or failure needs?**

- **Yes** Consider a Y-axis split so each part scales and fails separately, accepting the costs of distribution described in Chapter 6.
- **No** Stay with X-axis cloning: it is the cheapest axis to operate.

WATCH OUT

Common mistakes

- Scaling the tier that is easy to scale instead of the one that is saturated, typically adding web servers in front of a database that is already the bottleneck.
- Autoscaling I/O-bound services on CPU, so that capacity arrives only after latency has collapsed.
- Relying on reactive autoscaling for step changes such as on-sales, broadcasts or marketing pushes that arrive faster than instances can start.
- Setting no maximum instance count, so that a slow dependency is misread as load and the autoscaler multiplies connections to it.
- Sizing for the average day rather than for the peak with a zone or instance lost.
- Assuming instances are stateless without checking for in-memory sessions, local files, single-instance scheduled jobs and in-process locks.

CHECK YOUR UNDERSTANDING

▼ An order API's CPU is at 30% but p99 latency has tripled and all worker threads are busy. Why has the CPU-based autoscaler not reacted, and what signal would have?

The service is waiting on I/O, so threads are exhausted while the CPU is idle. In-flight requests per instance, compared with the thread count, would have risen with the load.

▼ Which axis of the scale cube addresses a database that has outgrown one server's write capacity?

The Z axis: partition the data by a key so that each shard or cell handles a slice of the writes.

▼ Why do autoscalers usually scale out quickly but scale in slowly?

Too little capacity hurts users immediately while too much only costs money, and scaling in too eagerly causes flapping when load returns.

▼ Why can reactive autoscaling not protect a ticket on-sale that starts at exactly 10:00?

Demand jumps within seconds, but detecting the rise, starting instances and warming them up takes minutes, so the capacity must be scheduled before the sale starts.

▼ **Try it: The busiest minute reached 1,500 rps, and 20% growth is expected. You plan 200 rps per instance, with 120 ms mean response and 32 threads. Three zones, surviving the loss of one. How many instances? Are threads sufficient?**

Peak target: 1,500 x 1.2 = 1,800 rps, so 9 busy instances. Two zones must carry it all, so 4.5 per zone, rounded up to 5, which is 15 in total; ten instances then give 2,000 rps. By Little's law, each instance has 200 x 0.12 = 24 requests in flight, within its 32 threads. Check database connections, and schedule capacity ahead of lunch, because reactive scaling lags.

SOURCES

- 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

SECTION 9.2 OF 9.8 · INTERMEDIATE · 27 MIN READ · LAST REVIEWED NOVEMBER 2026

Load balancing, statelessness and sessions

BEFORE YOU READ THIS

This section builds on:

- 0.4 How computers talk
- 7.4 Replication and partitioning
- 9.1 Scaling

Terms it uses from earlier: Consistent hashing, DNS, TCP, gRPC

THE IDEA IN BRIEF

Picture a supermarket with a member of staff directing shoppers to tills. Sending people to each till in turn is fair but naive: one shopper with a trolley for a family of six holds up a till while the next is idle. Sending each shopper to the till with the shortest queue works much better. The director also needs to know which tills are open, must stop sending people to a till that is about to close and let its queue finish, and works best if any till can serve any shopper, so nobody is told to come back to the same cashier who has their loyalty card.

WHAT YOU NEED TO KNOW

- A load balancer spreads connections or requests over healthy instances, hides instance changes behind one address and often terminates TLS; it may sit at the edge or inside clients and sidecars.
- Layer 4 balancers forward connections by address and port, fast and protocol-blind; layer 7 balancers read each HTTP request and can route, retry and rate-limit, which per-request protocols such as gRPC over HTTP/2 require.
- Round robin ignores request cost; least connections and the power of two random choices (Mitzenmacher) adapt to load, and two random choices avoid the herding that many balancers cause when all chase the least-loaded instance.
- Liveness failures restart an instance and readiness failures only remove it from rotation; keep shared dependencies out of liveness checks so a database blip cannot restart the whole fleet.
- Connection draining stops new requests to a departing instance and lets in-flight work finish, so deployments and scale-in cause no errors.
- Sticky sessions cause uneven load and lost sessions on failure; move session state to signed tokens, a shared store or the database so any instance can serve any request.

- Global load balancing steers users to regions by DNS answers, limited by caching and TTLs, or by anycast, which announces one address from many sites and lets routing choose the nearest.

What a load balancer does

A load balancer sits between clients and a pool of interchangeable instances and decides, for each connection or request, which instance receives it. Distribution is only part of the job. A load balancer also checks which instances are healthy and stops sending traffic to those that are not; it gives clients one stable address while instances come and go with deployments and autoscaling; it often terminates TLS, so certificates are managed in one place; and at layer 7 it can route, retry, rate-limit and record every request. It is therefore the component that makes horizontal scaling invisible to clients.

Load balancing appears at several levels. A managed cloud load balancer or a pair of proxies such as HAProxy, NGINX or Envoy usually stands at the edge. Between services, balancing is increasingly done on the client side: a gRPC client library or a service-mesh sidecar proxy (Section 6.3) holds the list of healthy instances from service discovery and chooses one itself, which removes a network hop and a central component. Whatever the form, the same decisions recur: what the balancer can see, which algorithm it uses, how it decides an instance is healthy, and how instances leave without dropping work.

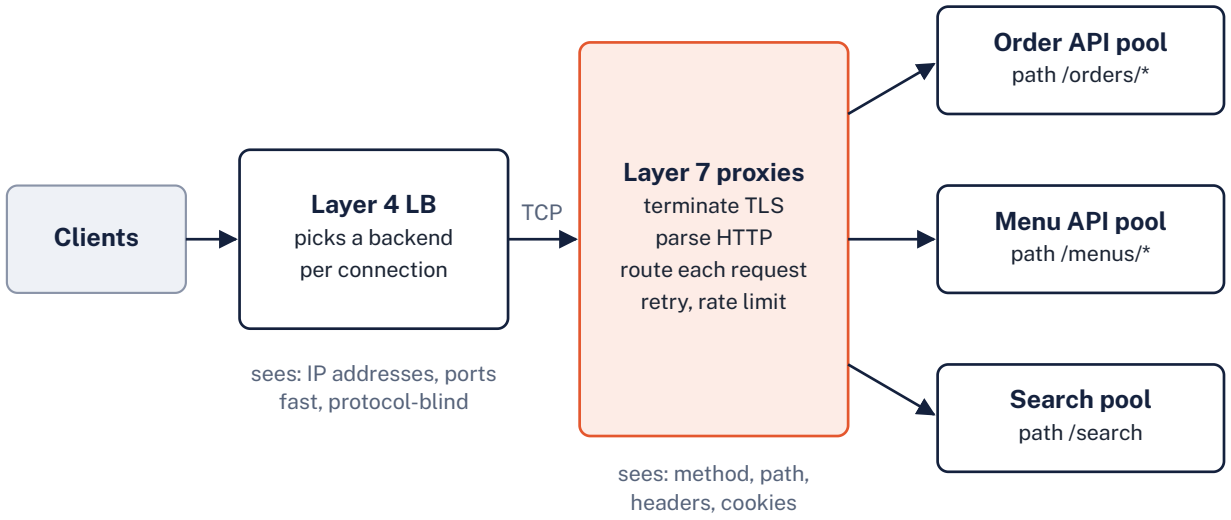
Layer 4 and layer 7 load balancing

The layer numbers come from the OSI network model. A layer 4 load balancer works at the transport layer: it sees IP addresses and TCP or UDP ports, picks a back end when a connection opens, and then forwards that connection's packets without looking inside them. Because it does so little, it is extremely fast and works for any protocol, including databases, message brokers and encrypted traffic it cannot read. Google's Maglev and the network load balancers of the major clouds are layer 4 designs that handle millions of packets per second.

A layer 7 load balancer is a reverse proxy that understands the application protocol, almost always HTTP. It terminates the client's connection (and usually TLS), reads each request, and opens or reuses its own connections to the back ends. That lets it route by path, host, header or cookie (`/orders/*` to the order pool, `/menus/*` to the menu pool), balance each request rather than each connection, retry a failed request on another instance, enforce rate limits, add tracing headers and log every request with its status and latency. The cost is more processing per request, a component that must hold the TLS keys, and one more place where timeouts and retries must be configured consistently with the rest of the system (Section 9.4).

One difference catches many teams out. HTTP/2 and gRPC multiplex many requests over one long-lived connection. A layer 4 balancer chooses a back end once per connection, so a client that opens one connection sends all its requests to one instance, however many instances exist: new instances added by autoscaling receive nothing until clients reconnect. Balancing gRPC traffic therefore needs a layer 7 proxy or client-side balancing per request.

A typical two-tier arrangement



A layer 4 tier spreads connections cheaply; a layer 7 tier reads each request and routes it by path to the right pool.

Load-balancing algorithms

The algorithm decides which healthy instance gets the next connection or request. The simple ones ignore how busy instances are; the better ones use some measure of load; hash-based ones deliberately send related requests to the same place. The table summarises the common choices.

Common load-balancing algorithms

Algorithm	How it chooses	Strengths	Weaknesses
Round robin	Each instance in turn	Simple, predictable, no state about load	Blind to request cost and to slow instances
Weighted round robin	In turn, in proportion to configured weights	Mixes instance sizes; ramps new versions up gradually (canaries)	Weights are static guesses; still blind to actual load
Random	Uniformly at random	Needs no shared state, so many balancers can use it independently	Uneven by chance, especially with few requests
Least connections (least outstanding requests)	Instance with the fewest requests in progress	Adapts to uneven request costs and slow instances	Needs accurate counts; each of many balancers sees only its own; new instances can be flooded
Latency-weighted (for example EWMA of response time)	Instance with the best recent latency, often combined with load	Steers away from degraded instances quickly	Can oscillate; a fast-failing instance looks attractive unless errors are penalised
Hash (source IP, header, consistent hashing)	Same key always maps to the same instance	Affinity for caches and sessions; consistent hashing moves few keys when the pool changes	Hot keys overload one instance; uneven key distribution
Power of two random choices	Pick two instances at random; send to the less loaded	Nearly as good as least connections, no herding, little coordination	Slightly worse than a perfect global choice with perfect information

WORKED EXAMPLE

Worked example: why least connections beats round robin when request costs differ

Two instances, A and B, each process one request at a time (real instances have more threads, but the effect is the same at larger scale). Eight requests arrive 10 ms apart. The first is a restaurant sales report costing 100 ms; the other seven are menu reads costing 10 ms each.

Round robin: assignment	A gets requests 1, 3, 5, 7; B gets 2, 4, 6, 8, regardless of cost
Round robin on A	R1 runs 0 to 100 ms. R3 (arrived at 20) waits and finishes at 110: 90 ms. R5 (40) finishes at 120: 80 ms. R7 (60) finishes at 130: 70 ms
Round robin on B	R2, R4, R6 and R8 each finish 10 ms after arriving; B is idle half the time
Round robin result	Menu reads: 90, 80, 70, 10, 10, 10, 10 ms, mean 290 ÷ 7 ≈ 41 ms, worst 90 ms. All eight requests: mean (100 + 290) ÷ 8 ≈ 49 ms
Least connections: assignment	R1 goes to A (both idle). At 10 ms A has 1 request in progress and B has 0, so R2 goes to B; B is always free again before the next arrival, so R3 to R8 all go to B
Least connections result	Every menu read takes 10 ms; the report takes 100 ms. Mean (100 + 70) ÷ 8 ≈ 21 ms, worst menu read 10 ms
Why	Round robin counts requests, not work. Least connections counts work in progress, which by Little's law (in flight = rate × time) grows on whichever instance is slower or busier, so new requests flow away from it automatically

The same effect appears whenever request costs vary widely (searches against lookups, exports against reads) or one instance is slower than its peers because of a noisy neighbour, a garbage-collection pause or a cold cache.

Least connections needs a meaningful count. With HTTP/2 or gRPC many requests share one connection, so balancers count outstanding requests instead; with long-lived WebSocket connections the count measures sessions, not load.

A newly started instance has zero connections and can be flooded before its caches are warm; many balancers offer a slow-start period that ramps its share up over a minute or so.

The power of two random choices

If least connections works so well, why not always pick the globally least-loaded instance? Because in a large system there is no single, current view of load. Dozens of load balancers or thousands of client-side balancers each see only their own requests, or a load report that is a few seconds old. If they all pick the instance that looked least loaded, they all pick the *same* one, which is promptly overwhelmed, then they all move to the next: a herd that chases the load around the pool.

Michael Mitzenmacher's doctoral thesis (1996), published as *The Power of Two Choices in Randomized Load Balancing* (IEEE Transactions on Parallel and Distributed Systems, 2001) and building on earlier work by Azar, Broder, Karlin and Upfal, analysed a strikingly simple alternative: choose two instances at random and send the request to the less loaded of the two. In the classic balls-into-bins model, placing n requests on n servers at random gives a most-loaded server with about log n ÷ log log n requests; with two random choices this falls to about log log n ÷ log 2, an exponential improvement, while a third choice adds only a little more. Randomness prevents herding, and the comparison avoids the worst instances.

The power of two random choices is now the default or a standard option in widely used proxies: Envoy's least-request balancer compares two randomly chosen hosts by default, and NGINX offers `random two least_conn`. It is a good default for client-side balancing in any system with many balancers.

Hashing and affinity

Sometimes the point is to send related requests to the *same* instance: to a cache node that already holds a restaurant's menu, or to the instance holding a WebSocket session. Hashing a key (restaurant identifier, user identifier, source IP) to an instance achieves this. Plain `hash mod N` remaps most keys whenever N changes, which empties every cache at the worst moment, during a scale-out. Consistent hashing (introduced in Section 7.4) moves only about 1 ÷ N of the keys when an instance joins or leaves. Its weakness is hot keys: the most popular restaurant at lunchtime lands on one instance. Variants such as consistent hashing with bounded loads cap each instance's share and spill the excess to the next instance on the ring.

Health checks: liveness and readiness

A load balancer must know which instances can take traffic. An active health check probes each instance periodically, for example `GET /healthz` every five seconds, marking it down after a few consecutive failures and up again after a few successes. Passive checks, often

called outlier detection, watch real traffic and eject an instance that returns, say, five consecutive server errors. The two complement each other: active checks find instances that receive no traffic, passive checks react to failures that a trivial probe does not exercise.

Kubernetes made an important distinction explicit. A liveness probe asks whether the process is stuck beyond recovery (deadlocked, out of memory in a loop); if it fails, the instance is restarted. A readiness probe asks whether the instance should receive traffic right now; if it fails, the instance is removed from the load balancer but left running, for example while it warms its caches at start-up, or while it is shedding load. A third, the startup probe, holds the other two off while a slow application starts.

The classic mistake is to check shared dependencies in these probes. If every instance's liveness probe calls the database, a brief database outage makes every probe fail, the orchestrator restarts every instance at once, and the restart storm (cold caches, thousands of new connections) prolongs the outage long after the database is back. Readiness checks that include a shared dependency can similarly mark the whole fleet unready, turning partial errors into a total outage. Keep liveness local and minimal; let readiness reflect the instance's own ability to serve; and handle dependency failures with the patterns of Section 9.4. Load balancers guard against the extreme case by failing open: an AWS Application Load Balancer routes to all targets if all of them fail their health checks, and Envoy ignores health status when the share of healthy hosts falls below a panic threshold (50% by default), on the reasoning that sending traffic to doubtful instances is better than sending it nowhere.

IF YOU ARE NEW TO THIS

Step by step: a check that restarts everything

This section's hardest idea is that the two health checks do different things, and that putting a shared dependency into the wrong one can turn a blip into an outage. A failed liveness probe gets an instance restarted; a failed readiness probe only takes it out of the load balancer's rotation.

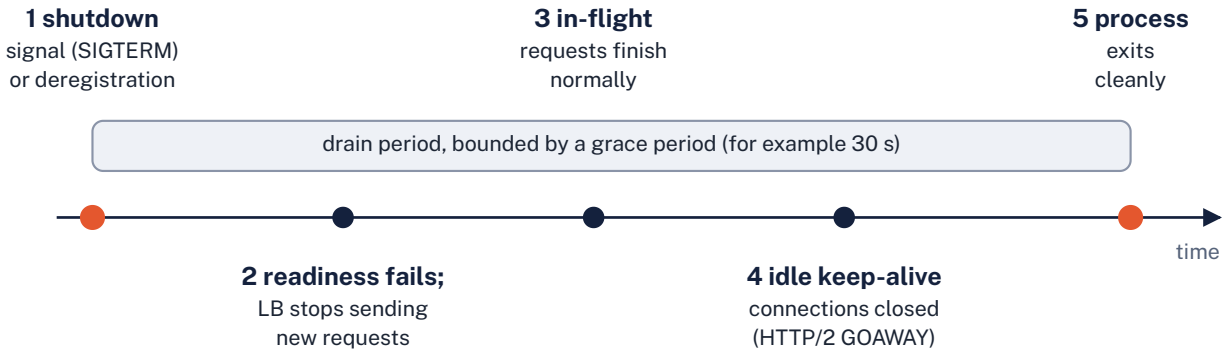
- Suppose the order API runs as 30 instances, each with a liveness probe that reports healthy only if it can reach the shared message broker.
- The broker is unreachable for about a minute while a node is replaced. Every probe fails, although the instances themselves are fine.
- After a few failures in a row, the orchestrator concludes that every instance is stuck beyond recovery and restarts all 30 together.
- The broker returns, but 30 fresh instances now reconnect to it at once and refill empty caches from the orders database. Nobody can order while they warm up, and the restart storm hits both, turning a one-minute blip into a long outage.
- Had the liveness probe checked only the instance itself, nothing would have restarted: the instances would have kept their warm caches and kept serving; only work needing the broker would have waited.
- Readiness needs care too. If every instance's readiness check also calls that broker, the whole fleet is marked unready together and the load balancer has nowhere to send anything. That is why some load balancers fail open: when every instance looks unhealthy, they send traffic to all of them anyway.
- **The lesson:** keep liveness local and minimal, let readiness reflect whether this instance can serve right now, and handle a failing shared dependency with the patterns of Section 9.4, not by restarting the fleet.

Connection draining and graceful shutdown

Instances leave the pool constantly: every deployment replaces them, autoscaling removes them, and the platform moves them during maintenance. If an instance is simply killed, every request it was processing fails, and clients see errors during what should be a routine release. Connection draining (called deregistration delay on AWS) removes the instance from the balancer's rotation so that it receives no *new* requests, waits for the requests already in progress to finish, and only then lets it stop.

The instance must cooperate. On receiving its shutdown signal it should fail its readiness check, keep serving in-flight requests, tell clients on keep-alive connections to go elsewhere (a `Connection: close` header on HTTP/1.1, a GOAWAY frame on HTTP/2), finish or hand back background work, and exit before the grace period ends. In Kubernetes, removing a pod from the load balancer and sending it SIGTERM happen concurrently, so a request can still arrive just after the signal; a short pre-stop delay of a few seconds covers that race. The default grace period is 30 seconds, after which the process is killed. Long-running requests (large uploads, streaming responses, WebSockets) need either a longer grace period or a protocol by which clients reconnect elsewhere without losing work.

Graceful shutdown in five steps



Draining turns removing an instance from a source of errors into a non-event: no new work arrives, existing work completes, then the process exits.

Sticky sessions and their problems

Older web applications kept each user's session (who is logged in, what is in the basket) in the memory of the server that created it. Behind a load balancer this works only if all of a user's requests go back to that server, so balancers offer sticky sessions (session affinity), usually by setting a cookie that names the instance. Stickiness is a quick way to put a stateful application behind a balancer, and it has a legitimate use as a performance hint, for example to keep a user on an instance whose local cache already holds their data.

As a correctness mechanism it causes trouble. Load becomes uneven, because a few heavy users or a large office behind one IP address stay pinned to one instance whatever its load. When an instance fails, its users lose their sessions: baskets vanish and people are logged out. Scaling in and every deployment do the same unless sessions are drained, which can take as long as the longest session. New instances added by autoscaling receive only new users, so they relieve the overloaded instances slowly. And the system is no longer stateless in the sense of Section 9.1, so every one of these problems reappears in failover between zones or regions.

Externalising session state

The remedy is to move session state out of the instance, so that any instance can serve any request. There are three main options, often combined. A **client-side token** carries the state with each request: a signed (and if necessary encrypted) cookie or a token such as a JWT holding the user's identity and a little data. Servers need no store at all, but tokens must stay small, cannot easily be revoked before they expire, and must be protected against tampering and theft (Section 8.3). A **shared session store**, typically Redis or Memcached, keeps sessions in a fast external store keyed by a session identifier in the cookie; it handles larger state and immediate logout, at the cost of a network call per request and a dependency that must itself be highly available. A **database** suits state that must survive anything, such as a basket the user expects to find tomorrow, which is really business data rather than session data.

A practical split for the food-delivery platform: identity in a short-lived signed token, the basket in the order service's database (users switch from phone to laptop), and transient UI state in the client. No instance keeps anything a later request needs, so the load balancer can use least-request balancing with no affinity, and instances can be added, removed and replaced at will.

Global load balancing: DNS and anycast

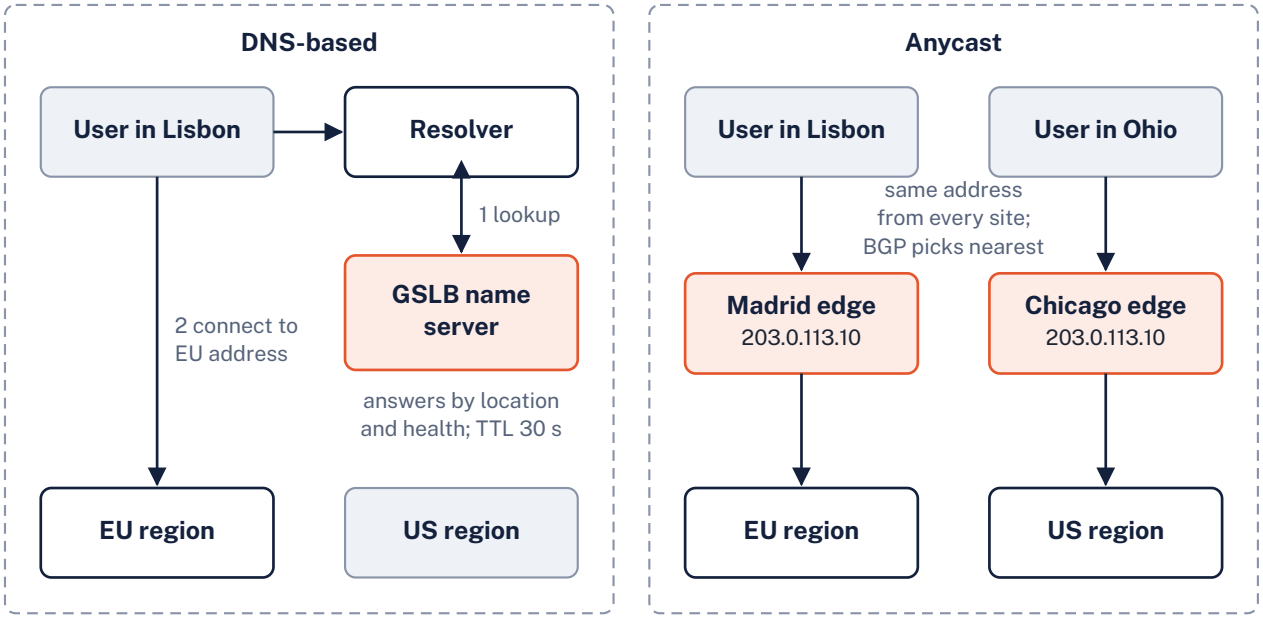
Once a system runs in several regions, something must send each user to a suitable one: the nearest healthy region, or the region that holds their data. Global load balancing is usually done in one of two ways, often combined.

DNS-based global server load balancing uses an authoritative name server that answers the same name with different addresses depending on where the query comes from, the health of each region and configured weights: a user in Lisbon is given the address of the EU region, a user in Ohio the US region. It needs no special network, but it is coarse and slow to react. Answers are cached by resolvers for at least the record's time to live (often 30 to 300 seconds) and sometimes longer, so after a region fails some users keep arriving there for minutes. The name server sees the resolver's location rather than the user's, although the EDNS Client Subnet extension (RFC 7871) lets resolvers pass on part of the client's address.

With anycast, the same IP address is announced through the internet's routing protocol (BGP) from many sites, and each user's packets are delivered to whichever site is nearest in routing terms. Failover is a matter of withdrawing the announcement from a failed site, which takes effect quickly without waiting for DNS caches. Anycast has long been used for DNS root and resolver services; CDNs and the global load balancers of the large cloud providers use it to

bring users to a nearby edge site, where layer 7 proxies terminate the connection and forward requests over the provider's network to a healthy region. Multi-region design, data placement and failover are the subject of Section 10.6.

DNS-based and anycast global load balancing



DNS steers users by answering with different addresses; anycast gives everyone the same address and lets internet routing deliver them to the nearest site.

WATCH OUT

Common mistakes

- Putting gRPC or HTTP/2 traffic behind a layer 4 balancer and wondering why new instances receive no requests.
- Using round robin for services whose requests vary widely in cost, or where some instances are slower than others.
- Calling the database from liveness probes, so that a database blip restarts the entire fleet.
- Killing instances without draining, so that every deployment and scale-in produces a burst of errors.
- Relying on sticky sessions for correctness, then losing baskets and logins whenever an instance fails or is replaced.
- Assuming DNS failover is instant; clients and resolvers can keep using an old answer for minutes.

CHECK YOUR UNDERSTANDING

▼ Why might a newly added gRPC server behind a layer 4 load balancer receive no traffic?

Layer 4 balancing chooses a back end per connection, and gRPC clients keep one long-lived HTTP/2 connection, so existing clients never send it requests until they reconnect.

▼ What is the difference in effect between failing a liveness probe and failing a readiness probe?

Failing liveness makes the orchestrator restart the instance; failing readiness only removes it from the load balancer until it passes again.

▼ Why do many independent load balancers choosing the least-loaded instance perform badly?

They act on the same stale view and all pick the same instance, overloading it in turn; picking the better of two random instances avoids this herding.

▼ Give two problems caused by keeping sessions in instance memory with sticky sessions.

Users lose their sessions when the instance fails or is replaced, and load becomes uneven because heavy users stay pinned to one instance.

▼ Try it: An order API's liveness check runs a database query. During a 30-second database blip, the orchestrator restarts every instance, and the outage lasts ten minutes. Explain why, and redesign the checks. Should readiness include the database?

Liveness failure means restart, so one shared-dependency blip restarted the whole fleet, and cold starts took far longer than the blip. Liveness should only ask whether this process is working, with no shared dependencies. Readiness removes an instance from rotation without restarting it, so it may include essentials, but if all instances fail readiness together, the service disappears completely. It may be better to keep serving and return errors or a degraded answer quickly.

SOURCES

- 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/

Asynchrony, queues and back pressure

BEFORE YOU READ THIS

This section builds on:

- 2.4 Performance and scalability
- 6.4 Event-driven
- 6.5 Communication and APIs

Terms it uses from earlier: Little's law, Idempotent, Temporal coupling, Throughput

THE IDEA IN BRIEF

A busy bakery at opening time hands out numbered tickets: customers wait in an orderly line instead of crowding the counter, and the bakers work at a steady pace. That is a queue. But a line can only be so long before the bread runs out or people give up, so a good bakery also stops handing out tickets when the line is too long, serves pre-orders first, limits each customer to a sensible number of loaves, and tells latecomers honestly to come back in ten minutes rather than letting them wait for nothing.

WHAT YOU NEED TO KNOW

- A queue between producer and consumer turns a spike into a backlog the consumers drain at their own rate, letting them be sized for the average; the cost is delay for every item behind the backlog.
- Competing consumers scale background work by adding workers; delivery is at least once, so consumers must be idempotent, and visibility timeouts and prefetch must suit processing times.
- A queue cannot absorb a sustained excess; bound every queue using Little's law (maximum length = processing rate × acceptable wait) and monitor the age of the oldest message as well as the length.
- Back pressure makes faster stages slow down; when a bounded queue is full the system must block the producer, reject new work or drop some work, and should choose deliberately.
- Load shedding rejects some work early and cheaply, by priority and by age, so that goodput stays high under overload.

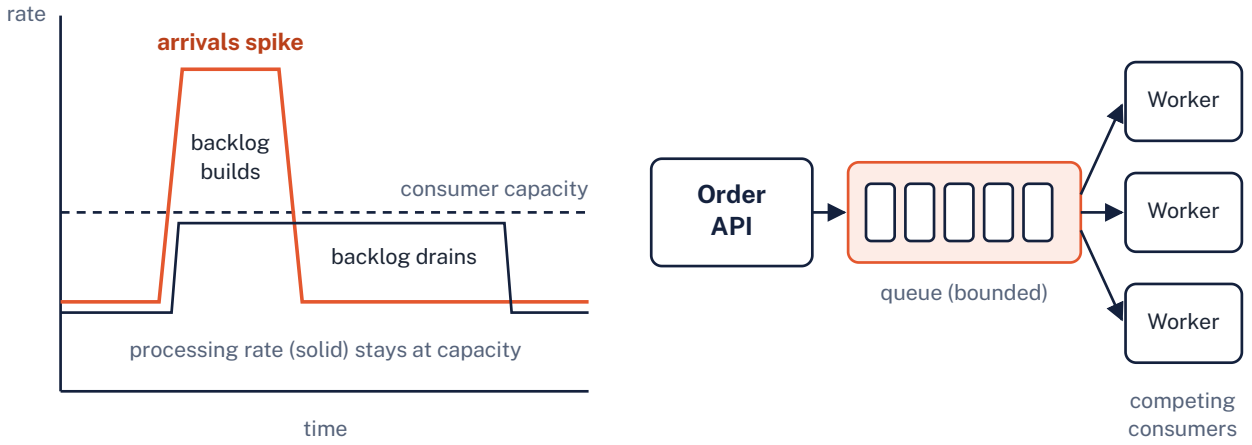
- Rate limiting shares capacity per client: the token bucket allows bursts up to its capacity and a steady refill rate, the leaky bucket smooths output, and fixed windows allow double bursts at boundaries.
- Answer limits with 429 (or 503 when the whole server is overloaded) plus Retry-After, and send repeatedly failing messages to an owned, monitored dead-letter queue.

Queues turn spikes into backlogs

Load never arrives evenly. Lunch orders bunch at 12:30, ticket buyers all press the button at 10:00:00, and a batch job dumps a million events at midnight. A synchronous service must either have capacity for the highest peak or fail during it. Putting a queue between the producer of work and the service that does it changes the arithmetic: during a spike the extra work waits in the queue, and the consumers process it at their own steady rate, catching up when the spike has passed. Microsoft's catalogue of cloud patterns calls this queue-based load levelling. Section 2.4 showed why queues form whenever arrivals bunch; this section is about using them deliberately.

The benefit is real: consumers can be sized for the average rather than the peak, a slow or restarting consumer does not lose work, and the producer is no longer temporally coupled to the consumer (Section 6.5). The price is that the work is now done later. A queue turns a spike into a *backlog*, and a backlog is simply a delay that every item behind it will experience. That is fine for sending confirmation emails or updating a search index, questionable for assigning a courier to an order, and unacceptable for a payment the customer is waiting to see confirmed. The first design question for any queue is therefore: how long may an item wait, and what happens to items that would wait longer?

A queue absorbing a spike



During the spike the queue absorbs arrivals above consumer capacity; afterwards the consumers drain the backlog at their steady rate.

Work queues and competing consumers

The most common use is a work queue with competing consumers: several identical workers read from one queue, each message goes to only one of them, and throughput grows by adding workers. It is the X axis of the scale cube applied to background work, and it scales naturally on the right signal: backlog per worker, or the age of the oldest message.

The mechanics matter for correctness. A worker takes a message, processes it and then acknowledges it; if the worker crashes first, the broker makes the message visible again after a timeout (SQS calls it the visibility timeout) and another worker processes it. Delivery is therefore at least once, so consumers must be idempotent (Section 6.4). A visibility timeout shorter than the real processing time causes the same message to be processed twice in parallel. Ordering is lost as soon as there is more than one consumer, unless messages are partitioned by key, in which case the number of partitions caps the number of useful consumers. And prefetching (a worker taking several messages at once to save round trips) should be kept low for slow tasks, or one worker hoards messages that idle workers could have processed.

A queue does not create capacity

A queue absorbs a temporary excess; it cannot absorb a permanent one. If work arrives faster than it is completed for long enough, the backlog grows without limit, and Little's law ($L = \lambda \times W$) says the waiting time grows with it. An unbounded in-memory queue eventually exhausts memory and crashes the process, losing everything in it; an unbounded durable queue fills its

disk or keeps work for hours after anyone cares about it. Worse, the queue hides the problem: producers get fast acknowledgements, error rates stay at zero, and the only symptom is that results arrive later and later. Engineers at Amazon describe such backlogs as potentially insurmountable, because a service that falls far enough behind may spend all its capacity processing work that its clients have long since given up on.

Two practices follow. First, measure the age of the oldest message as well as the queue length: age is what users experience, and it reacts immediately when consumers stall. Second, give every queue a bound derived from the waiting time you are willing to accept, using Little's law, and decide in advance what happens when the bound is reached. That decision is the subject of the next three parts of this section.

WORKED EXAMPLE

Worked example: sizing the order queue for a ticket on-sale

Tickets for a stadium concert go on sale at 10:00. Order submissions go onto a queue; workers authorise payment and allocate seats, each taking 250 ms. The forecast is 600 submissions per second for the first two minutes. The promise to customers is an answer within two minutes. There are 25,000 tickets and the average order is for 2.5 tickets.

Worker capacity with 40 workers	40 ÷ 0.25 s = 160 orders per second
Backlog after the two-minute peak	(600 − 160) × 120 s = 52,800 orders waiting
Wait for the order at the back (Little's law, W = L ÷ λ)	52,800 ÷ 160 = 330 s, five and a half minutes: the promise is broken
Longest queue that keeps the promise	L = λ × W = 160 × 120 = 19,200 orders
More workers?	The payment provider's contract allows 300 requests per second, so at most 75 workers (300 orders per second). The backlog becomes (600 − 300) × 120 = 36,000 and the wait 36,000 ÷ 300 = 120 s: the promise is met only just, with no margin
What the stock allows	25,000 ÷ 2.5 = 10,000 orders can succeed. At 600 per second that many arrive in under 17 seconds; queueing the other 62,000 or so orders only delays the inevitable 'sold out'
Decision	75 workers; a waiting room in front of the order page admits 300 shoppers per second; the order queue is bounded at 12,000 (the stock plus 20% for failed payments); once reservations cover the stock, new submissions are answered at once with 'sold out, join the waiting list' instead of joining the queue

Little's law turns a promise about waiting time into a maximum queue length. Any queue allowed to grow beyond it is a broken promise that has not been announced yet.

The downstream rate limit, not the number of workers the platform could start, set the real capacity, so adding consumers had a hard ceiling.

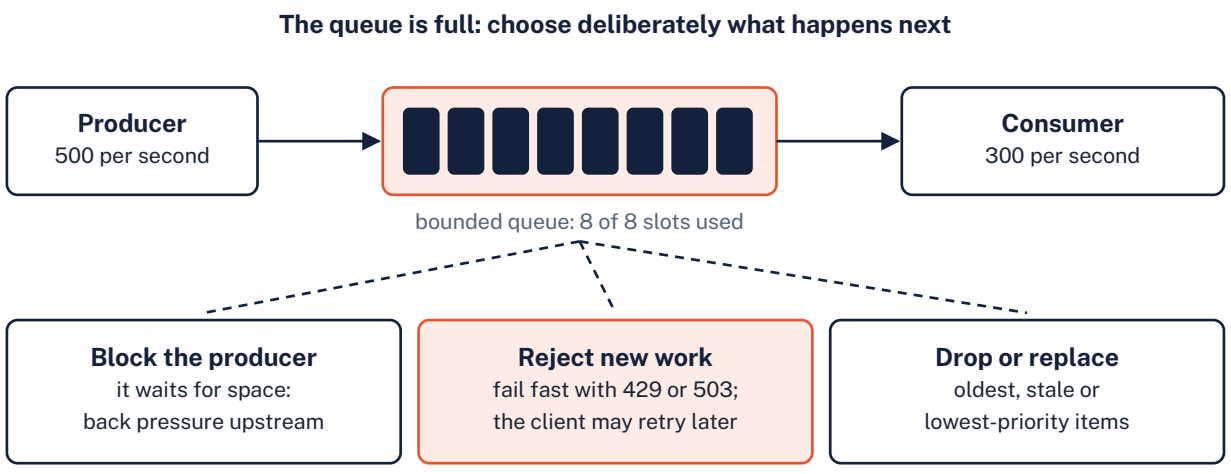
The best bound on a queue is often a business limit: there is no point holding work that cannot possibly succeed.

Back pressure: making producers slow down

Back pressure is any mechanism by which a slower downstream stage makes a faster upstream stage slow down, instead of letting work pile up between them. The term comes from fluid in pipes, and the idea was met in pipes and filters (Section 5.4). It is built into many layers of the stack you already use. TCP's flow control stops a sender when the receiver's buffer is full. A bounded blocking queue makes the producer's `put` wait for space. Pull-based consumers, as in Kafka, take work only when they are ready, so the broker's log rather than the consumer's memory holds the backlog, and a Kafka producer blocks or fails when its own send buffer is full. The Reactive Streams specification lets a subscriber request a specific number of items, so a stream never delivers more than the subscriber has asked for.

Back pressure works well inside one process or along one pipeline. Across a request-response boundary with human users it usually becomes rejection: an overloaded HTTP service cannot make a phone wait politely, so it answers `429 Too Many Requests` or `503 Service Unavailable`, ideally with a `Retry-After` header, and relies on the client to back off. When a bounded queue is full there are only three choices: block the producer, reject the new work, or drop something already accepted. Each is right somewhere; the mistake is not choosing.

Three responses to a full queue



When a bounded queue is full the system must block, reject or drop; choosing one deliberately is what keeps overload contained.

Choosing what happens when a queue is full

Strategy	Behaviour	Suits	Risk
Block the producer	Producer waits until there is space	Pipelines and batch jobs where the producer can pause	Blocked producers hold threads and connections; across services this spreads the slowdown upwards
Reject new work	Fail fast with an error such as 429 or 503	Interactive requests; the client can retry, wait or show a message	Clients that retry immediately turn rejection into more load (Section 9.4)
Drop oldest	Discard the item that has waited longest	Telemetry, location updates, live prices: newest data is most valuable	Silent loss; must be counted and visible
Drop by priority	Discard the least important items first	Mixed workloads: keep checkouts, drop analytics events	Low priorities may starve; priorities must be assigned honestly
Spill to durable storage	Overflow to disk or a durable queue	Work that must eventually happen but can wait hours	Just a larger queue: still needs a bound and a drain-time estimate

Load shedding and prioritisation

A server under more load than it can handle does worse than serving its capacity and refusing the rest. It accepts everything, every request waits in its internal queues, response times rise past the clients' timeouts, and it ends up spending its whole capacity on requests whose clients have already given up. Throughput stays high while *goodput*, the rate of requests completed in time to be useful, collapses towards zero. Load shedding prevents this by rejecting some requests deliberately so that the rest succeed.

Good shedding has three properties. It is **early and cheap**: rejecting a request must cost far less than serving it, so it happens at the front door, before authentication-heavy or database work, based on a cheap signal such as in-flight requests or queue age. It is **prioritised**: Google's SRE book describes labelling every request with a criticality (from CRITICAL_PLUS down to SHEDDABLE) so that batch and prefetch traffic is dropped before user-facing traffic. In the food-delivery platform, checkouts and courier location updates outrank recommendations, which outrank analytics events. And it is **aware of time**: a request that has already waited longer than its caller's timeout should be dropped rather than processed. Facebook's engineers

have described two refinements: a controlled-delay queue that allows only a short wait when a queue has not emptied recently, and switching to last-in, first-out under load, so that the freshest requests, whose users are still waiting, are served first.

A further step is to adapt the limit itself. Fixed concurrency limits are either too low on a good day or too high on a bad one; adaptive concurrency limiters raise the permitted number of in-flight requests while latency stays low and cut it when latency rises, borrowing the idea from TCP congestion control. At the very front of a system, admission control can take the form of a virtual waiting room, as in the ticket on-sale above: users are held in a queue at the edge and admitted at the rate the core can serve, which is load shedding presented as fairness.

IF YOU ARE NEW TO THIS

Step by step: busy but useless

The idea that needs the most care here is that an overloaded server which accepts every request can end up doing almost no useful work, although it never stops working. Refusing some requests on purpose, load shedding, is what keeps the rest useful.

- Suppose the food-delivery platform's checkout can complete 100 requests a second, and the mobile app gives up on any request not answered within 2 seconds.
- During a promotion 150 requests arrive each second. The service accepts them all, so its internal queue grows by 50 every second.
- After 4 seconds about 200 are waiting. At 100 a second, a new arrival waits about 2 seconds before work on it starts, so by then its app has already given up.
- The queue keeps growing, so from then on almost every request finishes too late. The service still completes 100 a second, so its throughput looks healthy, but its goodput, the requests completed in time to be useful, has fallen towards zero.
- Now add shedding at the front door: once the requests in progress reach a limit chosen so that accepted work finishes in time, new ones are refused at once with a 503 ('service unavailable') error, which costs far less than serving them. About 50 a second are refused; the 100 accepted are answered in time.
- Good shedding is also prioritised: a server handling several kinds of request refuses recommendations and analytics events before checkouts. It also drops any request that has already waited longer than its caller's timeout.
- **The lesson:** under overload, the real choice is not between serving everyone and serving some, but between refusing some on purpose and serving almost nobody in time. Refuse early, cheaply and by priority.

Rate limiting algorithms

Rate limiting restricts how much work a particular client, API key, tenant or IP address may submit in a period. It differs from load shedding in purpose: shedding protects the server from total overload whoever causes it, while rate limiting shares capacity fairly and protects against any single client, buggy or malicious. Both answer the same way, usually with HTTP 429. Four algorithms dominate.

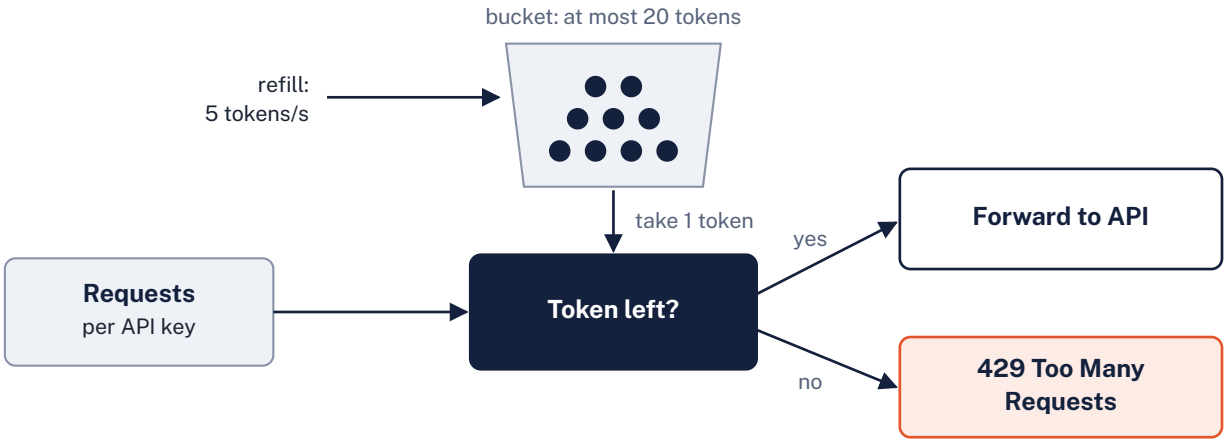
Rate-limiting algorithms compared

Algorithm	How it works	Bursts	State per key	Notes
Token bucket	Tokens are added at rate r up to capacity b; each request spends one	Allows bursts up to b, then r on average	Two numbers: tokens and last refill time	The most widely used; burst and sustained rate are tuned separately
Leaky bucket	Requests enter a bucket that empties at a constant rate; when it is full, new requests are rejected	Output is perfectly smooth; bursts wait or are refused	A queue or a counter and timestamp	Used for traffic shaping; NGINX's request limiting is based on it
Fixed window counter	Count requests per calendar window (per minute); reset at the boundary	Up to twice the limit across a boundary	One counter	Simplest; 100 requests at 12:00:59 and 100 at 12:01:00 both pass
Sliding window log	Keep the timestamp of every request in the last window; count them	Exact	One entry per request	Precise but memory-hungry at high rates
Sliding window counter	Weight the previous window's count by how much of it still overlaps, and add the current count	Close approximation	Two counters	30 s into a minute with 80 last minute and 40 now: 80 × 0.5 + 40 = 80

The token bucket deserves a closer look because it is the usual default. The bucket holds up to b tokens and gains r tokens per second. Each request takes a token; if none is left, it is rejected (or, in the shaping variant, delayed). A client that has been quiet can therefore send a burst of b requests at once, which suits real clients such as a mobile app opening a screen, while over any

longer period it cannot exceed r on average. The leaky bucket is the mirror image: it smooths output to a constant rate, which is useful for protecting a downstream system that cannot accept bursts at all.

The token bucket



Each request must take a token; the refill rate sets the sustained rate and the bucket's capacity sets the largest burst.

WORKED EXAMPLE

Worked example: a token bucket for a partner API

A restaurant-partner API key is limited by a token bucket with capacity b = 20 and refill rate r = 5 tokens per second (300 per minute on average). The bucket is full at t = 0.

t = 0: a burst of 30 requests	The first 20 take the 20 tokens and are forwarded; 10 are rejected with 429
t = 0 to 1 s	5 new tokens arrive (5 per second)
t = 1 to 10 s: a steady 8 requests per second	About 5 per second are accepted and 3 rejected; the bucket hovers near empty
t = 10 to 14 s: no requests	The bucket refills at 5 per second and is full again (20) at t = 14
Upper bound over any period T	At most b + r × T requests: in any 60 seconds, 20 + 5 × 60 = 320, so the long-run average can never exceed 5 per second
Retry-After for a rejected request	The next token arrives in 1 ÷ r = 0.2 s, but the header counts whole seconds, so the response says Retry-After: 1

Capacity sets the burst a well-behaved client may send after a quiet period; the refill rate sets the sustained rate. Choose them separately.

A client that honours Retry-After, with a little jitter, settles at the refill rate; one that retries immediately just collects more 429s.

```
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()
```

A token bucket with lazy refill: tokens are topped up for the time elapsed since the last call, so no timer is needed. In production the state lives in a shared store such as Redis, updated atomically.

HTTP 429, Retry-After and where to enforce limits

HTTP has standard signals for both situations. `429 Too Many Requests`, defined in RFC 6585 (2012), tells a client that it has sent too many requests in a given time. `503 Service Unavailable` says the server as a whole is temporarily overloaded or in maintenance. Either may carry a `Retry-After` header (RFC 9110), giving a number of seconds or a date after which retrying makes sense. Many APIs also return headers announcing the limit, the remaining allowance and the reset time, traditionally `X-RateLimit-*`; an IETF draft is standardising `RateLimit` and `RateLimit-Policy` fields. Well-behaved clients honour `Retry-After`, add jitter so that rejected clients do not all return at the same instant, and treat repeated 429s as a reason to slow down rather than to retry harder (Section 9.4).

Limits can be enforced at several layers, and mature systems use more than one: at the edge or API gateway per IP address and API key, to stop abuse and runaway clients before they cost anything; per tenant, so that one large customer cannot starve the others; and inside each service, to protect its own capacity and its dependencies. A global limit across many gateway instances needs shared state, usually atomic counters in Redis, which adds a network call per request; a common compromise is to give each of N instances a local limit of about 1 ÷ N of the total and accept some imprecision. Limits should be set from measured capacity and

published, so that partners can design to them, and the number of rejections should be monitored, because a sudden rise is either an attack, a client bug or a limit that no longer matches reality.

Dead-letter queues

Some messages fail every time: a malformed payload, an order for a restaurant that no longer exists, a bug triggered by one particular input. Retried indefinitely, such a poison message blocks a partition or wastes a worker forever. A dead-letter queue (introduced in Section 6.4) receives messages that have failed a set number of times (SQS calls this the maximum receive count in a redrive policy), so that the rest of the work can continue.

A dead-letter queue is a tool for defects, not for overload: if messages land there because consumers are too slow, the fix is capacity or back pressure, not a second queue. Every message in a DLQ means something the business expected did not happen, so the DLQ needs an owner, an alert on its depth or age, tools to inspect messages with their error and attempt count, and a way to replay them once the cause is fixed. Decide in advance what replaying late means for ordering and for messages whose business moment has passed: a refund can be replayed a day late, a courier assignment cannot.

WATCH OUT

Common mistakes

- Unbounded queues, in memory or on disk, that turn a temporary overload into a crash or into hours of stale work.
- Monitoring queue length but not the age of the oldest message, so stalled consumers go unnoticed while the length looks normal.
- Shedding load after doing the expensive part of the work, so rejection costs almost as much as success.
- Fixed-window rate limits that let clients send twice the limit across a window boundary.
- Clients that ignore 429 and Retry-After and retry immediately, turning a limit into a flood.
- Visibility timeouts shorter than processing time, so one message is processed by two workers at once.
- Using a dead-letter queue as an overflow tank instead of fixing capacity, and leaving it unowned and unmonitored.

CHECK YOUR UNDERSTANDING

▼ Workers process 200 messages per second and messages may wait at most 30 seconds. What is the largest acceptable queue length?

By Little's law, 200 × 30 = 6,000 messages; a longer queue means some messages will wait more than 30 seconds.

▼ Why is the age of the oldest message a better alarm than queue length?

Age measures the delay users experience and rises immediately if consumers stall, whereas a length that looks normal can hide a stuck queue or a slow drain.

▼ A token bucket has capacity 50 and refills at 10 per second. How many requests can a client make in the first second after a long quiet period?

About 60: the 50 tokens already in the bucket plus the 10 added during that second.

▼ What is the difference between load shedding and rate limiting?

Load shedding protects the server from total overload whoever causes it; rate limiting caps what each client may send so capacity is shared fairly.

▼ Try it: A ticket sale brings 300 submissions a second for two minutes. There are 20 workers, each taking 200 ms, and fans are promised an answer within two minutes. Does it hold? Propose a queue limit, or a worker count.

Capacity is 20 / 0.2 = 100 a second, so the backlog grows by 200 a second, reaching 24,000 at two minutes. Clearing that takes another 240 seconds, breaking the promise. A bound from Little's law is 100 x 120 = 12,000, reached after 60 seconds; beyond it, reject quickly with 429 and Retry-After. Alternatively, 60 workers give 300 a second with no backlog, if payment and database limits allow.

SOURCES

- 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

SECTION 9.4 OF 9.8 · ADVANCED · 32 MIN READ · LAST REVIEWED NOVEMBER 2026

Timeouts, retries, circuit breakers, bulkheads and graceful degradation

BEFORE YOU READ THIS

This section builds on:

- 2.4 Performance and scalability
- 6.1 Why distribute
- 6.5 Communication and APIs

Terms it uses from earlier: Partial failure, Percentile (p95, p99), Tail latency, Little's law, Idempotency key

THE IDEA IN BRIEF

Imagine phoning a supplier who never picks up. If you hold the line all day, you get nothing else done, and if every colleague does the same, the whole office stops. Sensible offices hang up after a few rings (a timeout), try again a little later but not endlessly (a retry with a pause), stop calling a supplier who is clearly closed and check again in an hour (a circuit breaker), make sure one supplier cannot tie up every phone (a bulkhead), and have a plan B such as yesterday's price list (a fallback). Software services need exactly the same habits.

WHAT YOU NEED TO KNOW

- Cascading failures happen when callers wait on a slow dependency until their own threads and connections run out; every integration point needs deliberate protection (Nygard, Release It!).
- Set explicit connect and request timeouts from measured latency percentiles, shrinking down the call chain, and propagate the remaining deadline so no service works on an abandoned request.
- Retry only transient errors on idempotent operations, a few times, with capped exponential backoff and full jitter (Brooker, 2015).
- Retries multiply across layers (three attempts at three layers gives 27 calls); retry at one layer and cap retries with a budget such as 10% of requests.
- A circuit breaker fails fast while a dependency is down, moving from closed to open on failures and through half-open trial calls back to closed.

- Bulkheads give each dependency or caller its own bounded pool, and fallbacks keep the core journey working, but fallbacks must be simple, independent of the failed part, tested and fail-closed for security.
- Idempotency makes retries, hedges and replays safe; hedged requests cut tail latency for idempotent reads at a small, budgeted cost in extra load.

Failure is normal; spreading it is optional

In a distributed system some dependency is always slow or failing: a payment provider has a bad minute, a database fails over, a network link drops packets, a deployment goes wrong. Section 6.1 showed why these partial failures cannot be prevented and why a caller often cannot even tell what happened. The architectural question is therefore not how to avoid failure but how to stop a local failure becoming a global one. A cascading failure is the typical way systems die: a slow dependency makes its callers slow, their resources fill with waiting requests, they in turn become slow for their own callers, and within seconds a problem in one minor component has taken out the whole platform.

Michael Nygard catalogued the patterns that interrupt such chains in *Release It!* (2007; second edition 2018), drawing on outages he had investigated. His central observation is that integration points, every place where one system calls another, are the main source of instability, and that each one needs deliberate protection. The patterns in this section (timeouts, deadlines, retries with backoff, circuit breakers, bulkheads, fallbacks, idempotency and hedging) are the standard toolkit. They are tactics in the sense of Section 2.5: each is a single decision aimed at availability, and each has a cost.

Timeouts: the first and most important pattern

A timeout bounds how long a caller waits. Without one, a call to a dependency that never answers holds a thread, a connection and memory for as long as the operating system keeps the connection open, which can be many minutes. Many client libraries have no timeout by default (Python's widely used `requests` library waits indefinitely unless told otherwise), so the absence of a timeout is a design decision made by omission. The worked example below shows the consequence.

Two timeouts usually matter. The **connect timeout** bounds establishing a connection and can be short, a few hundred milliseconds inside a data centre, because a healthy server accepts connections almost immediately. The **request timeout** bounds the whole exchange and should come from measurement: slightly above the dependency's high percentile latency (p99 or p99.9) under normal load, so that it cuts off pathological waits without failing ordinary slow requests. It must also fit inside the caller's own budget. If the gateway gives up after 2 seconds

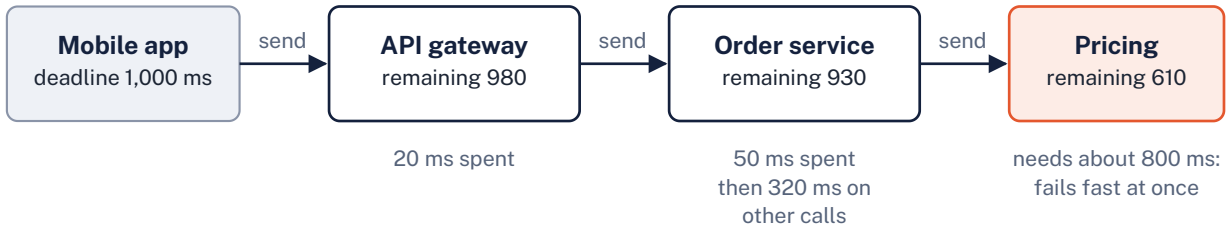
but the order service waits 10 seconds for pricing, the order service keeps working for 8 seconds on a request nobody is waiting for. Timeouts should therefore shrink as you go down a call chain, which is the idea behind deadlines.

Deadlines and their propagation

A deadline is an absolute point in time by which the whole request must be finished. Deadline propagation passes the remaining time to every downstream call, so that each service knows how much budget is left rather than applying a fixed timeout of its own. gRPC builds this in: a client sets a deadline, the remaining time travels with the call, and a server can check whether the deadline has passed and stop work. Over HTTP, teams propagate a header with the remaining milliseconds or the absolute deadline, ideally through the same middleware that propagates tracing context (Section 9.6).

Propagation gives two benefits. A service can fail fast when the time left is obviously too short for the work, rather than starting a database query that cannot finish in time. And when the user gives up, the work below stops too, instead of continuing to consume capacity during exactly the overload that caused the delay. This is load shedding of dead requests, one of the most effective protections a system under stress can have.

A deadline propagated through a call chain



Each hop sends the time left, not a fresh timeout of its own

Each hop passes on the time remaining; a service that cannot finish in the time left fails at once rather than doing work nobody will wait for.

WORKED EXAMPLE

Worked example: how one missing timeout becomes a full outage

The food-delivery order service runs 6 instances with 200 worker threads each: 1,200 threads. At lunchtime it receives 400 requests per second: 100 checkouts, which call the payment provider (normally 300 ms, so a checkout holds its thread for about 350 ms in all), and 300 other requests (order history, basket, tracking) taking 50 ms. The payment client has no timeout configured. At 12:31 the payment provider stops answering but keeps connections open.

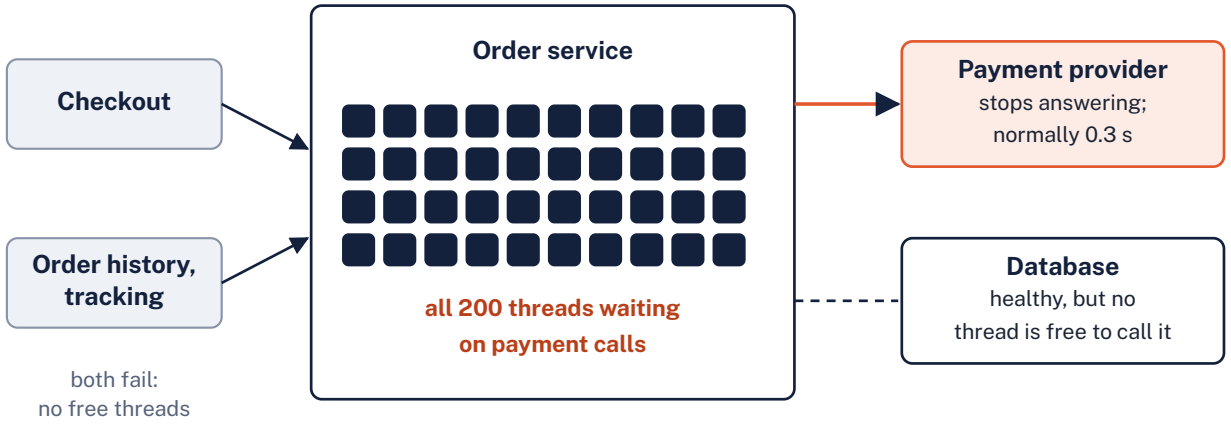
Threads busy normally (Little's law)	Checkout 100 × 0.35 s = 35; other traffic 300 × 0.05 s = 15; total 50 of 1,200 (about 4%)
After 12:31	Every checkout now holds its thread indefinitely: 100 more threads are stuck every second
Time to exhaustion	(1,200 − 15 − 35) ÷ 100 ≈ 11.5 seconds: within about 12 seconds every thread is waiting on the payment provider
Consequence	Order history, basket and tracking, which never call payments, now queue for threads and time out at the gateway. A failure in 25% of the traffic has taken out 100%
And upstream	The gateway's connections to the order service pile up in the same way; health checks time out, instances are marked unhealthy, and if the liveness probe is affected they are restarted, cold, into the same conditions
With a 2-second timeout	Checkout threads in use: 100 × 2 s = 200 across the fleet (Little's law). About 985 threads remain for other traffic, so only checkout fails, and it fails with a clear error after 2 s
Adding a circuit breaker	Once, say, half of the last 20 calls have failed, the breaker opens: checkouts fail in milliseconds and use almost no threads; a trial call every 30 s detects recovery
Adding a bulkhead	Limiting payment calls to 40 threads per instance (240 in total) guarantees at least 960 threads for everything else, whatever timeouts are set
Adding a fallback	Checkout offers the customer another payment method, or explains that card payments are delayed, instead of a generic error

The arithmetic is Little's law again: threads in use = request rate × time each request holds a thread. Without a timeout, the holding time is unbounded, so the threads needed are unbounded.

Timeouts limit how long; bulkheads limit how many; circuit breakers stop calling altogether. Each closes a different gap, which is why they are used together.

Nygard's *Release It!* describes real outages that followed exactly this pattern, often triggered by an integration point nobody considered critical.

Thread-pool exhaustion



Without a timeout, a slow payment provider holds every thread of the order service, so even requests that never touch payments fail.

Retries with exponential backoff and jitter

Many failures are transient: a dropped packet, a connection reset during a failover, a brief 503 from an overloaded instance. Retrying turns these into successes the user never sees. But retries are also load, and badly designed retries are among the most common causes of outages. Four rules make them safe. Retry only **transient** errors (timeouts, connection failures, 502, 503, 504, and 429 after the indicated delay), never errors that will recur, such as 400 or 404. Retry only **idempotent** operations or those carrying an idempotency key, because a timed-out request may have succeeded. Retry a **small number** of times, typically two or three. And **wait** between attempts, with exponential backoff: a delay that doubles with each attempt, up to a cap.

Backoff alone is not enough. If a thousand clients fail at the same moment, as they do when a dependency blips, they all retry after 100 ms, then all after 200 ms, arriving in synchronised waves that can keep the dependency down. Marc Brooker's article *Exponential Backoff And Jitter* (AWS Architecture Blog, 2015) simulated this and showed that adding randomness, jitter, greatly reduces both the total work and the time to complete. His recommended “full jitter” sleeps for a random time between zero and the exponential bound: `sleep = random(0, min(cap, base × 2^attempt))`. Jitter belongs on anything many clients do at once: retries, reconnections, cache expiry times and scheduled jobs.

```
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
```

Retrying a transient failure with capped exponential backoff and full jitter (Brooker, 2015). Only idempotent operations, or those carrying an idempotency key, should be retried.

IF YOU ARE NEW TO THIS

Step by step: why retries need randomness

The step most people trip over here is that exponential backoff alone does not spread retries out: when many callers fail at the same moment, they also wait the same times and come back together. Adding randomness, jitter, is what breaks up the crowd.

- Suppose 1,000 payment calls, each with an idempotency key so retries are safe, are in progress when the provider has a brief blip. All 1,000 fail in the same instant.
- Each caller follows plain exponential backoff: wait 100 ms, retry; if that fails, wait 200 ms, then 400 ms. Because they all failed together, all 1,000 retries arrive together, 100 ms later.
- A provider that is just recovering cannot take 1,000 calls in one instant, so many fail again, and the next wave arrives together 200 ms later. Between waves the provider is quiet; during them it is swamped, and the waves can keep it down.
- With full jitter, each caller waits a random time between zero and the backoff delay. The first retries are spread across the 100 ms: about 10 every millisecond instead of 1,000 at once.
- Callers that fail again wait a random time up to 200 ms, then up to 400 ms, so the load arrives spread out, which a recovering provider can far more easily absorb. Brooker's simulations found that this cuts both the total work and the time to finish.
- The same crowding happens with anything many machines do at once, such as reconnecting, cache entries expiring or scheduled jobs, so jitter belongs there too.
- **The lesson:** backoff makes callers wait longer each time; jitter makes sure they do not all wait the same time. Waiting politely is not enough if everyone waits in step.

Retry storms and retry budgets

Retries multiply across layers. Suppose a tap in the mobile app passes through the API gateway, the order service and the payment client, and each of the three makes up to three attempts. If the payment provider is down, one user action produces 3 × 3 × 3 = 27 calls to it, and an outage that should have meant a steady error rate becomes a retry storm of 27 times the normal load, which keeps the provider down even after the original cause is fixed. Users tapping again add a fourth layer.

Three defences work together. First, **retry at one layer only**, usually the one closest to the failing dependency, and have other layers pass the error up; Google's SRE book recommends that a server which is overloaded say so explicitly, so that callers above do not retry. Second, use a retry budget: the SRE book describes allowing retries only while they are below 10% of a

client's requests, which caps the extra load from retries at about 1.1 times normal instead of up to three times. AWS SDKs implement a similar quota of retry tokens, and gRPC's retry policy includes throttling for the same purpose. Third, combine retries with a circuit breaker, so that a dependency known to be down is not retried at all.

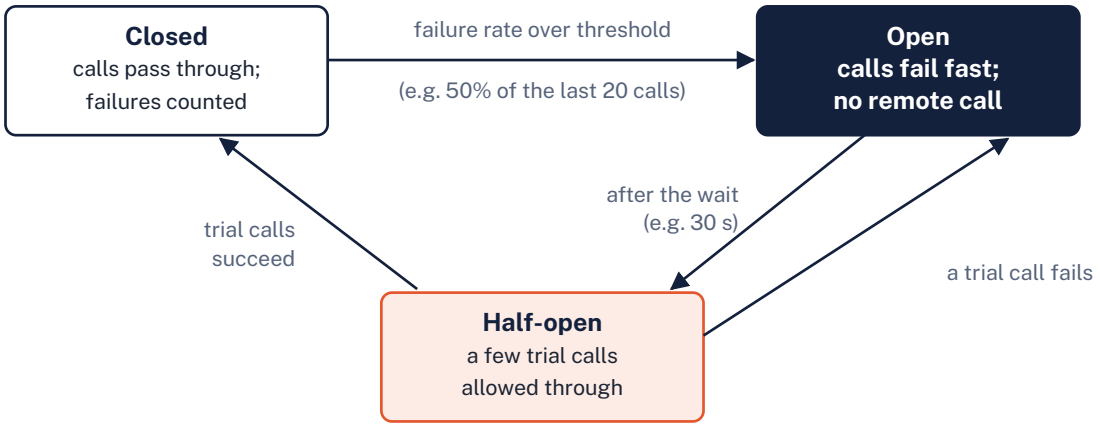
Metastable failures: when the outage outlives its cause

A retry storm is one example of a wider pattern. In a **metastable failure** (Bronson, Aghayev, Charapko and Zhu, HotOS 2021), a trigger such as a brief database blip pushes a system into a bad state that persists after the trigger has gone, because a feedback loop holds it there. Orders time out, clients retry, the database stays overloaded, and removing the blip fixes nothing. The authors treat the loop, not the trigger, as the root cause, and note that it is often built from features that improve efficiency or reliability, such as retries and caches. A later study (Huang and colleagues, OSDI 2022) examined 22 such failures across 11 organisations. Escaping usually takes strong action: shedding load (Section 9.3) or changing retry policy. So test recovery from overload, not just arrival at it.

Circuit breakers

A circuit breaker, named by Nygard after the electrical device, wraps calls to a dependency and watches their outcomes. While the dependency is healthy, the breaker is **closed** and calls pass through, while failures and timeouts are counted. When failures cross a threshold, for example half of the last 20 calls, the breaker **opens**: for a set period every call fails immediately without touching the network, which protects the caller's threads and gives the dependency room to recover. When the period expires the breaker becomes **half-open** and lets a few trial calls through. If they succeed it closes; if one fails it opens again for another period. Martin Fowler's 2014 description made the pattern widely known, and Netflix's Hystrix library (now in maintenance mode) popularised it in microservices; Resilience4j for Java and Polly for .NET are the common libraries today, and service meshes offer related protections such as concurrency limits and outlier detection (in Envoy these are called circuit breaking, although they work differently).

The circuit breaker state machine



Closed counts failures, open fails fast, and half-open tests recovery with a few trial calls.

The settings deserve thought. What counts as a failure? Timeouts and 5xx responses should; 4xx responses usually should not, because they reflect the caller's request rather than the dependency's health. Slow successes can also count, so that a dependency that answers in 5 seconds is treated as failing. The window must be large enough to avoid tripping on a couple of unlucky calls (libraries require a minimum number of calls) and the open period long enough for recovery but short enough not to prolong an outage. Breakers should be per dependency, and sometimes per endpoint, so that a failing reporting endpoint does not cut off ordering. Retries should treat an open breaker as a final answer rather than something to retry. And every change of state should be logged and graphed, because an open breaker is a symptom someone needs to see.

```
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)
```

A minimal circuit breaker. Production libraries such as Resilience4j (Java) and Polly (.NET) add failure-rate windows, slow-call detection, limits on concurrent trial calls, thread safety and metrics.

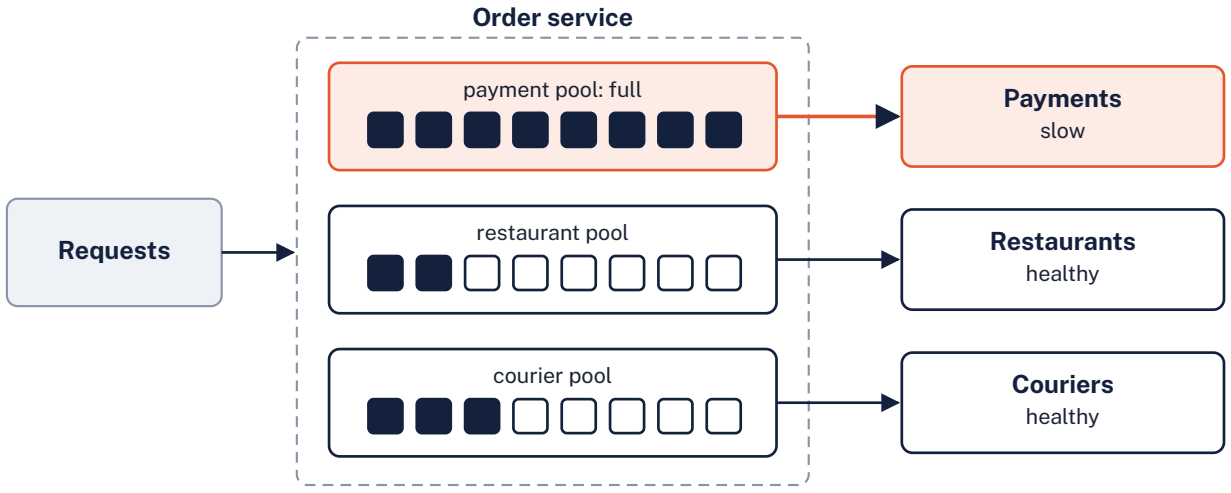
Bulkheads

A ship's hull is divided by bulkheads into watertight compartments, so that a breach floods one compartment rather than sinking the ship. In software a bulkhead partitions resources so that one failing or overloaded part cannot consume everything. The commonest form gives each dependency its own bounded thread pool, connection pool or semaphore: payments may use at most 40 concurrent calls per instance, restaurant lookups 40, courier assignment 40. When payments slow down, their pool fills and further payment calls are rejected at once, but restaurant and courier calls carry on. A timeout limits how long each call can hold resources; a bulkhead limits how many can be held at once, so it protects even when timeouts are set too long.

Bulkheads also work at larger scales. Separate deployments of the same service for different callers (one pool of order-service instances for the consumer app, another for the restaurant-partner API) stop a misbehaving partner integration from starving customers. Cell-based

architectures, the Z axis of the scale cube, make each cell a bulkhead for a slice of customers, and shuffle sharding, described by AWS engineers, assigns each customer a random small combination of instances so that one customer's poison traffic affects very few others. The cost of every bulkhead is utilisation: capacity reserved for one compartment cannot be lent to another, so pools must be sized from measured demand.

Bulkheads inside one service



With a separate pool per dependency, a slow payment provider fills only the payment pool; restaurant and courier calls continue normally.

Fallbacks and graceful degradation

When a call fails or a breaker is open, the caller must still respond. A fallback decides what it returns. The options, from best to worst for the user, are: a **cached or stale value** (the last known menu, with prices confirmed at checkout); a **sensible default** (an estimated delivery time of 30 to 45 minutes instead of a live estimate); a **reduced feature** (popular dishes instead of personalised recommendations); **deferring the work** (accepting a review and posting it when the review service recovers); and a **clear, fast error** that tells the user what to do. Designing these per feature, with the product owner, is what graceful degradation means in practice: the core journey (browse, order, pay, track) keeps working while non-essential features step aside. The same thinking guides load shedding (Section 9.3): know in advance which features to switch off first.

Security decisions are the exception: if the authorisation service is unreachable, the safe fallback is to deny (fail closed), not to allow. Practitioners also disagree about fallbacks in general. Nygard and Netflix's engineers advocate them widely. Engineers at Amazon have argued, in the Builders' Library article *Avoiding fallback in distributed systems*, that fallback paths are rarely exercised, hard to test, and can themselves trigger outages by switching a large fleet to an untested mode at the worst moment; they prefer making the primary path

more reliable and failing fast. Both positions agree on the practical rule: a fallback must be simpler than the thing it replaces, must not depend on the component that failed, and must be exercised regularly, for example in game days (Section 9.8), or it should not exist.

Static stability inside a service

Section 10.6 applied static stability to regions; the idea works at every scale. AWS separates the *control plane*, which makes changes to a system, from the *data plane*, which does the daily work, and in a statically stable design the system keeps working when a dependency is impaired. For the food-delivery platform, that means running enough order-service capacity in each zone to survive losing one, rather than counting on autoscaling at the moment of failure. It also means keeping last-known-good routing rules and restaurant settings locally, so that a configuration service outage stops changes, not orders. Spare capacity costs money, so reserve this for the core journey. For every fallback, ask what must change, and what must be working, at the moment it is needed.

Cache stampedes as a resilience problem

Section 7.6 sets out the defences against stampedes: request coalescing, jittered expiry, stale-while-revalidate, early refresh and warming. The resilience lesson is that a cache with a high hit rate is load-bearing. Bronson and colleagues note that with a 90% hit rate, losing the cache multiplies database queries roughly tenfold, so the database behind the menu cache may be unable to survive without it, and the failure can become metastable. Three habits follow. Keep a limit on concurrent database reads even for cache misses, using a bulkhead, because AWS warns that falling back to a dependency during a long cache outage can overload it. Use a soft and a hard expiry, so that existing values are still served if a refresh fails. And rehearse a cold start, because new servers begin empty.

Async runtimes and thread pools fail differently

The bulkheads above counted threads, but the failure depends on the concurrency model. With one thread per request, a slow dependency exhausts the pool, which is painful but easy to see. Async runtimes such as Node.js and Python's asyncio serve many clients with few threads, so waiting is cheap, but the danger moves. One blocking or CPU-heavy callback delays everyone on that event loop: Python's documentation notes that a one-second calculation holds up all tasks by a second. And because nothing runs out, in-flight work can grow until memory or a downstream connection pool gives way. Java's virtual threads keep the blocking style, and Oracle says they give scale, not speed; they should not be pooled, so limit concurrency with semaphores instead. Whatever the runtime, set explicit limits per dependency, keep heavy work off the event loop and measure how long tasks wait to run.

Idempotency: what makes retries and hedges safe

Idempotency is the property that doing something twice has the same effect as doing it once. Every pattern in this section that repeats work depends on it: retries after a timeout, hedged requests, message redelivery after a consumer crash, and replays from a dead-letter queue. Section 6.5 showed how an idempotency key turns `POST /orders` into a safe retry, and Section 6.4 how idempotent consumers deduplicate messages. The architectural rule is simple: before adding a retry, check that the operation is idempotent, and if it is not, add a key or a duplicate check first. A retry policy on a non-idempotent payment call is not a resilience pattern; it is a double-charging generator.

Hedged requests

Retries deal with failures; hedged requests deal with slowness. Section 2.4 introduced them from Dean and Barroso's *The Tail at Scale* (2013): send a request to one replica, and if it has not answered within, say, the 95th-percentile latency, send a copy to a second replica and use whichever reply comes first, cancelling the other. Because only the slowest few per cent of requests are duplicated, the extra load is small, while the tail shrinks dramatically, as the benchmark quoted in Section 2.4 showed.

Hedging has preconditions. The operation must be idempotent, normally a read. There must be independent replicas, so that the second copy is unlikely to be slow for the same reason. The extra requests must be budgeted and switched off under overload, because when every replica is slow, hedging doubles the load on a system that is already struggling, exactly like an unbudgeted retry. Used carefully on read-heavy, fan-out paths such as search and menu lookups, it is one of the most effective tail-latency tools available.

The patterns at a glance

Pattern	Protects against	Main cost or risk
Timeout	Waiting indefinitely; resources held by dead calls	Too short fails healthy slow calls; too long protects nothing
Deadline propagation	Work continuing after the caller has given up	Needs support in every service and library on the path
Retry with backoff and jitter	Transient failures	Extra load; duplicates unless idempotent
Retry budget	Retry storms multiplying load	Some transient failures are not retried during bad periods
Circuit breaker	Repeatedly calling a dependency that is down	Tuning; may cut off a dependency that is only partly failing
Bulkhead	One dependency or caller consuming all resources	Reserved capacity cannot be shared
Fallback and degradation	Total failure of a feature	Untested code paths; stale or approximate results
Hedged request	Tail latency from slow replicas	Extra load; reads only

Checklist: every remote call

Review each integration point, including databases, caches, brokers and third-party APIs, against this list.

- [ ] **Connect and request timeouts are set explicitly**
  Chosen from the dependency's measured high percentiles and the caller's own deadline, never left at a library default.
- [ ] **The remaining deadline is propagated**
  Downstream calls receive the time left, and work stops when it runs out.
- [ ] **Retries are limited and safe**
  Only transient errors, only idempotent operations or those with an idempotency key, at one layer, with capped exponential backoff and jitter.
- [ ] **A retry budget caps the extra load**
  Retries stay a small fraction of traffic (for example 10%), so an outage cannot multiply load.
- [ ] **A circuit breaker guards the dependency**
  Thresholds, wait time and what counts as a failure are tuned and its state is exported as a metric.
- [ ] **Concurrency to the dependency is bounded**
  A bulkhead (dedicated pool or semaphore) stops this dependency consuming every thread or connection.
- [ ] **Failure behaviour is decided**
  A fallback, a degraded response or a clear error, agreed with the product owner, and fail-closed for security decisions.
- [ ] **Timeouts, retries, breaker trips and fallbacks are observable**
  Counted, graphed and alerted on, so silent degradation is noticed (Section 9.6).
- [ ] **The failure path is tested**
  Fault injection or a game day shows that the timeouts, breaker and fallback actually work (Section 9.8).

WATCH OUT

Common mistakes

- Leaving timeouts at library defaults, which are often infinite or minutes long.
- Downstream timeouts longer than upstream ones, so work continues after the caller has gone.
- Retrying at every layer, retrying non-idempotent operations, or retrying without jitter.
- Counting 4xx responses as failures in a circuit breaker, so one buggy client trips it for everybody.
- One shared thread or connection pool for all dependencies, so the slowest decides for all.
- Fallbacks that are never exercised and fail, or overload something else, the first time they are needed.

CHECK YOUR UNDERSTANDING

▼ A service receives 50 requests per second, each calling a dependency with a 4-second timeout. If the dependency hangs, how many threads can these calls tie up?

By Little's law about 50 × 4 = 200 threads, which is why the timeout and a bulkhead limit must fit the pool size.

▼ Why is full jitter better than plain exponential backoff when many clients fail at once?

Plain backoff makes all clients retry at the same moments, in synchronised waves; random delays spread the retries out so the dependency sees a smooth, lower load.

▼ In which state does a circuit breaker make no remote calls, and what ends that state?

In the open state; when the wait period expires it becomes half-open and lets trial calls through.

▼ Why should an authorisation check not fall back to allowing access when the authorisation service is down?

Security decisions must fail closed: allowing access during an outage would let anyone in whenever the service failed or was attacked.

▼ **Try it: The gateway and the order service each make up to 4 attempts at the payment call, a POST with no idempotency key and no timeout. The provider slows down. What is the worst case, and what three changes would you make?**

Each incoming request can cause 4 x 4 = 16 payment calls, which adds load to a struggling provider and could charge customers twice. Threads also wait forever. First, set a timeout from measured percentiles, and pass on the remaining deadline. Second, retry at one layer only, with capped backoff, jitter and a retry budget, and only with an idempotency key. Third, add a circuit breaker and a bulkhead, so that checkout's failure cannot exhaust threads for other features.

SOURCES

- 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/
- Amazon Builders' Library: Avoiding fallback in distributed systems: https://aws.amazon.com/builders-library/avoiding-fallback-in-distributed-systems/
- Metastable Failures in Distributed Systems (HotOS 2021): https://sigops.org/s/conferences/hotos/2021/papers/hotos21-s11-bronson.pdf
- Metastable Failures in the Wild (OSDI 2022): https://www.usenix.org/conference/osdi22/presentation/huang-lexiang
- Metastable Failures in Distributed Systems (Marc Brooker's commentary): https://brooker.co.za/blog/2021/05/24/metastable/
- Static stability using Availability Zones (Amazon Builders' Library): https://builder.aws.com/content/3K9SiqiJ369TGkkGzqPM95BuYWI/static-stability-using-availability-zones
- Caching challenges and strategies (Amazon Builders' Library): https://builder.aws.com/content/3K9TeXp6strMiJ4vhakpRD7BQWO/caching-challenges-and-strategies
- Developing with asyncio (Python documentation): https://docs.python.org/3/library/asyncio-dev.html
- Don't Block the Event Loop (or the Worker Pool) (Node.js): https://nodejs.org/en/learn/asynchronous-work/dont-block-the-event-loop

- Virtual Threads (Oracle Java 21 documentation): https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html

SLIs, SLOs, SLAs and error budgets

BEFORE YOU READ THIS

This section builds on:

- 2.4 Performance and scalability
- 2.5 Availability and reliability

Terms it uses from earlier: Availability, Reliability, Percentile (p95, p99), Latency

THE IDEA IN BRIEF

Imagine a bus company that promises 'nearly all buses on time'. To make that useful, it must say what counts as on time (within two minutes), how it measures it (at each stop) and what 'nearly all' means (99 in every 100 over a month). If it misses that, the timetable team stops trying new routes and fixes punctuality first. Because perfection is impossible and very expensive, the one late bus in a hundred is an allowance it can spend on roadworks and new routes. Software reliability works exactly like this.

WHAT YOU NEED TO KNOW

- An SLI is a user-centred measurement, best expressed as good events divided by valid events; an SLO is a target for it over a window; an SLA is a contract with consequences, set looser than the SLO.
- Useful SLIs cover availability, latency and quality for request-driven services, freshness, correctness and coverage for pipelines, and durability for storage, measured as close to the user as is practical.
- The error budget is one minus the SLO: 99.9% over 30 days allows 43.2 minutes of full outage or 0.1% of requests, and a written error budget policy says what happens when it is spent.
- Burn rate is the observed error ratio divided by the allowed ratio; budget spent equals burn rate times alerting window divided by SLO window, so a burn rate of 14.4 for one hour spends 2% of a 30-day budget.
- The SRE Workbook's multiwindow, multi-burn-rate alerts page at 2% of budget in one hour or 5% in six hours and ticket at 10% in three days, each confirmed by a short window one twelfth as long.
- Common mistakes are 100% targets, too many SLOs, SLIs that users do not experience, measuring in the wrong place, and SLOs with no policy or alerts behind them.

Three terms, one idea

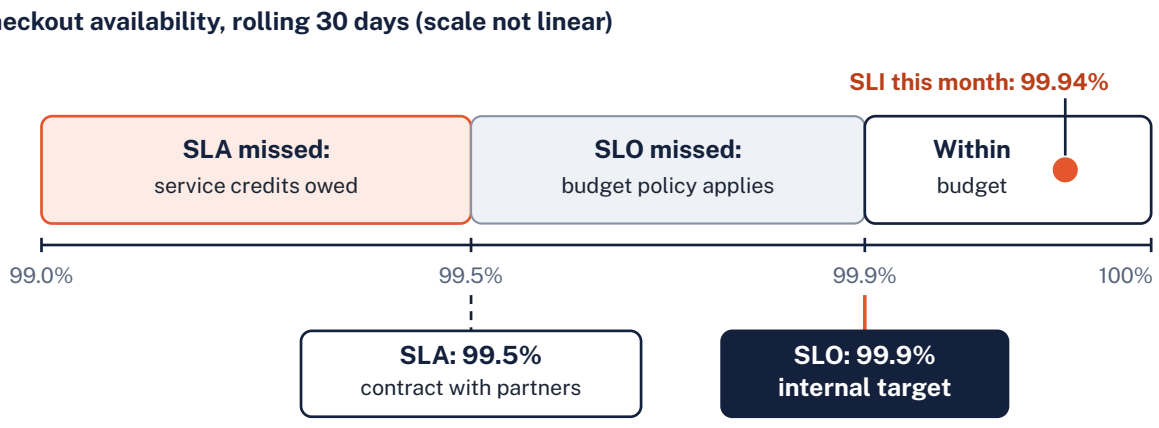
Chapter 2 introduced availability targets and the nines (Section 2.5) and showed that 99.9% over 30 days leaves 43.2 minutes. This section turns that arithmetic into an operating discipline. The vocabulary comes from Google's *Site Reliability Engineering* book (Beyer, Jones, Petoff and Murphy, O'Reilly, 2016) and its companion, *The Site Reliability Workbook* (2018), and is now used far beyond Google.

A service level indicator (SLI) is a carefully defined, quantitative measure of some aspect of the service users receive. The most useful form is a ratio: good events divided by valid events. "The proportion of checkout requests that return a successful response within 800 ms, measured at the load balancer" is an SLI. It ranges from 0% (nothing works) to 100% (nothing is broken), and the ratio form means every SLI can be read, compared and alerted on the same way.

A service level objective (SLO) is a target for an SLI over a stated window: "99.9% of checkout requests are good, measured over a rolling 30 days." It is an internal promise that the team, the product owner and the people who depend on the service agree to. A service level agreement (SLA) is a contract with customers that names consequences, usually service credits, if a level is missed. Many services have SLOs but no SLA; internal platforms almost never have SLAs at all.

The idea that ties the three together is the error budget. If the target is 99.9%, then 0.1% of events may be bad without anyone having failed. That 0.1% is not waste; it is the room the team has to take risks: releasing features, running migrations, experimenting with configuration. The budget turns an argument about feelings ("is the service reliable enough?") into an argument about numbers ("have we spent our budget?"), and gives product and engineering a shared rule for when to move fast and when to slow down.

SLI, SLO and SLA on one scale



The SLI is the measurement, the SLO the internal target and the SLA a looser contractual floor; the gap between SLO and SLA is the warning zone.

Choosing good SLIs

An SLI is only as useful as its closeness to what users experience. The Workbook offers a short menu that covers most services. Request-driven services (APIs, web pages) need **availability** (the proportion of valid requests served successfully), **latency** (the proportion served faster than a threshold) and sometimes **quality** (the proportion served without degradation, such as a menu shown without personalised recommendations because that dependency was down). Data pipelines need **freshness** (the proportion of data updated more recently than a threshold), **correctness** (the proportion of output that is right) and **coverage** (the proportion of input processed). Storage systems add **durability** (the proportion of written data that can still be read).

Latency SLIs are written as proportions, not averages: “99% of menu requests faster than 300 ms”. This connects directly to the percentiles of Section 2.4: saying 99% of requests must be under 300 ms is the same as saying p99 must be under 300 ms, but the proportion form adds up cleanly across time and servers (count the good ones and the valid ones), while percentiles do not. It is common to set two latency thresholds, for example 90% under 200 ms and 99% under 800 ms, to capture both typical and tail experience.

Where you measure matters as much as what you measure. The Workbook separates the *SLI specification* (the user-relevant outcome, such as “orders placed successfully”) from the *SLI implementation* (how you count it: server logs, load balancer metrics, synthetic probes or instrumentation in the mobile app). Measuring inside the application server misses requests that never reached it because DNS, the content delivery network or the load balancer failed. Measuring in the client sees everything but includes users' bad mobile connections and arrives late. Most teams start with load balancer metrics, which are cheap and close to the edge, and add synthetic probes for low-traffic paths.

Define what counts as *valid* and what counts as *good* precisely, because the edge cases decide whether the number means anything. Is a 404 for a mistyped URL bad? Usually not: it is a correct answer. Is a 429 rate-limit response bad? It depends on whether the client was within its agreed quota. Are health checks and load tests valid events? No; exclude them, or they dilute the ratio. Write these rules down next to the SLO.

SLIs for the running examples

Journey	SLI type	SLI specification (good / valid)	Example SLO
Food delivery: browse menus	Latency	Menu requests answered in under 300 ms / all valid menu requests	99% over 30 days
Food delivery: place order	Availability	Order requests answered with success or a correct business refusal / all valid order requests	99.9% over 30 days
Food delivery: courier location	Freshness	Map views whose courier position is under 30 s old / all map views	99% over 30 days
Food delivery: restaurant payouts	Correctness	Payout batches reconciled without discrepancy / all payout batches	99.99% over a quarter
Ticketing: on-sale checkout	Latency and availability	Seat holds confirmed within 2 s / all hold attempts by admitted fans	99.5% during each on-sale
Ticketing: e-ticket delivery	Freshness	Tickets in the fan's app within 5 minutes of payment / all paid orders	99.9% over 30 days

Setting the target

A target is a business decision, not a technical one. The cost of each extra nine rises steeply, while users stop noticing improvements beyond some point because their own phone network, Wi-Fi and devices fail more often than that. Google's SRE book makes the point bluntly: aiming for 100% is wrong for almost everything, because it leaves no budget for change, costs a fortune and cannot be achieved anyway. A 100% target also turns every single failure into a policy breach, which teaches people to ignore the SLO.

Good starting points are past performance (what did the service achieve over the last quarter, and were users happy?), the targets of the services you depend on (you cannot promise 99.99% on top of a 99.9% database, as Section 6.1 showed for chains), and the business cost of unreliability. For the food-delivery platform, failed orders at lunchtime cost revenue and customer trust, so 99.9% on order placement is defensible; courier position updates are less critical and 99% is enough. Begin a little below current performance, so the team is not instantly in breach, and tighten later with evidence.

Choose the window deliberately. A rolling window such as the last 28 days matches how users remember outages (they do not forget on the first of the month), and 28 days contains exactly four of each weekday, which keeps weekly patterns fair; this section uses 30 days because the published burn-rate figures are based on it, and the sums work the same way for 28. Calendar

months suit reporting and contracts. Event-based services such as a ticket on-sale may need an SLO per event, because an hour of failure during the on-sale matters far more than a quiet week.

Error budgets in numbers

The error budget can be counted in time or in events. Time-based budgets suit services that are either up or down; request-based budgets are fairer for services with uneven traffic, because an outage at 03:00 affects fewer users than one at noon.

WORKED EXAMPLE

Worked example: the order-placement budget

The food-delivery platform's order API has an SLO of 99.9% good requests over 30 days. It handles about 3 million valid order requests in that period, heavily concentrated at lunch and dinner.

Budget as a fraction	1 - 0.999 = 0.001, or 0.1%
Budget in time	0.001 × 30 days × 1,440 minutes = 43.2 minutes of total outage
Budget in requests	0.001 × 3,000,000 = 3,000 bad requests
Incident 1: bad deployment at 15:00	12 minutes × 800 requests per minute = 9,600 requests; 40% failed = 3,840 bad requests
Result	3,840 is more than the whole 3,000-request budget: one incident exhausts the month
Same incident at 03:00	12 × 50 requests per minute × 0.4 = 240 bad requests, 8% of the budget

Request-based budgets weight failures by how many users they hit, which is why the same 12 minutes can cost 8% or more than 100% of the budget.

The time-based view (43.2 minutes) would have scored both incidents identically, at 12 × 0.4 = 4.8 minutes of equivalent outage, about 11% of the budget.

Error budget policies

A budget means nothing unless something happens when it runs out. An error budget policy is a short document, agreed in advance by engineering, product and leadership, that says what the team will do. The example policy in the Workbook measures the budget over a trailing four weeks; if the service exceeds its budget, all changes and releases stop except urgent fixes (P0

issues) and security fixes until the service is back within its SLO. It also requires a postmortem with at least one top-priority action item whenever a single incident consumes more than 20% of the four-week budget, and escalates disagreements to the chief technology officer.

The value of a policy lies in being agreed before it bites. In the middle of a release crunch, nobody wants to freeze; with a signed policy, the freeze is a pre-agreed rule rather than a fight between an SRE and a product manager. Good policies also say what does *not* count against the team: a company-wide network outage, a dependency that is itself frozen, or traffic outside the SLO's scope such as a load test.

Practitioners disagree about how mechanical the policy should be. Some teams apply it strictly, because exceptions erode trust in the whole system. Others treat the budget as information for a conversation, arguing that a hard freeze can delay the very fixes that would restore reliability. A middle path is common: freeze risky feature releases automatically, keep shipping reliability fixes, and allow a named executive to grant documented exceptions. The opposite case also matters: a team that never spends its budget is probably too cautious and could ship faster, run more experiments or lower its costs.

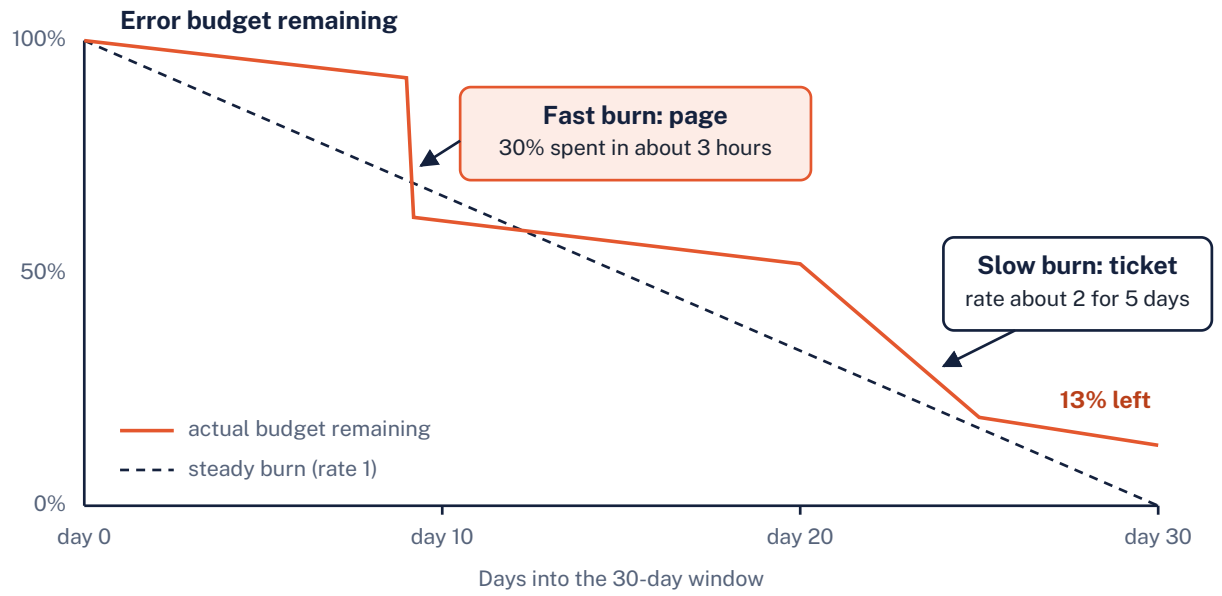
Burn rate and alerting on SLOs

Alerting on raw error rates (“page if more than 1% of requests fail for 5 minutes”) is noisy: short blips page people at night for problems that would never threaten the SLO, while a steady 0.5% failure rate can quietly use up the month without ever crossing the threshold. The Workbook's answer is to alert on the burn rate: how fast the budget is being spent compared with the rate that would use it up exactly at the end of the window.

Burn rate = observed error ratio / allowed error ratio. With a 99.9% SLO the allowed ratio is 0.1%, so an error ratio of 0.1% is a burn rate of 1 (the budget lasts exactly the window), 1% is a burn rate of 10, and 100% (a full outage) is a burn rate of 1,000. The fraction of the window's budget spent during an alerting window is burn rate x alerting window / SLO window. For a 30-day (720-hour) window, a burn rate of 14.4 sustained for one hour spends 14.4 × 1 / 720 = 0.02, that is 2% of the month's budget; at that rate the whole budget would be gone in 720 / 14.4 = 50 hours.

The Workbook recommends a multiwindow, multi-burn-rate alert as a sensible default for a 99.9% SLO: page if 2% of the budget is spent in one hour (burn rate 14.4) or 5% in six hours (burn rate 6), and open a ticket if 10% is spent in three days (burn rate 1). Each condition must hold over a long window and a short one, with the short window one twelfth of the long (5 minutes, 30 minutes and 6 hours respectively). The long window ensures the problem is significant; the short window ensures it is still happening, so the alert stops firing within minutes of a fix instead of staying red for an hour.

Fast and slow burns



A fast burn spends a large share of the budget in hours and should wake someone; a slow burn spends it over days and belongs in a 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%
```

Burn rate, budget spent and a two-window paging rule. In production the same logic is written as alerting rules in the monitoring system (for example Prometheus recording and alerting rules).

The thresholds also fix detection time. If the order API fails completely (error ratio 100%, burn rate 1,000), the one-hour error ratio crosses 1.44% after 0.0144 × 60 = about 0.9 minutes, so the page fires in under a minute. If 2% of requests fail (burn rate 20), the one-hour ratio reaches 1.44% after 0.72 × 60 = about 43 minutes. Severe problems page fast; mild ones page only if they persist, which is exactly the behaviour people want at three in the morning.

Low-traffic services break this arithmetic: with 10 requests an hour, one failure is a 10% error ratio and a burn rate of 100. The Workbook's remedies are synthetic traffic to add signal, grouping related low-traffic services under one SLO, making single failures matter less through retries and fallbacks, or accepting a longer window. For the ticketing system, which is almost idle between on-sales and then extremely busy, per-event SLOs with burn-rate alerts during the event window work better than a monthly figure.

SLAs and service credits

An SLA moves the target from engineering into contract law. It defines the service covered, how availability is measured, exclusions (scheduled maintenance, customer misuse, force majeure), the remedy and how a customer claims it. The usual remedy is a service credit: a percentage of the monthly fee returned, tiered by how badly the level was missed. Credits are typically small compared with the customer's actual loss, which is why SLAs are better read as a signal of the provider's confidence than as insurance; customers whose losses would be large need their own resilience, not a bigger credit.

Consider an illustrative SLA the food-delivery platform might offer restaurant partners for its order-intake API: 99.5% monthly availability, with a 10% credit on that month's platform fee below 99.5% and 25% below 99.0%. The internal SLO is 99.9%. The gap between them is deliberate: the team should exhaust its SLO budget, apply its policy and recover long before the contract is at risk. Never set the SLO equal to the SLA; you would learn about trouble only when you already owe money.

Two architectural consequences follow. First, an SLA can only be honest if you know the SLAs of your own dependencies (cloud provider, payment processor, map provider) and how they combine. Second, SLAs need evidence: the measurement method in the contract must be one you can produce from your own telemetry, which is a requirement on the observability design.

Common mistakes

Too many SLOs. A service with thirty SLOs has none, because nobody can reason about them or act on them. Start with two or three per critical user journey: usually one availability and one or two latency SLOs. **Measuring the wrong thing.** CPU utilisation, queue depth and replica lag are useful causes to watch, but they are not SLIs, because users do not experience them

directly. **Measuring in the wrong place.** Server-side success rates look perfect while the CDN returns errors. **Averaging.** An average latency SLO hides the slow tail that users complain about.

Targets without consequences. An SLO with no budget policy and no burn-rate alerts is a dashboard decoration. **Targets set by engineers alone.** The product owner must agree, because the SLO trades features for reliability. **Never revising.** Review SLOs every quarter or two: if users complain while the SLO is green, the SLI or the target is wrong; if the budget is never touched, the target may be too loose or the team too cautious. **One SLO for very different journeys.** Lumping menu browsing and payment into one ratio lets plentiful cheap requests hide failures in the scarce, valuable ones.

SLOs connect to the rest of this chapter. Burn-rate alerts are the symptom-based alerts recommended in observability; budget-burning incidents feed post-incident reviews; and an SLO's response measure is a natural fitness function for the architecture.

IF YOU ARE NEW TO THIS

If you are new to this

Think of the error budget as a monthly allowance for mistakes. The SLI is the bank statement, the SLO is the allowance you agreed with your family, and the SLA is the loan agreement with the bank that charges you if you go badly overdrawn. Spending the allowance is fine; that is what it is for. Running out means you stop buying treats (new features) until the next month.

CHECK YOUR UNDERSTANDING

▼ **Why is an SLI usually written as good events divided by valid events?**

Because the ratio always lies between 0% and 100%, adds up cleanly across servers and time by counting, and lets every SLI be read, compared and turned into an error budget the same way.

▼ **A 99.9% SLO over 30 days sees a burn rate of 6 for six hours. What fraction of the budget is spent?**

6 × 6 / 720 = 0.05, so 5% of the 30-day budget.

▼ **Why should an SLA be looser than the internal SLO?**

So that the team detects and responds to a spent error budget well before the contract is breached and service credits become payable.

▼ **What does the short window in a multiwindow alert add?**

It confirms the problem is still happening, so the alert resets within minutes of a fix instead of staying active for the whole long window.

▼ **Try it: An SLO is 99.9% over 30 days, on 2 million requests. Over 3 hours of steady traffic, 1% of requests fail. Find the budget in bad requests and minutes, the burn rate and the share of budget used.**

Budget: 0.1% of 2,000,000 = 2,000 bad requests, or 43.2 minutes of full outage. Burn rate: 1% / 0.1% = 10. Budget spent: 10 x 3 hours / 720 hours = about 4.2%. A burn of 10 misses the 14.4 fast-page threshold, but the 5%-in-six-hours alert would fire if it continued, and it is well worth a look. The error budget policy then decides what happens next.

SOURCES

- 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/

SECTION 9.6 OF 9.8 · INTERMEDIATE · 26 MIN READ · LAST REVIEWED NOVEMBER 2026

Observability: logs, metrics and traces

BEFORE YOU READ THIS

This section builds on:

- 2.4 Performance and scalability
- 6.1 Why distribute
- 9.5 SLOs and error budgets

Terms it uses from earlier: Percentile (p95, p99), Service level objective (SLO), Service level indicator (SLI), Utilisation, Personal data

THE IDEA IN BRIEF

A car dashboard shows a few dials and warning lights: speed, fuel, engine temperature. That is monitoring: it tells you about problems someone predicted. When the car makes a strange noise that no light explains, a mechanic plugs in a diagnostic computer and reads detailed records of what every part was doing, so they can work out a fault nobody anticipated. Observability is building software so that this kind of investigation is always possible: every part keeps clear records, and the records of one journey can be followed from part to part.

WHAT YOU NEED TO KNOW

- Monitoring checks predefined conditions; observability is the ability to explain new, unpredicted behaviour from existing telemetry, and must be designed in through consistent instrumentation and shared identifiers.
- Metrics are cheap aggregates for trends and alerts, structured logs record individual events, traces show one request's path through services, and profiles show where code spends resources.
- Counters, gauges and histograms are the core metric types; percentiles cannot be averaged across servers or time, but histogram buckets can be added and the percentile read from the merged result.
- RED (rate, errors, duration) describes services, USE (utilisation, saturation, errors) describes resources, and the four golden signals are latency, traffic, errors and saturation.
- Distributed tracing (Dapper, 2010) links spans through context propagation, standardised by the W3C traceparent header; OpenTelemetry, a CNCF project graduated in 2026, is the vendor-neutral standard for producing traces, metrics and logs.

- Unbounded label values explode metric cardinality and cost, so per-user or per-order detail belongs in traces and logs, and tail-based sampling keeps the traces that matter.
- Page on symptoms users feel, such as SLO burn rates, not on causes such as CPU, and build each dashboard to answer one question for one audience.

Monitoring and observability

Monitoring means collecting predefined measurements and checks and raising an alarm when they cross a threshold: disk 90% full, error rate above 1%, health check failing. It answers questions you knew to ask in advance. Observability is a property of the system: how well you can understand its internal state, including states nobody predicted, from the telemetry it already emits, without shipping new code to find out. The word comes from control theory, where Rudolf Kálmán defined it in 1960 as the ability to infer a system's internal state from its outputs.

The distinction matters because distributed systems mostly fail in new ways. A food-delivery order can be slow because one restaurant's tablet app retries aggressively, because one payment provider's endpoint in one region is degraded, or because a new menu feature makes one query slow only for restaurants with more than 500 items. No one wrote a dashboard for any of these. Monitoring tells you *that* checkout is slow; observability lets you ask *for whom, where and why* by slicing the telemetry by any attribute until the pattern appears.

Practitioners disagree about how far the distinction goes. One view, common among tool vendors, treats observability as an umbrella over three pillars: logs, metrics and traces. Charity Majors, Liz Fong-Jones and George Miranda (*Observability Engineering*, O'Reilly, 2022) argue that the pillars framing misses the point, and that observability requires rich, high-cardinality, structured events that can be queried in arbitrary combinations. Both camps agree on the architectural consequence: observability is a quality attribute that must be designed in, through consistent instrumentation, shared identifiers and context propagation, because it cannot be bolted on during an incident.

Kinds of telemetry

Signal	What it is	Best at	Main cost or limit
Metrics	Numeric time series, aggregated per interval and labelled by a few dimensions	Cheap long-term trends, dashboards, alerting on SLIs	Pre-aggregated: cannot answer questions about individual requests; labels must be low-cardinality
Logs	Timestamped records of discrete events, ideally structured	Detail about one event: an error, a decision, an audit entry	Volume and cost grow with traffic; free text is hard to query
Traces	Trees of timed spans showing one request's path through services	Finding where time goes and which hop failed in a distributed call	Must be sampled at scale; useless if any hop drops the context
Profiles	Samples of where code spends CPU time or allocates memory	Explaining why one service is slow or expensive at the code level	Newer tooling; continuous profiling adds a little overhead
Wide events	One richly attributed record per request per service	Ad hoc slicing by any attribute (customer, version, region)	Needs a store built for high-cardinality queries

Structured logging

A log line such as `Payment failed for order 81F3 after 3001ms` is easy to write and hard to use: every query needs a fragile regular expression, and every developer phrases things differently. Structured logging writes each entry as key-value fields, usually JSON, with agreed names: timestamp, level, service, event name, business identifiers (order, restaurant, courier), outcome, duration and the trace and span identifiers that link the line to its trace. Queries become precise (“all card_authorisation_failed events for provider cardco in the last hour, grouped by http_status”), and logs from different services can be joined.

A few rules keep logs useful and affordable. Log events, not prose; use a fixed event name per kind of event. Use levels consistently, and do not log at error level for conditions nobody needs to act on. Never log secrets, tokens, full card numbers or unnecessary personal data; telemetry pipelines are copied, shipped to third parties and kept for months, so they are a common source of data leaks, and personal data in logs falls under the GDPR like any other copy (Section 8.7). Security logging and detection, a related but different purpose with its own retention and integrity needs, is covered in Section 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"}
```

A W3C traceparent header and a structured log entry from the payment service. The shared trace_id lets a query jump from this log line to the whole trace.

Metrics and why percentiles cannot be averaged

Most metric systems, such as Prometheus (a CNCF graduated project) and the OpenTelemetry metrics model, offer the same few types. A counter only increases, such as total requests served; you read it as a rate (requests per second over the last five minutes), which also survives restarts. A gauge is a current value that rises and falls: queue length, connections in use, memory. A histogram counts observations into buckets (for example requests under 50 ms, under 100 ms, under 250 ms and so on) plus a total count and sum, from which percentiles are estimated. Prometheus also has a summary type that computes quantiles inside each process.

The distinction between histograms and summaries is not a detail. As Section 2.4 noted, a percentile is a property of a whole distribution, and the percentile of a combined distribution cannot be computed from the percentiles of its parts. Averaging p99 values across servers, or across minutes, produces a number that is not a percentile of anything. Bucket counts, by contrast, can simply be added: sum each bucket across all servers, then estimate the percentile from the merged histogram. That is why fleet-wide latency SLIs should be built on histograms (or on counts of requests under a threshold, as in Section 9.5), never on per-instance summaries.

WORKED EXAMPLE

Worked example: averaging two p99s

The menu service runs on two instances, each handling 10,000 requests in a five-minute interval. Instance A is healthy: all its requests take 20 to 300 ms, and its p99 is 180 ms. Instance B has a noisy neighbour: 7,000 requests take 20 to 300 ms, but 3,000 are spread evenly between 500 and 1,100 ms.

B's p99 (its slowest 100 of 10,000)	1,100 - 600 × (100 / 3,000) = 1,080 ms
Average of the two p99s	(180 + 1,080) / 2 = 630 ms
True p99 of all 20,000 requests	The slowest 200 are all on B: 1,100 - 600 × (200 / 3,000) = 1,060 ms
Error from averaging	630 ms reported against 1,060 ms experienced: the dashboard understates the tail by 430 ms
With histograms	Add A's and B's bucket counts, then read the 99th percentile from the merged buckets: about 1,060 ms, limited only by bucket width

Averaged percentiles can be wrong in either direction, and the error grows when instances carry different traffic.

The same applies over time: the p99 for an hour is not the average of twelve five-minute p99s.

What to measure: RED, USE and the golden signals

Three short checklists cover most of what a service should expose. The RED method, proposed by Tom Wilkie in 2015, applies to anything that serves requests: **Rate** (requests per second), **Errors** (failed requests per second) and **Duration** (the distribution of response times). The USE method, from Brendan Gregg, applies to resources: for every CPU, disk, network link, connection pool and queue, check **Utilisation** (how busy it is), **Saturation** (how much work is waiting) and **Errors**. The four golden signals from the Google SRE book's chapter on monitoring distributed systems are **latency**, **traffic**, **errors** and **saturation**: in effect RED plus saturation.

In practice, RED describes what users experience and USE explains why. If the payment service's duration rises (RED), USE on its resources might show a connection pool at 100% utilisation with 40 requests waiting (saturation), which in turn points to the slow card provider

holding connections open. Measure latency separately for successes and failures: a fast error is not good latency, and a timeout is both an error and a slow response.

Three checklists compared

Checklist	Applies to	Signals	Typical question it answers
RED (Wilkie)	Request-driven services	Rate, errors, duration	Is the order service serving users well right now?
USE (Gregg)	Resources: CPU, memory, disks, pools, queues	Utilisation, saturation, errors	Which resource is the bottleneck?
Four golden signals (SRE book)	User-facing systems	Latency, traffic, errors, saturation	Is the system healthy, and how close is it to its limits?

Distributed tracing

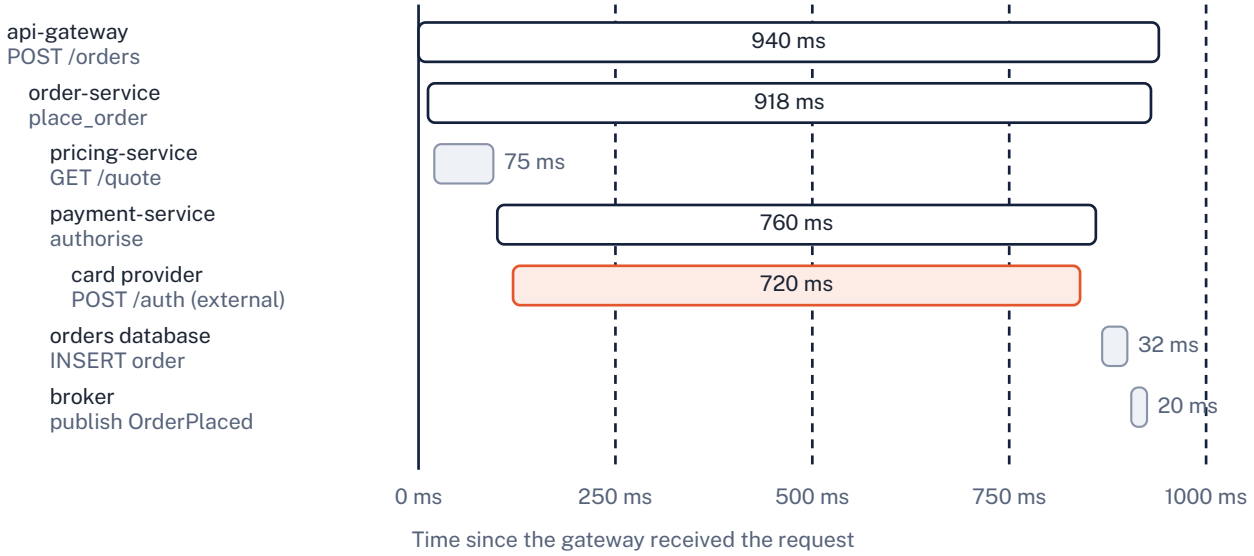
In a monolith a stack trace shows where a slow request spent its time. Across services there is no single stack. Distributed tracing rebuilds one: every request gets a trace identifier at the edge, every unit of work becomes a span (with a name, start time, duration, status, attributes and the identifier of its parent span), and a tracing backend assembles the spans into a tree. Google's Dapper paper (Sigelman and colleagues, 2010) described the approach at scale: low overhead, instrumentation in shared libraries rather than in application code, and sampling so that only a fraction of requests is recorded.

Tracing depends entirely on context propagation: each service must pass the trace context to the next. The W3C Trace Context specification (a W3C Recommendation since 2020) standardises this for HTTP with a `traceparent` header holding a version, a 16-byte trace ID, the 8-byte ID of the calling span and flags such as whether the trace is sampled, plus an optional `tracestate` header for vendor data. For messaging, the same values travel in message headers, and the consumer's span refers back to the producer's. A single service that drops the header (an old library, a hand-written HTTP client, a batch job) breaks every trace that passes through it, which is why propagation belongs in shared platform libraries, not in each team's code.

Recording every span of every request is expensive at scale, so traces are sampled. Head-based sampling decides at the start of the request (keep 1 in 100) and is cheap, but it throws away most of the rare slow and failed requests you care about. Tail-based sampling buffers spans until the request finishes and then keeps all errors, all requests slower than a threshold and a small fraction of the rest; it needs more memory in the collector but keeps the interesting traces.

A trace across the food-delivery services

Trace 4bf92f35…: one Place order request (each bar is a span)



The waterfall shows at once that 720 of the 940 ms are spent waiting for the external card provider, not in the platform's own services.

IF YOU ARE NEW TO THIS

Step by step: following one order through the services

This section's hardest idea is how a trace is stitched together from pieces recorded by different services, none of which sees the whole request. The glue is two identifiers passed along with every call.

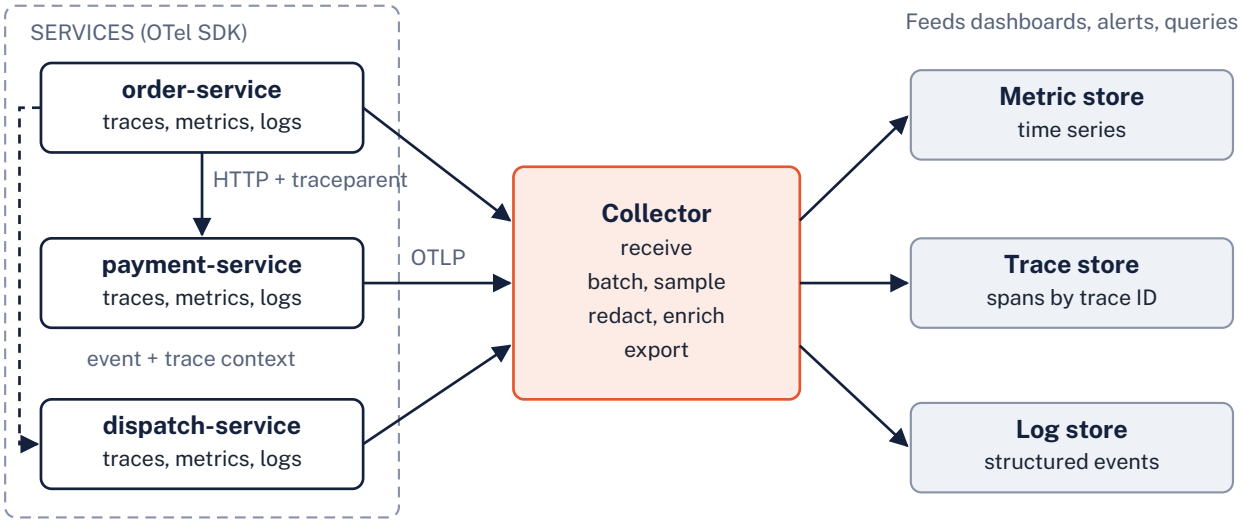
- A customer places an order. The API gateway receives POST /orders, gives the request a new trace ID and starts its own span: a record with a name, a start time and, when it finishes, a duration.
- When the gateway calls the order service, it adds a traceparent header carrying the trace ID and its own span's ID. The order service starts its place_order span and records the gateway's span as its parent.
- The order service does the same on each call it makes, to pricing and to payment, and puts the trace context in the headers of the OrderPlaced event. Every new span records the trace ID and its parent.
- Each service sends its finished spans separately to the tracing backend. The shared trace ID lets the backend gather them, and the parent links let it arrange them into the waterfall in the diagram.
- Now suppose the order service called pricing through a hand-written HTTP client that does not add the header. Pricing would start a new trace of its own, and its spans would be missing from the order's waterfall.
- That is why context propagation belongs in shared libraries that every team uses: one hop that forgets breaks every trace passing through it.
- **The lesson:** each service records only its own part of a request. The trace ID and parent links carried on every call turn those parts into one story, so a trace is only as complete as its weakest hop.

OpenTelemetry: one standard for instrumentation

Until about 2019, each tracing and monitoring vendor had its own agents and libraries, so changing vendor meant re-instrumenting every service. OpenTelemetry (OTel) was formed in 2019 by merging two earlier open-source projects, OpenTracing and OpenCensus, under the Cloud Native Computing Foundation (CNCF), and graduated as a CNCF project in May 2026. It provides language APIs and SDKs, automatic instrumentation for common frameworks and clients, the OTLP wire protocol, the Collector, and semantic conventions that fix attribute names such as `http.request.method` and `http.response.status_code` so that telemetry from different teams and languages lines up. Traces, metrics and logs are stable; profiles were added as a fourth signal and entered public alpha in 2026.

The architectural value is the separation of instrumentation from backend. Services emit OTLP to a Collector, usually running beside each node or as a central gateway; the Collector batches, samples, removes sensitive attributes, adds metadata such as the deployment version, and exports to whichever stores the organisation uses. Changing vendor, or sending the same data to two backends during a migration, becomes a Collector configuration change instead of a code change in two hundred services. A platform team typically provides the SDK setup, propagation and Collector as a paved road, so that product teams only add business attributes.

Telemetry pipeline



Instrumentation lives in the services, processing in the Collector and storage in replaceable backends.

```
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
```

Manual instrumentation with the OpenTelemetry Python API and a structured logger. Auto-instrumented HTTP clients create child spans and propagate context without extra code.

Cardinality and cost

Telemetry can easily cost more than the system it observes. The main driver for metrics is cardinality: every distinct combination of label values is a separate time series that must be stored and indexed. Labels with a few values (HTTP method, status class, region) are fine. Labels with unbounded values (user ID, order ID, full URL with identifiers in it, error message text) multiply series without limit and can bring a metrics system down.

WORKED EXAMPLE

Worked example: counting time series

The order service exports a request-duration histogram with 12 buckets. Prometheus stores each bucket as a series, plus a sum and a count, so 14 series per label combination.

Route and method pairs	60
Status classes (1xx to 5xx)	5
Instances	30
Series for this one metric	60 × 5 × 30 × 14 = 126,000
Add a region label (5 regions)	126,000 × 5 = 630,000 series
Add restaurant_id (40,000 restaurants)	126,000 × 40,000 = about 5 billion potential series: unworkable

Per-restaurant or per-customer questions belong in traces, logs or wide events, where high cardinality is expected, not in metric labels.

Exemplars, which attach a sample trace ID to a histogram bucket, link a latency spike on a cheap metric to an expensive, detailed trace.

Other cost levers are sampling traces (tail-based sampling keeps the useful ones), shorter retention for detailed data and longer retention for aggregates, dropping debug logs in production by default, and charging telemetry back to the teams that produce it so that its cost is visible. Treat telemetry volume as a design parameter with a budget, like any other resource.

Alerting on symptoms, not causes

The SRE book's guidance is that pages should be about symptoms users feel, not about the many possible causes. High CPU, a full disk on one replica or a restarted container are causes: often harmless, sometimes already handled by automation, and never a complete list of what can go wrong. A burning SLO, failing checkouts or a stalled courier-position pipeline are symptoms: they matter whatever their cause. Symptom-based alerts are fewer, rarely false and cover failures nobody predicted; cause-based signals belong on dashboards that help diagnose a symptom once it is paged.

Every page should be urgent, actionable and real. Alert fatigue is a reliability risk in itself: on-call engineers who are woken by meaningless alerts learn to dismiss them and miss the one that matters. A practical rule is to page on SLO burn rates (Section 9.5), send slower burns and resource warnings to a ticket queue, and link every page to a runbook and a dashboard.

DECISION GUIDE

Decision guide: should this alert page a human?

Apply these questions to every alert rule before it is allowed to wake someone, and again whenever it fires without anyone needing to act.

Q1 **Does the condition mean users are being hurt now, or will be soon (an SLO burning fast, a journey failing)?**

- **Yes** Continue to the next question.
- **No** Do not page. Put it on a dashboard for diagnosis, or open a ticket if someone should look in working hours.

Q2 **Does it need a human to act, rather than an automatic response such as a restart or scale-out?**

- **Yes** Continue to the next question.
- **No** Automate the response and alert only if the automation fails.

Q3 **Must the action happen before the next working day?**

- **Yes** Page, and link the alert to a runbook and the relevant dashboard.
- **No** Send it to the team's ticket queue instead of the pager.

Q4 **Has this alert paged in the last month without anyone needing to do anything?**

- **Yes** Fix the threshold or the signal, or delete the alert: a page nobody acts on teaches people to ignore pages.
- **No** Keep it, and review it at the next on-call handover.

Dashboards that answer questions

A wall of fifty graphs answers no question quickly. Good dashboards are designed like any user interface: for a named audience and a specific question. A service overview answers “is this service healthy?” with its SLOs, remaining error budget and RED metrics at the top, and the USE metrics of its key resources below. A journey dashboard for the on-sale answers “are fans getting tickets?” with admissions, seat holds, payments and failures per minute end to end. Diagnostic dashboards are reached by drilling down, not shown first.

Keep dashboards in version control as code, give each an owner, show deployments and feature-flag changes as annotations on the graphs (most incidents follow a change), and delete dashboards nobody has opened for months. During an incident, the first useful view is usually the user-facing journey, the second the trace of a failing request, and only then the resource metrics.

WATCH OUT

Common observability mistakes

- Adding tracing to a few services only: one hop without propagation breaks every trace through it.
- Putting user IDs, order IDs or raw URLs into metric labels.
- Averaging percentiles across instances or time windows.
- Measuring only on the server, so failures in DNS, the CDN or the load balancer are invisible.
- Logging personal data and secrets, turning the log store into the least protected copy of customer data.
- Alerting on every cause, so that real symptoms are lost in noise.

CHECK YOUR UNDERSTANDING

▼ **Why does the p99 for a whole fleet have to be computed from merged histograms rather than from each instance's p99?**

Because a percentile of combined data is not a function of the parts' percentiles; bucket counts add exactly, so the merged histogram gives a true (bucket-limited) estimate, while averaged p99s can be badly wrong.

▼ **What breaks if one service in a call chain does not forward the traceparent header?**

Every trace that passes through it is split: spans after that hop start a new trace, so the end-to-end picture and the link between cause and symptom are lost.

▼ **Why is order_id a poor metric label but a good span attribute?**

As a label it creates a new time series per order, exploding cardinality and cost; as a span or log attribute it is stored once per event, where high cardinality is expected and useful for queries.

▼ **Which of these should page someone: CPU at 95% on one instance, or checkout SLO burning at rate 20?**

The SLO burn, because it is a symptom users feel; the CPU reading is a possible cause that belongs on a diagnostic dashboard.

▼ **Try it: A latency histogram has 10 buckets, plus sum and count, across 40 routes, 5 status classes and 20 instances. How many time series? A developer adds a customer_id label for 200,000 customers. Work out the effect and advise.**

12 x 40 x 5 x 20 = 48,000 series. Adding customer_id could multiply that by 200,000, up to 9.6 billion, which would be unaffordable and may break the metrics system. Metrics suit bounded labels. Put customer ID in traces and structured logs instead, where one person's request can be followed. Link them through the trace ID, so a chart showing a latency spike leads to a trace from an affected customer.

SOURCES

- 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/
- Charity Majors, Liz Fong-Jones and George Miranda: Observability Engineering (O'Reilly, 2022): https://www.oreilly.com/library/view/observability-engineering/9781492076438/

SECTION 9.7 OF 9.8 · ADVANCED · 25 MIN READ · LAST REVIEWED NOVEMBER 2026

Capacity planning and performance testing

BEFORE YOU READ THIS

This section builds on:

- 2.4 Performance and scalability
- 9.1 Scaling
- 9.3 Queues and back pressure

Terms it uses from earlier: Headroom, Little's law, Percentile (p95, p99), Autoscaling, Fitness function

THE IDEA IN BRIEF

Before a big wedding, a caterer estimates how many guests will come, works out how many plates each cook can serve in an hour, adds a few spare staff in case someone is ill, and orders the extra ovens weeks ahead, because they cannot be bought on the day. Then they hold a rehearsal at full size to check the kitchen really copes. Software teams do the same: they forecast demand, measure what each server can handle, add spare capacity, order what takes time to get, and rehearse with realistic simulated customers.

WHAT YOU NEED TO KNOW

- Capacity planning forecasts peak demand, measures capacity per unit under realistic conditions, adds headroom and redundancy, and orders anything with a long lead time, including quotas and third-party limits.
- Load, stress, soak, spike and breakpoint tests answer different questions: meeting targets at peak, failure behaviour beyond it, degradation over time, reaction to sudden surges and the true limit.
- Closed workload models slow down when the system does, while open models keep arriving at a set rate; public traffic and on-sales are open, so closed-model tests can be dangerously optimistic (Schroeder, Wierman and Harchol-Balter, 2006).
- Coordinated omission (Gil Tene) hides slow responses when a tester waits for replies before sending; measure from the intended send time with open-model generators and full histograms.
- k6, Gatling, JMeter and Locust are the main open-source tools; realistic scenarios, data volumes, hot keys, dependency latency and an unloaded generator matter more than the tool.

- Performance budgets, such as Core Web Vitals thresholds or a per-hop latency allocation, are checked automatically in the delivery pipeline as fitness functions.
- The scarcest shared resource, often a third party, sets the safe admission rate for an event such as a ticket on-sale.

Capacity is a plan, not a hope

Capacity planning answers one question ahead of time: will we have enough of everything when demand peaks? “Everything” is broader than servers. It includes database connections and write throughput, cache memory, message broker partitions, cloud account quotas, third-party rate limits, licences and the people on call. Cloud elasticity makes some capacity quick to add, but not all: a database primary cannot be doubled in seconds, a payment provider may need weeks of notice to raise a limit, and autoscaling (covered in Section 9.1) reacts in minutes to a step change that arrives in seconds.

The method has four steps, repeated on a regular cycle (quarterly for steady growth, per event for launches and on-sales):

The capacity planning cycle

1. **Forecast demand**
   Combine organic growth (orders per day rising 4% a month), seasonality (lunch and dinner peaks, Friday nights, Christmas) and known events (a marketing campaign, a stadium on-sale). Express demand in the units the system scales by: requests per second per journey, writes per second, messages per second, at the peak, not the daily average.

2. **Measure capacity per unit**
   Find, by testing, how much load one instance, one database or one partition can take while still meeting its latency SLO. The knee of the response-time curve, where queueing starts to dominate (Section 2.4), is the practical limit, not the point where it crashes.

3. **Add headroom and redundancy**
   Size for the forecast peak plus a margin for forecast error, bursts and retries, and so that the loss of an instance or a whole availability zone still leaves enough capacity (N+1 or zone-level redundancy).

4. **Plan lead times and review**
   List what takes time to obtain (quota increases, reserved capacity, database upgrades, third-party limits, hardware for on-premises systems) and order it early enough. Afterwards compare actual demand with the forecast and improve the model.

Sizing arithmetic with Little's law was worked through in Section 9.1; this section concentrates on how to obtain trustworthy numbers to put into it. The most common error in capacity planning is not bad arithmetic but a per-instance capacity measured under unrealistic conditions, and the second is forgetting a shared constraint, such as a third-party limit, that no amount of headroom in your own servers can fix.

Kinds of performance test

Performance testing is the only way to measure capacity, find bottlenecks before customers do and verify that a release has not made things worse. The tests differ in the load shape they apply and the question they answer. A load test checks the expected peak; a stress test goes beyond it to see how the system fails; a soak test runs for hours to expose slow leaks; a spike test checks reactions to sudden jumps; a breakpoint test ramps up until something breaks, to find the true limit.

Performance test types

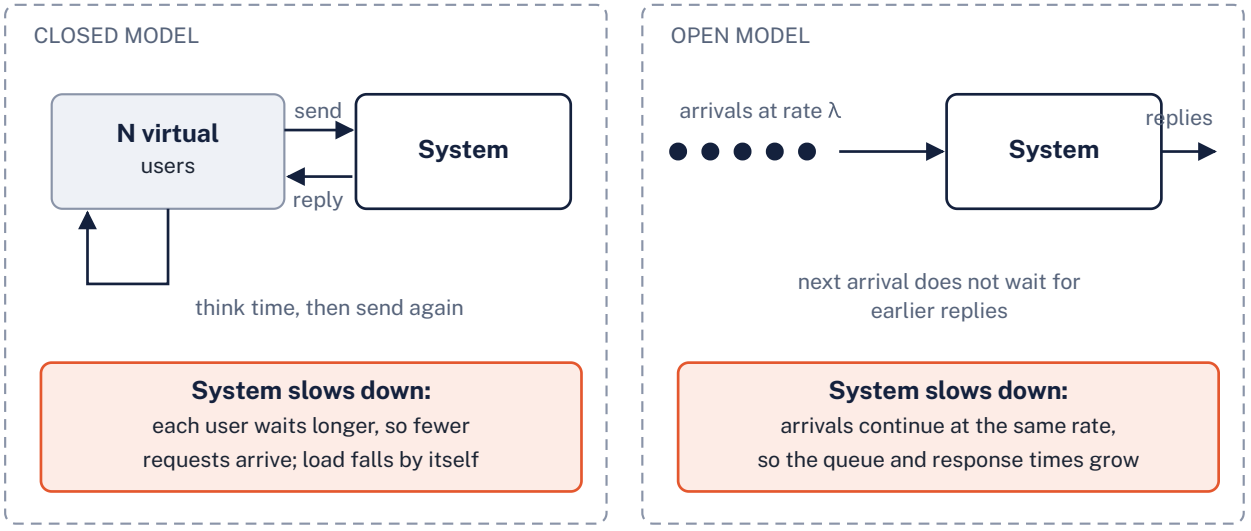
Test	Question it answers	Load shape	Typical duration	Watch for
Smoke	Does the test script and environment work at all?	A few users	Minutes	Script errors, wrong environment
Load	Do we meet our latency and error targets at expected peak?	Ramp to forecast peak, hold	30-60 minutes	p99 latency, error rate, resource saturation
Stress	How does the system fail beyond peak, and does it recover?	Ramp to 1.5-3x peak, then back down	30-60 minutes	Graceful degradation, load shedding, recovery time
Soak (endurance)	Does anything degrade over time?	Normal or peak load, held	Hours to days	Memory leaks, growing queues, connection leaks, disk filling
Spike	Can we absorb a sudden surge?	Jump from low to very high in seconds	Minutes	Autoscaling lag, queue overflow, rate limiter behaviour
Breakpoint (capacity)	Where is the limit and what is the first bottleneck?	Steady ramp until failure	Until it breaks	The knee in latency, the first saturated resource

Open and closed workload models

Load generators simulate users in one of two ways, and choosing the wrong one produces results that are not just inaccurate but dangerously optimistic. In a closed workload model the tool runs a fixed number of virtual users; each sends a request, waits for the response, pauses for a think time and repeats. If the system slows down, each virtual user sends fewer requests, so the load falls by itself. In an open workload model requests arrive at a set rate, independently of how quickly earlier ones are answered, as real visitors to a public website do: thousands of fans pressing Buy at 10:00 do not wait for each other.

Bianca Schroeder, Adam Wierman and Mor Harchol-Balter showed ("Open Versus Closed: A Cautionary Tale", NSDI 2006) that the two models can give very different response times for the same system and the same average load, and that scheduling or capacity decisions tuned under one can be wrong under the other. Internal systems with a fixed population of clients, such as a call-centre application with 300 agents, are reasonably closed. Public internet traffic, event-driven consumers fed by a queue and on-sales are open, or close to it. For such systems an open-model test is the realistic one; k6's `constant-arrival-rate` and `ramping-arrival-rate` executors, for example, implement it, while its `constant-vus` executor is closed.

Closed versus open



A closed-model test politely backs off when the system struggles; real public traffic does not.

IF YOU ARE NEW TO THIS

Step by step: a test that backs off politely

The idea that needs the most care here is that the way a load generator sends requests can hide overload. In a closed model, simulated users wait for each answer, so a struggling system automatically receives less load; real fans at an on-sale do not wait for each other.

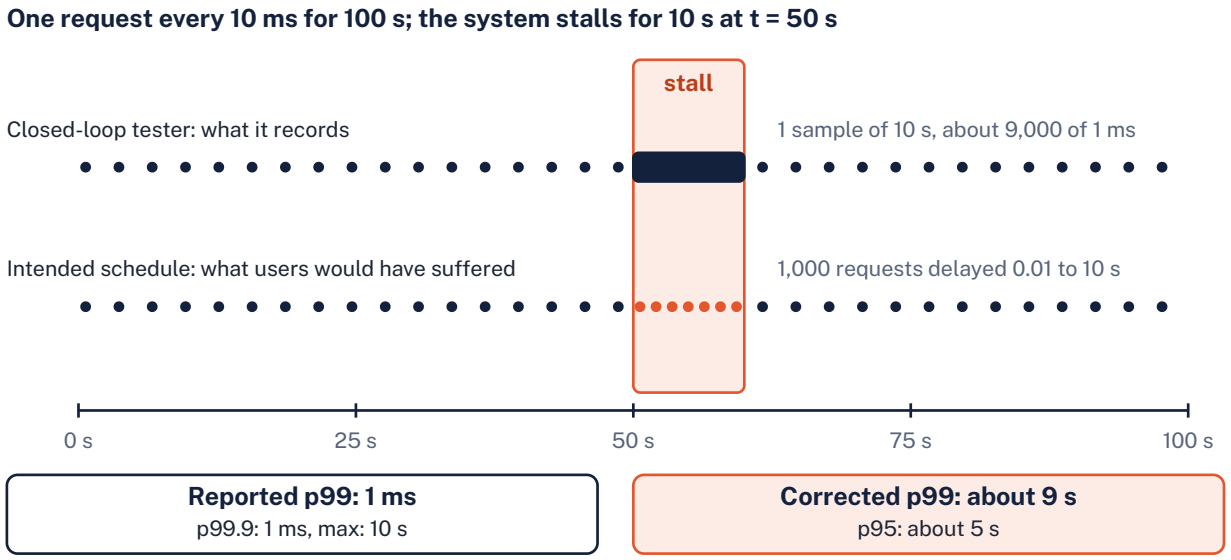
- A team tests the on-sale seat map with 300 virtual users. Each fetches the map, waits for the answer, pauses 2.5 seconds and repeats. With answers in 0.5 seconds, each cycle takes 3 seconds: 300 ÷ 3 = 100 requests a second.
- Halfway through, a slow database query makes each answer take 5 seconds. Each cycle now lasts 7.5 seconds, so the same 300 users send only 300 ÷ 7.5 = 40 requests a second.
- The load has more than halved just when the system is struggling, a breathing space real traffic never gives, so the report looks far better than the on-sale will.
- Real fans follow an open model: 100 a second keep arriving whatever happened to earlier requests. At 5 seconds each, Little's law gives 100 × 5 = 500 requests in progress, ten times the normal 50, and if the system cannot finish 100 a second, the queue keeps growing.
- k6's arrival-rate executors reproduce this, starting new requests at a set rate however slowly earlier ones are answered, provided the test allows enough virtual users.
- Closed tests suit a truly fixed population, such as a call-centre application used by 300 agents who send nothing while waiting.
- **The lesson:** match the model to who sends the traffic. For public sites, queue consumers and on-sales, test at a set arrival rate, or the test is most optimistic exactly where it matters most.

Coordinated omission

Gil Tene, of Azul Systems and the author of the HdrHistogram library, named a measurement error that affects many latency tests and monitoring tools: coordinated omission. A closed-loop load generator waits for each response before sending the next request. When the system stalls, the generator stalls with it, so the requests that real users would have sent during the stall, and that would all have been slow, are never sent and never measured. The tester “coordinates” with the system it is testing and omits exactly the bad samples.

Consider a test that intends to send one request every 10 ms for 100 seconds against a service that normally answers in 1 ms. At 50 seconds the service freezes for 10 seconds (a long garbage-collection pause, say). The closed-loop tester records about 9,000 fast samples and one 10-second sample, so it reports a p99 of 1 ms and even a p99.9 of 1 ms. On the intended schedule, however, about 1,000 requests would have arrived during the freeze, waiting between 10 seconds and 10 ms, about 5 seconds on average. Ten per cent of all requests were badly delayed: the true p95 is about 5 seconds and the true p99 about 9 seconds.

What the tester misses



The tester's one slow sample hides a thousand slow user experiences.

The remedies are to use open-model (arrival-rate) generators for open systems, to measure latency from the time a request was *intended* to be sent rather than when it was actually sent, to record full histograms rather than averages, and to be suspicious of any latency report whose maximum is far above its p99.9 with few samples in between. Tene's modified HTTP benchmarking tool wrk2 and HdrHistogram's correction options exist for exactly this. The same error appears in production monitoring when a client library measures only the requests it managed to send.

Tools

Four open-source tools cover most needs. **Apache JMeter** is the veteran: Java-based, with a graphical test-plan editor, many protocol plug-ins and thread groups that are closed by default. **Gatling** writes tests as code (Scala, Java and Kotlin, with JavaScript and TypeScript added more recently), is efficient at high load and has open and closed injection profiles. **k6**, maintained by Grafana Labs, runs JavaScript test scripts on an engine written in Go, has explicit open- and closed-model executors and pass-or-fail thresholds suited to continuous integration.

Locust describes user behaviour in Python and distributes load across worker processes; its users are closed-loop. Commercial and cloud-hosted services add large-scale distributed load generation and reporting.

The tool matters less than the scenario. A good script reproduces the real mix of journeys (from production analytics: perhaps 70% browsing, 20% basket changes, 10% checkout), realistic think times, realistic data (different users, restaurants and seats, not the same one 10,000 times) and the correct workload model, and it checks responses for correctness, because a fast error page is not a successful request.

```
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 });
}
```

A k6 open-model test for the on-sale seat map, ramping to 4,900 requests per second and failing the run if p99 exceeds 500 ms or more than 0.5% of requests fail.

Realistic environments and data

A performance test is only as trustworthy as its environment. A test database with 10,000 rows whose indexes fit in memory says nothing about production with 200 million rows that do not. The checklist below catches the usual differences.

Size and shape of data: production-scale volumes and realistic distributions, including hot keys such as the most popular restaurants or the best seats, which create contention a uniform test never sees. **Same building blocks:** the same instance types, container limits, database configuration, connection pool sizes and network topology (including cross-zone hops). **Cache state:** test both warm and cold caches, because a deployment or failover empties them (Section 7.6). **Dependencies:** third-party services replaced by stubs that reproduce their real latency distribution and rate limits, not instant responses. **Isolation:** no other team's tests sharing the environment, or results are noise. **The generator itself:** enough load-generator capacity in the right locations; a generator running at 100% CPU measures itself.

Some organisations test in production, carefully, because no copy is ever quite like it: synthetic users and test events that are excluded from business reporting, traffic shadowing (copying real requests to a new version and discarding its responses), and gradual ramps with automatic abort conditions. This needs the same safeguards as chaos experiments: a small blast radius, a kill switch and close watching of SLOs.

Performance budgets

A performance budget is a limit that a change may not exceed, agreed in advance and checked automatically, so that performance does not erode one small regression at a time. For web front ends, budgets cover page weight, script size and the Core Web Vitals, for which Google's thresholds for a good experience, measured at the 75th percentile of page loads, are a Largest Contentful Paint of 2.5 seconds or less, an Interaction to Next Paint of 200 ms or less and a Cumulative Layout Shift of 0.1 or less.

For back ends, a latency budget divides an end-to-end SLO between the hops of a journey. The food-delivery checkout has a p99 target of 800 ms; the team allocates 20 ms to the gateway, 60 ms to the order service, 80 ms to pricing, 450 ms to payment (most of it the external card provider), 40 ms to the database, 100 ms to network and serialisation, and keeps 50 ms in reserve. Each team now knows its own target, and a change that pushes pricing to 150 ms is visible as a budget breach before it becomes an SLO breach. Strictly, percentiles of the parts do not add up to the percentile of the whole, so treat the split as an engineering allocation and verify the end-to-end figure separately.

Budgets are enforced by running a short, stable performance test in the delivery pipeline and failing the build when a threshold is crossed, as the k6 thresholds above do. This is a fitness function for performance. Pipeline tests must be repeatable more than realistic: same environment, same data, same load, so that a 10% change means a real regression and not noise.

WORKED EXAMPLE

Worked example: a capacity plan for a ticket on-sale

This is a different, larger event from the on-sale in Section 9.3, with its own figures. A stadium tour has three nights of 60,000 seats, 180,000 tickets in all, going on sale at 10:00 on a Friday. Fans enter a virtual waiting room and are admitted to the purchase flow at a controlled rate. Load tests have measured each API instance's knee at 520 requests per second; the team plans at 400 to keep p99 under 500 ms.

Demand forecast	520,000 presale registrations, about 75% expected to turn up: plan for 400,000 fans in the waiting room. At an average of 2.5 tickets per order, 72,000 orders sell out the tour.
Binding constraint	The payment provider's agreed limit is 120 payments per second. With 80% of admitted fans buying and a third of the limit kept for retries and error, admit 100 fans per second (6,000 per minute): 80 payments per second.
Active fans (Little's law)	100 fans/s × 240 s in the purchase flow = 24,000 fans active at once
API load	100 fans/s × 25 requests per journey = 2,500 requests/s; × 1.3 for clustering of seat-map calls = 3,250 requests/s design load
Instances	3,250 / 400 = 8.1, so 9 instances; to survive losing one of three zones, 9 must fit in two zones: 5 per zone × 3 = 15 instances, started before 09:30
Seat-hold database	1.5 hold attempts per fan = 150 writes/s against a measured 600 writes/s at p99 under 50 ms: 25% utilisation, but test contention on the most popular seats
Waiting room	400,000 fans polling every 20 s = 20,000 requests/s, served at the CDN edge and never reaching the API tier
Outcome	72,000 orders / 80 per second = 900 s: sell-out in about 15 minutes, after about 90,000 fans have been admitted; the waiting room must tell everyone else promptly
Lead times	Payment limit confirmed 3 weeks ahead; cloud quota raised 1 week ahead; load test at 1.5x (about 4,900 requests/s), spike and 2-hour soak tests 2 weeks ahead in a production-sized environment

The admission rate is set by the scarcest shared resource, here the payment provider, not by how many servers the team can afford; doubling servers would not sell tickets faster.

Autoscaling is a safety net here, not the plan: a step from zero to 3,250 requests per second at 10:00:00 arrives faster than instances can start.

WATCH OUT

Pitfalls that make performance tests lie

- Closed-model generators for open traffic, and latency measured from actual rather than intended send time (coordinated omission).
- Reporting averages instead of percentiles and histograms.
- Tiny, uniform test data that fits in memory and has no hot keys.
- Stubs that answer instantly where the real dependency takes 400 ms and has a rate limit.
- Counting fast error responses as successful requests.
- A load generator that is itself the bottleneck.

CHECK YOUR UNDERSTANDING

▼ Why does a closed-model load test tend to underestimate response times for a public website?

Because its virtual users wait for each response before sending more, so when the system slows the offered load drops, whereas real visitors keep arriving and queues grow.

▼ A tester shows p99 of 2 ms and a maximum of 8 s with almost no samples between. What should you suspect?

Coordinated omission: the tester probably stalled with the system and failed to record the many requests that would have been delayed during the pause.

▼ In the on-sale example, why does adding API instances not allow a faster sell-out?

Because the payment provider's 120 payments per second limit, not the API tier, caps how many fans can buy per second.

▼ What is a soak test designed to find?

Problems that only appear over time at steady load, such as memory or connection leaks, growing queues and filling disks.

▼ Try it: A team tests its on-sale with 1,000 virtual users, each waiting for a reply before sending again, and reports a p99 of 300 ms. Real fans will arrive at 2,000 a second. Why not trust this? What would you run instead?

This is a closed workload: when the system slows, the testers slow too, so pressure eases, which real fans would not do. It also hides delay through coordinated omission, since latency is measured from the actual send, not the intended one. Run an open-model test at a fixed arrival rate, measuring from the intended send time with full histograms. Add a spike test, using realistic data, hot seats and the payment provider's limits.

SOURCES

- 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/

SECTION 9.8 OF 9.8 · INTERMEDIATE · 25 MIN READ · LAST REVIEWED NOVEMBER 2026

Chaos engineering, incidents and blameless learning

BEFORE YOU READ THIS

This section builds on:

- 2.5 Availability and reliability
- 9.4 Resilience patterns
- 9.5 SLOs and error budgets

Terms it uses from earlier: Mean time to repair (MTTR), Error budget, Service level objective (SLO), Runbook, Failover

THE IDEA IN BRIEF

Fire brigades do not wait for a real fire to find out whether the hoses work. They run drills, start small controlled fires to practise, and after every real fire they sit down together to work out what happened, without hunting for someone to blame, so that the next fire goes better. Software teams do the same. They deliberately break small parts of their systems to check the safety nets work, follow a clear plan with named roles when something really goes wrong, and afterwards study each incident honestly so the system, not just the people, gets better.

WHAT YOU NEED TO KNOW

- Chaos engineering, which grew from Netflix's Chaos Monkey (described in December 2010), experiments on a system to build confidence that it withstands turbulent conditions; it is a controlled test of a hypothesis, not random breakage.
- The Principles of Chaos Engineering are: a hypothesis about steady-state behaviour, real-world events, experiments in production, continuous automation and a minimal blast radius with abort conditions.
- Game days rehearse significant failures with the whole team and test people, access, runbooks, backups and communication as well as software.
- Incident management separates roles: an incident commander who coordinates rather than debugs, operations responders, a communications lead who owns the status page and a scribe; severity levels are defined by user impact, and mitigation comes before diagnosis.
- Blameless post-incident reviews (Google SRE book; John Allspaw, 2012) seek second stories and several contributing factors rather than a single root cause or human error.
- MTTR across many incidents is a misleading metric because durations are highly skewed; SLO impact, recurring themes and completed system changes are better guides.

- Strong action items remove hazards or add guardrails, have owners and due dates, and are prioritised through the error budget policy; reminders to be careful change nothing.

Practising failure on purpose

Everything in this chapter so far is a promise about how the system will behave when something goes wrong: timeouts will fire, circuit breakers will open, load will be shed, replicas will take over, alerts will reach someone who knows what to do. Chapter 2 warned that failover which has never been exercised often fails when it is finally needed (Section 2.5). This section is about turning those promises into evidence, and about how organisations respond to, and learn from, the failures that happen anyway.

Two practices belong together. Before failures, *chaos engineering* and *game days* deliberately cause small, controlled failures to find weaknesses and to train people. After failures, *incident management* limits the damage and *blameless post-incident reviews* turn each incident into lasting improvement. Both rest on the same belief: complex systems are always partly broken, and resilience comes from finding out how, early and safely.

Chaos engineering

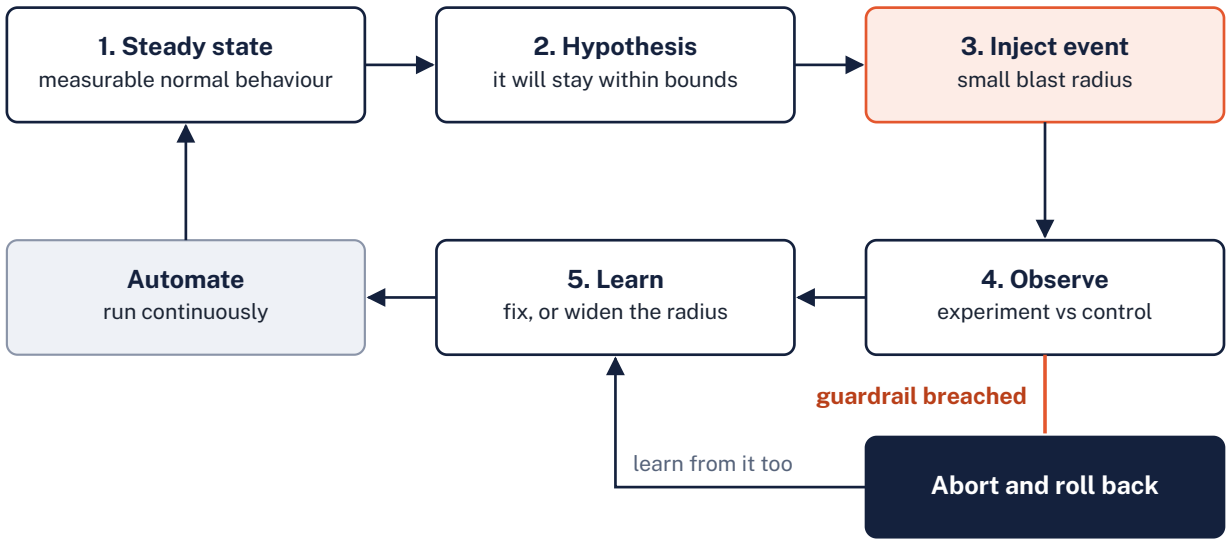
Chaos engineering grew out of Netflix's migration from its own data centres to Amazon Web Services, which began after a serious database corruption in 2008. Individual cloud instances could disappear at any time, so Netflix built Chaos Monkey, a tool that randomly terminated production instances during business hours, when engineers were at their desks to notice and fix any resulting problem. Netflix first described it publicly in a December 2010 blog post, and described a family of related tools called the Simian Army in July 2011. Later tools went further: Chaos Kong simulated the loss of an entire AWS region by evacuating traffic from it.

Netflix engineers and others later wrote down the discipline as the *Principles of Chaos Engineering* (principlesofchaos.org, last revised in 2019), which define it as experimenting on a system to build confidence in its capability to withstand turbulent conditions in production. The word that matters is *experimenting*: chaos engineering is not breaking things at random to see what happens, but the scientific method applied to resilience. The principles list five practices:

Build a hypothesis around steady-state behaviour. Choose a measurable output that indicates normal operation, such as orders placed per minute or the checkout SLI, rather than internal metrics like CPU. The steady-state hypothesis predicts that this output will stay within bounds while the failure is injected. **Vary real-world events.** Inject the failures that actually happen: instances dying, a dependency slowing down, packet loss, a full disk, an expired certificate, clock skew, a spike in traffic. **Run experiments in production.** Only production has the real traffic, data and configuration. **Automate experiments to run continuously.** A one-off

experiment proves resilience once; systems change daily. **Minimise the blast radius.** Start with the smallest experiment that can teach something (one instance, a small percentage of traffic) and keep an abort switch ready.

The chaos experiment loop



A chaos experiment is a controlled test of a hypothesis, with guardrails that stop it before users are seriously harmed.

Running experiments in production is the most debated principle. Its defenders argue that staging environments differ in traffic, data, scale and configuration, so only production experiments reveal real weaknesses, and that a small, controlled experiment at 15:00 is far kinder to users than the same failure arriving uncontrolled at 03:00. Critics point out that many organisations lack the observability, automation and organisational trust to do this safely. The practical consensus is a progression: begin in a staging environment, then run in production with a tiny blast radius and automatic abort conditions tied to SLOs, and widen only as confidence grows. Never start with a system you already know is fragile; fix the known weaknesses first, because an experiment whose outcome you can predict teaches nothing.

Tooling is now widely available: cloud providers offer managed fault injection (AWS Fault Injection Service, Azure Chaos Studio), Chaos Mesh and LitmusChaos are open-source CNCF projects for Kubernetes, and commercial platforms such as Gremlin provide libraries of attacks with safety controls. Service meshes and proxies can inject latency and errors into a percentage of calls without touching application code. The tools are the easy part; the hard part is choosing meaningful hypotheses and acting on the results.

WORKED EXAMPLE

Worked example: a latency experiment on the order service

The food-delivery order service calls the menu service to confirm item prices. The design (following Section 9.4) says the call has a 300 ms timeout and falls back to cached prices. Each order-service instance has 50 worker threads and receives about 40 orders per second, each normally holding a thread for 0.2 s.

Steady state	Order success ratio at least 99.9%; checkout p99 under 800 ms; orders per minute within 5% of the same minute last week
Hypothesis	If menu-service calls from one order-service instance take an extra 2,000 ms, that instance's steady-state metrics stay within bounds, because the timeout and fallback absorb the delay
Blast radius	1 of 20 instances (5% of order traffic) in one region, 15:00 on a weekday, 15 minutes, the other 19 instances as the control group
Abort conditions	Experiment instance's success ratio below 99% for 2 minutes, or the regional checkout SLO burn rate above 14.4: stop automatically
Expected thread use	With the 300 ms timeout: 40/s × (0.2 + 0.3) s = 20 busy threads out of 50
What happened	Success ratio fell to 97% within 3 minutes and the abort fired. Traces showed calls waiting 2.0 s: the timeout was really 3,000 ms, a library default that overrode the intended setting
Why it failed	40/s × (0.2 + 2.0) s = 88 threads needed, but only 50 exist: requests queued and timed out at the gateway
Actions	Set the timeout in the shared client configuration, add an automated test that fails the build if any outbound call lacks an explicit timeout, rerun the experiment, then widen to a whole zone

The experiment found, at 5% of one region's traffic and in 3 minutes, a weakness that a real menu-service slowdown at lunchtime would have turned into a regional outage.

The design was right; the configuration was not. Experiments test what is deployed, not what is documented.

Game days

A game day is a planned exercise in which a team deliberately causes or simulates a significant failure and practises responding to it. Jesse Robbins popularised the practice at Amazon, where his job title was reported as Master of Disaster. Google runs large Disaster Recovery

Testing (DiRT) exercises, and its SRE book describes the Wheel of Misfortune, a role-play in which on-call engineers work through a past incident while a game master reveals what the monitoring would have shown.

Game days test more than software. They test whether alerts reach the right people, whether people have the access they need at 03:00, whether runbooks are current, whether the failover procedure actually works, whether backups restore, and whether the communications plan reaches customers. A typical food-delivery game day might fail over the order database to its standby during a quiet afternoon, with the whole on-call rotation watching, a facilitator injecting surprises ("the person who knows the runbook is on leave") and an observer noting every delay and confusion for the review afterwards. Before a big ticket on-sale, the same team might rehearse losing the payment provider and switching to the backup provider mid-sale.

On-call that people can sustain

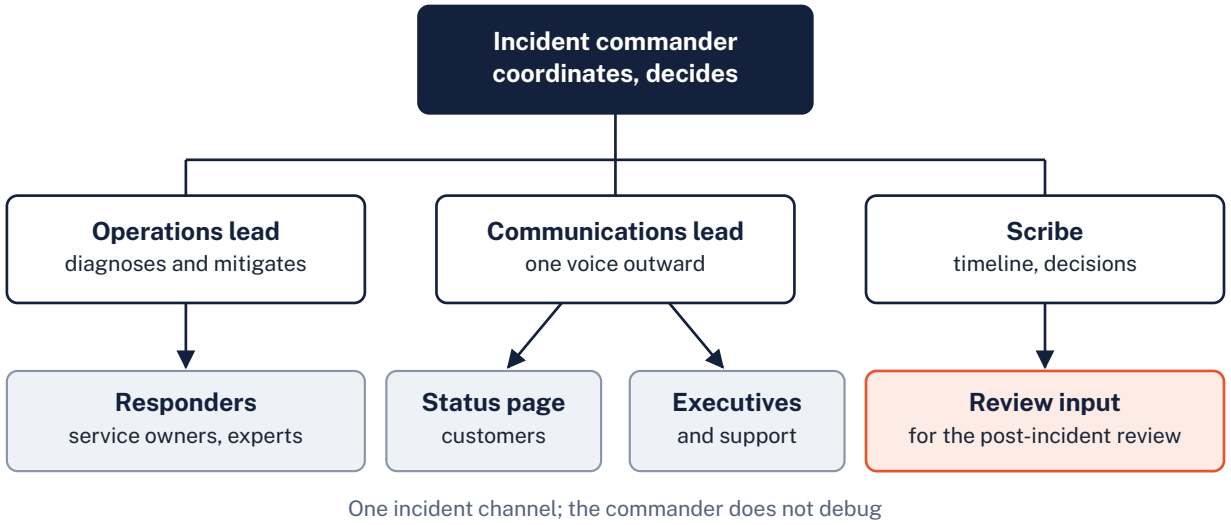
On-call is a system with a capacity limit, and burnt-out engineers are one of its failure modes. Google's SRE books give useful reference figures: at most 25% of an engineer's time on-call, a target of no more than two incidents per 12-hour shift, and a minimum of eight people for a single-site round-the-clock rota, or six per site across two sites. These are one company's numbers, not rules, but they show why a four-person team cannot sustainably cover the platform's evening peak and the night as well. Architects affect this directly, because each self-hosted component adds to someone's rota. Make every page actionable (Section 9.6), match response times to severity, treat repeat pages as defects to fix, hand over properly between shifts and compensate out-of-hours work.

Incident management

An incident is an unplanned event that disrupts, or threatens to disrupt, a service and needs a coordinated response. Incident management is the practice of running that response so that it is fast, calm and well communicated. Most modern practice borrows from the Incident Command System, developed for wildfire response in California in the 1970s, and adapted for software in the Google SRE book's chapter on managing incidents.

The central idea is separation of roles. The incident commander owns the incident: sets priorities, assigns roles, makes decisions and keeps the overview, and deliberately does *not* debug, because a commander buried in a log file is not commanding. The operations lead and responders diagnose and mitigate. The communications lead is the single voice to everyone outside the response: support teams, executives and customers through the status page, with updates at a promised cadence even when there is nothing new. A scribe records the timeline, decisions and hypotheses. In a small company one person may hold several roles, but the roles should still be named. Roles are handed over explicitly when shifts change.

Incident roles



Separating coordination, technical work and communication keeps each role focused.

Several habits make the difference. Declare incidents early and cheaply: it is better to stand down a false alarm than to discover an hour in that three teams have been debugging the same problem separately. Classify each incident with a severity level defined by user impact, not by cause, so that everyone knows who to call and how widely to communicate. Mitigate before diagnosing: roll back the last deployment, fail over, switch off the feature flag or shed load first, and find out why later. Use one incident channel. And remember external obligations: a personal-data breach may have to be reported to the regulator within 72 hours under the GDPR, which is why security incidents follow the related process in Section 8.8.

Illustrative severity levels for the food-delivery platform

Level	Example impact	Response	Communication
SEV1 (critical)	Ordering or payment unavailable for most users in a region; suspected data breach	Page immediately; incident commander and communications lead assigned at once	Status page within 15 minutes; updates every 30 minutes; executives informed
SEV2 (major)	A key journey badly degraded for many users, such as checkout p99 above 4 s in one city; fast SLO burn	Page the on-call engineer; commander assigned	Status page if customers can notice; internal updates hourly
SEV3 (minor)	Partial degradation with a workaround; an internal tool down	Handled in working hours by the owning team	Team channel; ticket
SEV4 (low)	Cosmetic issue or a near miss with no user impact	Backlog item	Optional review if it teaches something

Blameless post-incident reviews

After the incident comes the most valuable part: learning. A post-incident review (often called a postmortem) reconstructs what happened, why it made sense to the people involved at the time and what will change. The Google SRE book's chapter on postmortem culture sets out the expectations: a written record for every significant incident, covering impact, timeline, contributing causes and follow-up actions, reviewed and shared, and written without blame.

John Allspaw's 2012 post “Blameless PostMortems and a Just Culture”, written when he led technical operations at Etsy, explains why blamelessness is practical rather than merely kind. Engineers who expect punishment stop giving full accounts of what they saw and did, so the organisation loses exactly the information it needs to prevent the next incident. Allspaw draws on safety science, including Sidney Dekker's work on just culture, and on the idea of *second stories*: the first story says “an engineer ran the wrong command”; the second asks why the command was easy to run, why the tool did not warn, why the procedure seemed reasonable under time pressure, and finds conditions that will trap the next person too. Blameless does not mean unaccountable: people are still expected to give a full account, and the organisation is accountable for fixing the conditions.

Good reviews avoid hindsight bias (the outcome was not obvious at the time), counterfactuals (“should have”, “failed to”) and the search for a single root cause. Richard Cook's short paper “How Complex Systems Fail” (1998) argues that serious failures in complex systems need

several flaws to line up, so there is no single root cause, only contributing factors. A five-whys chain that ends at “human error” has stopped too early. Note also what went right: the expertise and improvisation that limited the damage are part of how the system really stays up, and they are invisible unless someone records them.

IF YOU ARE NEW TO THIS

Step by step: from who to how

The step most people trip over here is that a good post-incident review looks past the person who made the last mistake to the conditions that made it easy, because those conditions will trap the next person too. Imagine the worked example's timeout had not been caught by an experiment.

- One lunchtime the menu service slows down. The order service's calls to it wait up to 3 seconds instead of the designed 300 ms, its threads run out, and checkouts fail across the region.
- The first story is quick to write: a developer shipped the menu client with a 3-second timeout. Its action item writes itself: be more careful with configuration.
- The second story asks why the mistake was easy to make and hard to notice. The developer had written 300 ms, but a library default of 3,000 ms quietly overrode it; nothing in the build checked for explicit timeouts; and the design said 300 ms, so reviewers saw nothing wrong.
- These are contributing factors: several conditions that lined up. Any developer could walk into the same trap, so blaming this one changes nothing.
- Because the review is blameless, the developer can say openly what they configured and why, which leads the team to the library default.
- The actions, as in the worked example, change the system: set the timeout in the shared client configuration and add a build check that fails if any outbound call lacks an explicit timeout, each with an owner and a due date.
- **The lesson:** 'human error' is where a review starts, not where it ends. Ask how each action made sense at the time and fix the conditions; a reminder to be careful leaves the trap set.

Learning from incidents, and the limits of MTTR

Organisations naturally want a number to show whether incident response is improving, and mean time to recovery (MTTR) is the usual candidate. Chapter 2 used MTTR in A = MTBF / (MTBF + MTTR), where it is meaningful for a single component with repeatable failures. As an organisational metric across many different incidents it is much weaker. Incident durations are

highly skewed: most are short and a few are very long, so the mean is dominated by a handful of outliers. Courtney Nash's analyses of thousands of public incident reports in the Verica Open Incident Database (VOID reports from 2021 onwards) found exactly this skew, and a 2021 Google report by Štěpán Davidovič, *Incident Metrics in SRE*, used simulations to show that, with realistic numbers of incidents, changes in MTTR are mostly noise and cannot reliably show whether an improvement worked.

Better signals are the impact on SLOs and error budgets (which weight incidents by what users felt), recurring themes across reviews (the same kind of configuration change causing trouble three times), near misses, how quickly incidents are detected, how often reviews lead to completed system changes, and qualitative evidence such as whether engineers find reviews useful. Many organisations publish reviews internally in a searchable library and hold regular incident review meetings, so that one team's lesson reaches others.

Action items that actually change the system

A review is worth only the changes it produces. Actions differ greatly in strength. Strongest are those that remove the hazard (delete the dangerous command, make the unsafe state impossible), then guardrails that stop the mistake propagating (validation of configuration, staged rollouts and canaries, rate limits on changes, automatic rollback on SLO burn), then faster detection and mitigation (a new SLO alert, a tested runbook, an automated failover). Weakest are training, documentation and reminders. “Be more careful” is not an action item; it changes nothing about the conditions that will meet the next person.

Each action needs an owner, a priority and a due date, and must live in the team's normal backlog rather than in a forgotten document. The error budget policy from Section 9.5 provides the leverage: an incident that consumed a large share of the budget earns its fixes priority over features. Keep the list short; five completed actions beat twenty open ones. Where an action changes the architecture, such as adding a bulkhead or removing a synchronous dependency, record it in an architecture decision record that cites the incident, and link the relevant runbooks from the alerts that will fire next time.

Checklist: running a post-incident review

Use this for every incident above an agreed severity, for every incident that spends more than a set share of an error budget, and for interesting near misses.

- [ ] **Schedule it within a few working days**
  Memories fade fast; logs and chat history may expire. Book the people involved, a facilitator who was not involved, and the owners of affected services.
- [ ] **Build the timeline from evidence first**
  Assemble alerts, deployments, chat messages, dashboards and traces into a timeline before the meeting, so the discussion starts from facts rather than recollection.
- [ ] **State the impact in user terms**
  Who was affected, how many, for how long, which SLOs and how much error budget; include money, data and regulatory exposure where relevant.
- [ ] **Set the blameless frame explicitly**
  Open by saying that the goal is to understand how the system and the work really happen, and that nobody will be punished for describing what they did and why.
- [ ] **Ask how it made sense at the time**
  For each key decision, ask what the person knew, what they were trying to do and what the tools showed them. Avoid counterfactuals such as 'should have' and 'failed to'.
- [ ] **Look for several contributing factors**
  Technical, organisational and procedural conditions together; resist stopping at the first single 'root cause' or at human error.
- [ ] **Note what went well**
  Record the adaptations, expertise and luck that limited the damage; they are part of how the system really stays up.
- [ ] **Agree a few actions that change the system**
  Prefer removing hazards and adding guardrails over training and reminders; give each action an owner, a priority and a due date, and track it in the normal backlog.
- [ ] **Publish and share**
  Write the review up in a standard template, share it widely, add it to a searchable library and revisit open actions at a regular review meeting.

CHECK YOUR UNDERSTANDING

▼ Why does a chaos experiment need a steady-state hypothesis rather than simply 'see what breaks'?

Because a measurable prediction about normal user-facing behaviour tells you whether the system coped, gives abort conditions and turns the exercise into evidence rather than an anecdote.

▼ Why should the incident commander avoid debugging?

Because the role is coordination: priorities, roles, decisions and overview. A commander absorbed in technical detail leaves nobody steering the response.

▼ What is a 'second story' in a post-incident review?

The account that goes beyond who made a mistake to the conditions, tools, pressures and design that made the action seem reasonable and will trap the next person too.

▼ Why is average MTTR across a year of incidents a weak measure of improvement?

Incident durations are highly skewed, so a few long incidents dominate the mean, and with realistic incident counts changes in it are mostly noise.

▼ Try it: Each order instance has 30 threads and gets 20 orders a second, each normally holding a thread for 0.2 s. The menu call should time out after 300 ms and use cached prices. You inject 5 s of latency. Predict the results.

Normally, 20 x 0.2 = 4 threads are busy. With the timeout, each order holds a thread for about 0.5 s, so 10 are busy, and orders survive on cached prices. If the timeout is missing, each holds a thread for about 5.2 s, needing 104 threads against 30, so the instance exhausts them. Start with one instance, set abort conditions on the order success rate, and fix any weakness found.

SOURCES

- 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/
- Being On-Call (Site Reliability Engineering, chapter 11): https://sre.google/sre-book/being-on-call/
- On-Call (The Site Reliability Workbook, chapter 8): https://sre.google/workbook/on-call/

Further reading

- *Release It! Design and Deploy Production-Ready Software (2nd edition)*, Michael T. Nygard (2018), Pragmatic Bookshelf. The book that named the circuit breaker and bulkhead patterns; its stability patterns and anti-patterns, illustrated with real outages, are the basis of Section 9.4.
- *The Art of Scalability (2nd edition)*, Martin L. Abbott and Michael T. Fisher (2015), Addison-Wesley. Introduces the scale cube and connects technical scaling choices to organisation and process.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (editors) (2016), O'Reilly Media. Free online; the source of SLOs and error budgets as practised today, and of excellent chapters on overload, cascading failures and 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. The practical companion: implementing SLOs, multiwindow burn-rate alerting and on-call, with worked examples.
  https://sre.google/workbook/table-of-contents/
- *Exponential Backoff And Jitter*, Marc Brooker (2015), AWS Architecture Blog. A short simulation-based article showing why retries need randomised delays; the origin of the widely used full-jitter formula.
  https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/
- *Systems Performance: Enterprise and the Cloud (2nd edition)*, Brendan Gregg (2020), Addison-Wesley. The standard reference on performance analysis, including the USE method, observability tools and benchmarking pitfalls.
- *Chaos Engineering: System Resiliency in Practice*, Casey Rosenthal and Nora Jones (2020), O'Reilly Media. Written by practitioners from Netflix and elsewhere; explains the principles of chaos engineering and how organisations adopt it.
- *Fail at Scale: Reliability in the face of rapid change*, Ben Maurer (2015), ACM Queue, vol. 13, no. 8. How Facebook handles overload and slow dependencies with controlled-delay queues, adaptive LIFO and client-side concurrency limits.
  https://queue.acm.org/detail.cfm?id=2839461

Glossary

Anycast
Announcing the same IP address from many locations, so that the internet's routing delivers each user's packets to the nearest location announcing it.

Autoscaling
Automatically changing the number of instances as load changes, reactively from a metric, on a schedule, or predictively from a forecast.

Blameless culture
An approach in which people who were involved in an incident can describe what they did and why without fear of punishment, so the organisation learns how the system really works.

Blast radius
How much of the system and how many users a failure, change or experiment can affect; chaos experiments deliberately keep it small.

Breakpoint test
A performance test that increases load steadily until the system breaks, to find its maximum capacity and its first bottleneck.

Bulkhead
Partitioning resources such as thread pools, connection pools or instances, so that a failure or overload in one part cannot consume the resources of the others.

Burn rate
How fast an error budget is being spent relative to the rate that would use it up exactly at the end of the window; a burn rate of 1 spends it all in exactly one window.

Capacity planning
Forecasting future demand, measuring how much each unit of infrastructure can handle and arranging enough capacity, with headroom, before it is needed.

Cardinality
The number of distinct values a field can take; for metrics, the number of distinct label combinations, each of which becomes a separate time series to store.

Chaos engineering
Experimenting on a system, ideally in production and under control, by injecting realistic failures to build confidence that it withstands turbulent conditions.

Circuit breaker
A wrapper around calls to a dependency that, after enough failures, stops calling it for a while and fails fast, then lets trial calls through to test recovery (Michael Nygard, 2007).

Closed workload model
A load model with a fixed number of users, each sending a new request only after the previous one is answered (plus think time), so load falls when the system slows.

Competing consumers
A pattern in which several workers read from the same queue and each message is processed by only one of them, so throughput grows by adding workers.

Connection draining
Stopping new requests to an instance that is being removed while letting the requests already in progress finish, so that scaling in and deployments drop no work.

Context propagation
Passing trace identifiers (and other request context) from one service to the next with each call or message, for example in the W3C traceparent header.

Contributing factor
One of the several conditions that together made an incident possible or worse; reviews look for many contributing factors rather than a single root cause.

Cooldown (stabilisation window)
A period after a scaling action during which the autoscaler waits before acting again, so that the effect of the last change can be measured.

Coordinated omission
A measurement error, named by Gil Tene, in which a load generator waits for slow responses before sending more requests and so never records the delays real users would have suffered.

Counter
A metric whose value only goes up (or resets to zero on restart), such as total requests served; you usually look at its rate of increase.

Deadline propagation
Passing the time remaining for a whole request to every downstream call, so that no service keeps working on a request its caller has already abandoned.

Distributed tracing
Recording the path and timing of one request as it travels through many services, so that the slow or failing step can be found.

Error budget
The amount of unreliability an SLO allows: one minus the target, applied to the window. A 99.9% SLO over 1,000,000 requests allows 1,000 bad requests.

Error budget policy
An agreed, written statement of what a team will do when its error budget is being spent too fast or is exhausted, such as freezing risky releases.

Exponential backoff
Waiting longer between successive retries, typically doubling the delay each time up to a cap, so that a struggling service gets time to recover.

Flapping
An autoscaler repeatedly adding and removing capacity because it reacts to short-lived changes or to the effects of its own actions.

Four golden signals
Latency, traffic, errors and saturation: the four measurements the Google SRE book recommends monitoring for any user-facing system.

Game day
A planned exercise in which a team deliberately causes or simulates a failure and practises detecting, responding to and recovering from it.

Gauge
A metric that records a current value that can go up or down, such as queue length, memory in use or temperature.

Global load balancing
Distributing users across data centres or regions, usually by DNS answers that differ by location and health, or by anycast routing.

Headroom
Spare capacity kept deliberately above expected peak load, to absorb bursts, failures and the time it takes to add more capacity.

Health check
A periodic test, such as an HTTP request to a status endpoint, that a load balancer or orchestrator uses to decide whether an instance should receive traffic or be restarted.

Hedged request
Sending a second copy of a slow request to another replica after a short delay and using whichever answer arrives first, to cut tail latency.

Histogram
A metric that counts observations into value ranges called buckets, such as request durations, so that percentiles can be estimated and histograms from many servers can be added together.

Incident commander
The person who coordinates the response to an incident: sets priorities, assigns roles and makes decisions, rather than fixing things personally.

Jitter
Randomness added to retry delays or schedules so that many clients do not act at the same moment and create synchronised spikes.

Layer 4 load balancer
A load balancer that works at the transport level (TCP or UDP), forwarding connections or packets without reading the HTTP request inside them.

Layer 7 load balancer
A load balancer that understands the application protocol, usually HTTP, so it can route each request by path, header or cookie, terminate TLS and retry.

Leaky bucket
A rate-limiting or traffic-shaping algorithm in which requests enter a bucket and leave at a constant rate; when the bucket is full, new requests are rejected.

Least connections
A load-balancing algorithm that sends each new request to the instance with the fewest requests or connections currently in progress.

Liveness probe
A health check that asks whether a process is stuck beyond recovery; failing it causes the orchestrator to restart the instance.

Load shedding
Deliberately rejecting some requests, cheaply and early, when a system is overloaded, so that the requests it accepts still succeed in time.

Load test
A performance test that applies the expected (or forecast peak) load for a period to check that latency and error targets are met.

Monitoring
Collecting and watching predefined measurements and checks so that known kinds of failure are detected and reported.

Multiwindow, multi-burn-rate alert
An SLO alert that fires only when the burn rate is above a threshold over both a long window and a short window, combining fast detection with quick reset and few false alarms.

Open workload model
A load model in which new requests arrive at a set rate regardless of how quickly earlier ones are answered, like customers arriving at a website.

OpenTelemetry
A vendor-neutral, open-source CNCF project that provides APIs, SDKs, a protocol (OTLP) and a collector for producing and exporting traces, metrics and logs.

Performance budget
An agreed limit on a performance measure, such as page weight or p99 latency per service call, that a change must not exceed, ideally checked automatically.

Post-incident review
A structured meeting and written report after an incident that reconstructs what happened, why it made sense to the people involved and what will change; also called a postmortem.

Power of two random choices
A load-balancing method that picks two instances at random and sends the request to the less loaded one; Mitzenmacher showed it balances almost as well as checking every instance.

Rate limiting
Restricting how many requests a client, key or tenant may make in a period, to protect capacity and share it fairly; excess requests are usually answered with HTTP 429.

Readiness probe
A health check that asks whether an instance can serve traffic right now; failing it removes the instance from the load balancer without restarting it.

RED method
Tom Wilkie's checklist for monitoring request-driven services: Rate, Errors and Duration of requests.

Retry budget
A limit on retries expressed as a share of normal traffic (for example 10%), so that retries help with occasional failures but cannot multiply load during an outage.

Retry storm
A surge of load created when many clients, often at several layers, retry failing requests at once, which keeps an overloaded service overloaded.

Rolling window
A measurement period that moves forward continuously, such as the last 28 days, as opposed to a calendar month that resets on the first.

Service credit
A refund or discount owed to a customer under an SLA when the provider misses a promised service level.

Severity level
A predefined classification of how bad an incident is, such as SEV1 to SEV4, that decides who is called, how fast and how widely it is communicated.

Soak test
A long performance test at normal load, run for hours or days, to reveal slow problems such as memory leaks or growing queues; also called an endurance test.

Span
One timed operation within a trace, such as an HTTP call or a database query, with a name, start time, duration, attributes and a link to its parent span.

Spike test
A performance test that raises load suddenly, for example tenfold in seconds, to check autoscaling, queues and protection mechanisms.

Stateless service
A service instance that keeps no data between requests in its own memory or disk, so any instance can handle any request and instances can be added or removed freely.

Status page
A public or customer-facing page that reports current incidents and service health, updated by the incident's communications lead.

Steady-state hypothesis
In a chaos experiment, the prediction that a measurable sign of normal behaviour, such as orders per minute, will stay within bounds while a failure is injected.

Sticky session (session affinity)
A load-balancer setting that sends all requests from one client to the same instance, usually so that session data held in that instance's memory can be found.

Stress test
A performance test that pushes load beyond the expected peak to see how the system fails and whether it recovers.

Structured logging
Writing log entries as machine-readable key-value records (often JSON) with consistent field names, instead of free-text sentences.

Telemetry
Data a running system emits about its own behaviour: logs, metrics, traces, profiles and events.

Timeout
The maximum time a caller will wait for an operation before giving up and treating it as failed; every remote call needs one.

Token bucket
A rate-limiting algorithm in which tokens are added to a bucket at a fixed rate up to a capacity, and each request spends a token; it allows bursts up to the capacity and a steady average rate.

Trace sampling
Keeping only some traces to control cost: head-based sampling decides when a request starts, tail-based sampling decides after it ends, so it can keep slow and failed requests.

USE method
Brendan Gregg's checklist for analysing resources: for every resource, check Utilisation, Saturation and Errors.

Virtual user
A simulated user in a load-testing tool that runs a script of requests, usually in a loop.