

BE SOFTWARE ARCHITECTURE WISE · CHAPTER 10

Cloud-Native and Infrastructure Architecture

Where and how software runs is now an architectural decision. This chapter covers cloud service and deployment models and the shared responsibility model, containers and Kubernetes, the twelve-factor app, infrastructure as code and GitOps, CI/CD pipelines and deployment strategies, multi-region design, disaster recovery and edge computing, platform engineering, and FinOps and vendor lock-in. It is for developers and architects who know the operational qualities of Chapter 9 and want to choose, justify and run the platforms their systems depend on, without being sold a stack.

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

AT A GLANCE

You will learn to

- Compare IaaS, CaaS, PaaS, FaaS and SaaS and the deployment models, and assign duties for a workload under the shared responsibility model.
- Explain how Kubernetes maintains a declared desired state through its control plane, nodes and reconciliation loops, configure requests, limits and probes, and judge when Kubernetes is overkill.
- Audit an application against the twelve factors and the principles beyond them, and prioritise the fixes by risk.
- Design infrastructure code with remote state, locking, modules, environments, policy checks, tests and GitOps reconciliation.
- Choose a deployment strategy (rolling, blue-green, canary or shadow) and separate deployment from release with feature flags.
- Select a disaster recovery strategy from RTO, RPO and cost, and explain the consequences of replicating data across regions and moving work to the edge.
- Explain how an internal developer platform with golden paths reduces cognitive load, and how to measure whether it succeeds.
- Calculate unit costs such as cost per order, identify the architectural drivers of cloud cost and evaluate lock-in as a priced trade-off.

Before you start

Chapter 0, section 0.7 (servers and the cloud) and section 0.8 (how software gets built); Chapter 9, especially scaling, statelessness, SLOs and observability. Chapter 6 on distributed systems helps.

Time

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

Key terms

cloud native · shared responsibility model · availability zone · container image · Kubernetes · reconciliation loop · twelve-factor app · infrastructure as code · GitOps · canary release · RTO and RPO · golden path · FinOps · lock-in

Why infrastructure has become architecture

For most of the history of software, infrastructure was something that happened after the architecture: an operations team received a finished application and found servers to run it on. That separation has gone. In a modern system, the choice between virtual machines, containers, platforms and functions decides much of the cost; the number of availability zones and regions decides the availability; the deployment pipeline decides how quickly and safely a team can change anything; and the infrastructure itself is written as code, reviewed in pull requests and versioned with the application. Decisions about where and how software runs are

now among the most significant an architect makes, in exactly the sense of Chapter 1: they are expensive to reverse, they cut across every team and they fix the quality attributes that Chapter 2 asked you to specify.

This chapter builds on two earlier ones. Chapter 0 introduced servers, virtual machines, containers, cloud service models, regions and availability zones (servers and the cloud). Chapter 9 explained how to make systems scale, degrade gracefully and stay observable (scaling strategies onwards). Here those ideas meet the platforms that most systems now run on. The running example is the food-delivery platform used throughout the course: customer and restaurant apps, an order service, menus, payments, courier dispatch and a sharp evening rush.

How the sections fit together

The first four sections are about the foundations. Cloud service and deployment models sets out who manages what under IaaS, containers as a service, PaaS, FaaS and SaaS, the shared responsibility model, the real reasons for private, hybrid and multi-cloud, what the CNCF means by cloud native, and when renting is not cheaper than owning. Containers and orchestration explains images and the OCI standards, the Kubernetes control plane and nodes, its core objects and the reconciliation loop at its heart, with real YAML, and is honest about when Kubernetes is overkill. The twelve-factor app walks through all twelve factors with modern caveats, the principles beyond them, and an audit of an existing service. Infrastructure as code covers declarative tools (Terraform, OpenTofu, CloudFormation, Bicep, Pulumi), state, drift, modules, immutable infrastructure, GitOps, policy as code and testing.

The second four sections are about change, scale, people and money. CI/CD pipelines and deployment strategies follows a change from commit to production, compares rolling, blue-green, canary and shadow deployments, and separates deployment from release with feature flags. Multi-region, disaster recovery and edge computing compares disaster recovery strategies by RTO, RPO and cost, explains the consistency consequences of replicating data across regions, and covers cell-based architecture and edge functions. Platform engineering treats the internal developer platform as a product with golden paths and guardrails, and shows how to measure whether it helps. FinOps, cost architecture and lock-in turns cost into an architectural quality, measured per order, and treats vendor lock-in as a trade-off to be priced rather than a sin to be avoided.

Each section can be read on its own, but the order is deliberate: you cannot reason about deployment strategies without knowing what Kubernetes does during a rollout, or about platform engineering without infrastructure code and pipelines to build the platform from.

One idea runs through every section

The single most important idea in this chapter is **declared desired state, continuously reconciled by automation**. Kubernetes keeps the number of running copies equal to the number you declared. Infrastructure code describes networks and databases, and a tool computes and applies the difference from what exists. GitOps keeps the cluster equal to what is in a Git repository. A deployment pipeline turns every change into the same tested, repeatable sequence of steps. Immutable infrastructure replaces rather than repairs. In each case a person changes a description, which is reviewed, versioned and auditable, and machines do the rest. This is what lets a small team operate a large system safely: change becomes routine, and recovery becomes a rerun rather than an act of heroism.

The second idea is that every option on this chapter's menu is a trade-off, and the right choice depends on the workload and the organisation. Kubernetes is excellent for many services and many teams and wasteful for three services and two developers. Multi-region active-active buys minutes of recovery time at a multiple of the cost. A platform team helps only if developers want what it builds. Cloud is cheapest for variable load and can be beaten by owned hardware for large, steady load. Lock-in is a cost to be weighed against the cost of portability. The chapter therefore gives you decision guides, worked examples with numbers and the arguments on both sides, rather than a single recommended stack.

How to use this chapter

Readers new to operations should read the beginner boxes and Chapter 0's section on servers and the cloud first, and should not worry about memorising tool names: products change every few years, while the mechanisms (reconciliation, immutability, state and drift, progressive delivery, recovery objectives, unit costs) last. Experienced engineers may want to start with the worked examples and the decision guides, then read the sections whose trade-offs they disagree with. Throughout, vendors such as AWS, Microsoft Azure, Google Cloud and Cloudflare are named as examples, not recommendations.

Chapter 11 follows with evaluation, governance and modernisation: how to judge an architecture, including its infrastructure, and how to change it safely over time.

SECTION 10.1 OF 10.8 · INTERMEDIATE · 34 MIN READ · LAST REVIEWED NOVEMBER 2026

Cloud service and deployment models

BEFORE YOU READ THIS

This section builds on:

- 0.7 Servers and the cloud
- 2.5 Availability and reliability

Terms it uses from earlier: Virtual machine (VM), Container, Elasticity, Failover

THE IDEA IN BRIEF

Think of how people get somewhere to live. You can buy land and build a house, rent an empty flat and furnish it, rent a furnished flat, or stay in a hotel. Each step hands more of the work to someone else and leaves you fewer choices; the hotel is wonderful for a week and expensive for ten years. Running software is the same: you can own servers, rent bare machines, rent a ready platform, or rent finished software. Whatever you choose, you still lock your own door: your data and who can reach it stay your job.

WHAT YOU NEED TO KNOW

- NIST SP 800-145 defines three service models (IaaS, PaaS, SaaS) and four deployment models (public, private, community, hybrid); containers and functions as a service fill the gaps between them.
- Each step from IaaS towards SaaS removes operational work and control; the useful question is which concrete tasks (patching, scaling, backups, failover, upgrades) remain yours.
- Under the shared responsibility model the provider secures the cloud itself and the customer secures what it puts in it; data, identities and configuration are always the customer's.
- Private, hybrid and multi-cloud have legitimate reasons (regulation, steady load, migrations, distinctive services, exit plans), but running every workload portably across clouds is costly.
- Spread workloads across at least two, preferably three, availability zones; know whether each dependency is zonal, regional or global.
- The CNCF defines cloud native as practices for building loosely coupled, secure, resilient, manageable, sustainable and observable systems programmatically and repeatably; it is about how, not where.

- The cloud is cheapest for variable load; for large, steady load owned hardware can be cheaper, as 37signals argued, once headroom, staff and commitment discounts are honestly counted.

From one definition to many models

Chapter 0 introduced servers and the cloud: the NIST definition (Special Publication 800-145, 2011), with its five essential characteristics of on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service, and the idea that service models hand more or less of the stack to a provider. This section takes those ideas to the depth an architect needs to choose where a system runs and to defend the choice.

NIST defines three service models (infrastructure, platform and software as a service) and four deployment models: **public** (open to anyone, owned by a provider), **private** (for one organisation, wherever it is located and whoever operates it), **community** (shared by organisations with common concerns, such as government bodies) and **hybrid** (two or more of these, bound together so that data and applications can move between them). The industry has since added functions and containers as a service to the service models, and multi-cloud to the deployment vocabulary.

Where software runs is part of what Chapter 1 called the allocation structure of an architecture (defining architecture): the mapping of software onto hardware, networks and organisations. It is significant in the Chapter 1 sense because it is expensive to reverse, and because it fixes much of a system's cost, its availability, its security boundary and how quickly its teams can release.

Service models: who manages what

The service models form a spectrum. At one end you rent bare capacity and do everything above it yourself; at the other you rent a finished application and only configure it. Each step to the right removes operational work and removes choices. The table adds the two models that the NIST document predates: containers as a service and functions as a service.

Service models compared

Model	You deploy and run	The provider runs	Examples	Typical fit
IaaS	Virtual machines: operating system, patches, runtime, application, scaling rules	Hardware, network, hypervisor, data centre	Amazon EC2, Azure Virtual Machines, Google Compute Engine	Software that needs a particular operating system or kernel, legacy applications moved as they are, full control
CaaS (containers as a service)	Container images and their configuration; with managed Kubernetes, also worker-node choices and upgrades	Container control plane, often the nodes too; with serverless containers, all servers	Amazon EKS and ECS, Azure Kubernetes Service and Container Apps, Google Kubernetes Engine and Cloud Run	Many services built as containers, teams that want portable packaging
PaaS	Application code and its configuration	Operating system, runtime, scaling, load balancing, often logging	Heroku, Azure App Service, Google App Engine, AWS Elastic Beanstalk	Standard web applications and APIs that fit the platform's conventions
FaaS	Individual functions and the events that trigger them	Everything else, including scaling to zero; billing per invocation and duration	AWS Lambda, Azure Functions, Google Cloud Run functions, Cloudflare Workers	Event-driven, spiky or low-volume work in short steps
SaaS	Configuration, users, permissions and your data	The whole application	Microsoft 365, Salesforce, Workday, Zendesk	Generic capabilities that do not differentiate the business

The boundaries blur in practice. Serverless container platforms such as Google Cloud Run and Azure Container Apps run a container image like a PaaS runs code, scaling to zero like a function. Managed Kubernetes can be anything from a hosted control plane, where you still patch and upgrade the worker machines, to a fully managed mode where the provider runs the

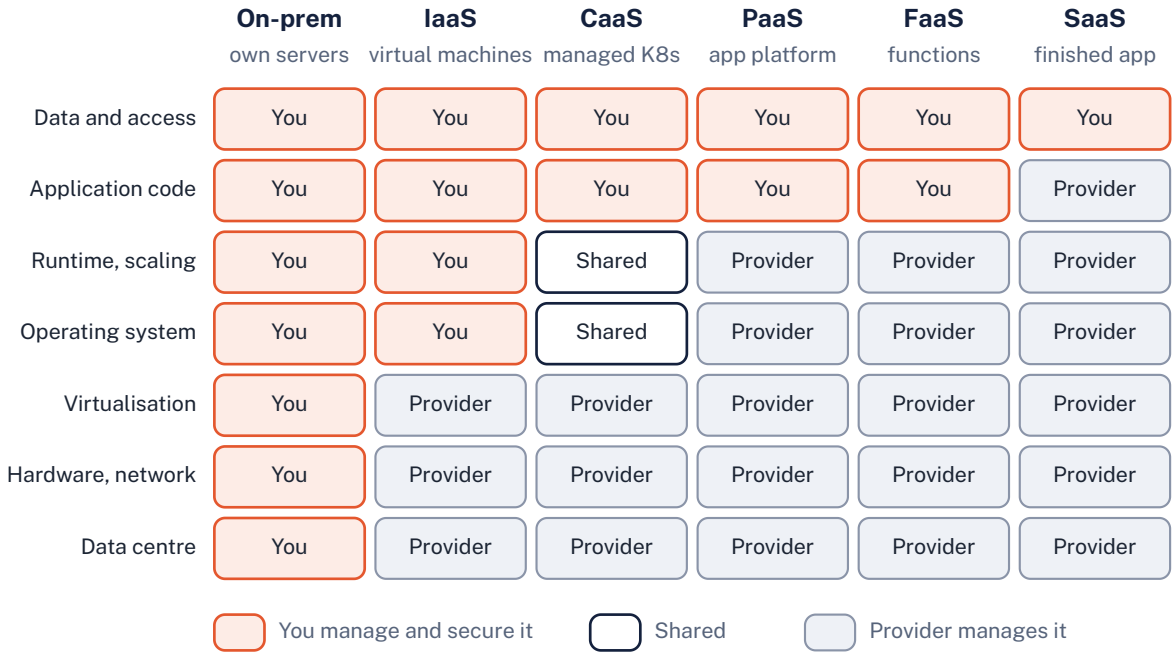
nodes too. The useful question is never which label applies but which concrete tasks remain yours: patching, scaling decisions, capacity planning, backups, failover, upgrades and on-call response.

A second useful distinction is the unit of deployment and billing. IaaS bills machines by the second or hour whether they are busy or not; FaaS bills invocations and execution time and costs nothing when idle; PaaS and serverless containers sit in between. Chapter 6 explains why per-request pricing is cheaper than always-on capacity below a crossover volume and more expensive above it (serverless architectures). The shape of a workload's load, not its size, decides which side of the crossover it is on.

The shared responsibility model

Every major provider publishes a shared responsibility model. AWS summarises it as the provider being responsible for security *of* the cloud and the customer for security *in* the cloud. The provider secures buildings, hardware, the network, the hypervisor and the services it runs; the customer secures what it builds and configures on top. Where the line falls depends on the service model, as the diagram shows.

Who is responsible for what, by service model



Moving right hands lower layers to the provider, but data, identities and access configuration stay with the customer in every model.

Three things never transfer. The customer always owns its data (what is stored, where, for how long, who can read it), its identities and permissions, and the configuration of the services it uses. Many cloud breaches have been customer configuration failures rather than provider failures: object storage left readable by anyone, administrator keys committed to public repositories, overly broad permissions. In 2019, for example, an attacker exploited a misconfigured web application firewall at Capital One to obtain cloud credentials and copy data relating to more than 100 million people in the United States and Canada; the provider's infrastructure behaved as designed.

For an architect, the model is a checklist of work that must be assigned to someone. If a team moves the food-delivery platform's orders database from a virtual machine to a managed database service, patching the database engine moves to the provider, but choosing a maintenance window, testing major version upgrades, granting least-privilege access, enabling encryption and checking that backups can be restored do not. Chapter 8 covers the security principles behind these duties in depth (security principles for architects).

Deployment models and the real reasons behind them

Organisations rarely choose a deployment model for one reason, and the stated reason is not always the real one. It helps to separate the motives that survive scrutiny from the ones that do not.

Public, private, hybrid and multi-cloud

1. **Public cloud**
   Chosen for elasticity, speed (a new environment in minutes, not a hardware purchase in months), a large catalogue of managed services and global reach. It suits variable load, new products whose demand is unknown and teams that want to spend their effort on the product rather than on servers.

2. **Private cloud**
   Chosen for regulatory or contractual control, predictable steady load, existing data centre investment or very low latency to equipment on site. A private cloud means self-service, API-driven provisioning on dedicated infrastructure (for example with OpenStack, VMware or a vendor's on-premises edition of its public cloud), not merely a virtualised server room where requests still go through a ticket queue.

3 **Hybrid cloud**

Usually the result of a gradual migration, data that must stay on premises (a mainframe ledger, factory or hospital systems), or edge sites such as shops and warehouses. Bursting from a private data centre into the public cloud at peak times is often promised but rarely practical, because the data the workload needs stays behind and copying it costs time and money.

4 **Multi-cloud**

Using two or more public providers. Defensible reasons include using a distinctive service from each provider, acquisitions that arrive on another cloud, negotiating leverage and regulation that demands credible exit plans. In the EU, the Digital Operational Resilience Act (DORA), which applies from January 2025, requires financial entities to have exit strategies for critical ICT providers. Running every workload portably across several clouds is a different and far more expensive goal.

The common error is to adopt multi-cloud as an abstract hedge. Making one workload run identically on two providers means using only the services both offer, building and testing two sets of infrastructure code, staffing skills for both, and often paying data transfer charges between them. That is the lowest-common-denominator trap: you pay the cost of portability every day in exchange for an exit you may never use. Section 10.8 treats lock-in as a trade-off to be priced, not a sin to be avoided at any cost (FinOps, cost architecture and vendor lock-in).

Regions and availability zones as design inputs

A region is a geographic area with its own independent set of data centres; an availability zone is one or more data centres within a region with separate power, cooling and networking, placed far enough from the others that a fire, flood or local power failure affects only one, and close enough for round trips of a few milliseconds at most. Chapter 0 explained the idea. For design, three further points matter.

First, every managed service has a failure scope: *zonal* resources (a single virtual machine, a disk) die with their zone; *regional* services (most managed databases in multi-zone mode, object storage, queues) are replicated across zones by the provider; a few *global* services (DNS, identity, some content delivery and load-balancing services) span regions, which makes their rare outages wide. Knowing which scope each dependency has is the first step in any availability analysis (see availability and reliability).

Second, spreading across zones is the cheap, high-value default. Running the food-delivery platform's stateless services in three zones behind a load balancer, with a database that keeps a synchronous standby in a second zone, survives the loss of a data centre with automatic

failover and a small latency cost. Three zones rather than two matter for anything that uses quorums, such as etcd in Kubernetes or a consensus-based database, because a majority must survive. Cross-zone traffic is not always free, though: most large providers charge for data moving between zones, so chatty services spread across zones add a visible line to the bill.

Third, regions are not identical. Newer or smaller regions may lack services, some have fewer zones, prices differ by region, and legal jurisdiction follows the region. Section 10.6 deals with multiple regions, disaster recovery and data residency (multi-region, disaster recovery and edge computing).

IF YOU ARE NEW TO THIS

Step by step: what stops when a zone fails

This section's hardest idea is failure scope: each part of a system lives in one zone, across a region or across the world, and that decides what stops when something fails. Follow one zone failure through the food-delivery platform.

- The platform runs its stateless services in three availability zones behind a load balancer, with a managed orders database that keeps a synchronous standby in a second zone. Customers find it through DNS.
- Each piece has a scope. A single virtual machine is zonal: it lives in one zone. The multi-zone database is regional: the provider keeps it copied across zones. DNS is global: it spans regions.
- A flood takes one zone offline during the evening rush. The service copies in that zone stop with it. The load balancer stops sending them requests, and the copies in the other two zones carry on, scaling up if they need to.
- If the database's primary copy was in the flooded zone, the provider switches to the standby after a short pause: an automatic failover. Because the standby is synchronous, each write reached it before the database confirmed it, as Chapter 7 explained, so confirmed orders survive. Orders caught mid-way during the pause may fail and need to be retried.
- Global services fail differently. Their outages are rare, but spreading across zones, or even regions, does not protect you from one, because every zone and region depends on the same service.
- **The lesson:** know each dependency's scope before you trust the design. Zonal pieces need copies in more than one zone; regional ones survive the loss of a zone but not of their whole region; global ones fail rarely but everywhere at once.

Managed services versus self-managed

A managed service is a component (a database, queue, cache, search engine, Kubernetes control plane) that the provider installs, operates, patches, backs up and scales for you. The alternative is to run the same open-source software yourself on virtual machines or in containers. The trade-off is labour and risk against control, cost and portability.

Managed services win when the component is undifferentiated: every food-delivery company needs a reliable PostgreSQL database, but none wins customers because of how it applies database patches. In domain-driven design terms these are generic subdomains, the ones you should buy rather than build. A managed database gives multi-zone failover, point-in-time recovery and minor-version patching that would otherwise need a skilled team on call. Self-management wins when you need a version, extension or tuning parameter the provider does not offer, when the provider's limits or quotas do not fit, when the managed premium becomes large at scale, or when portability across environments matters more than convenience.

Managed does not mean maintenance-free. The provider will force major version upgrades on its own schedule when old versions reach end of support, will change defaults, will impose quotas you discover at the worst moment and will have outages you cannot fix, only wait out. You still own schema design, query performance, capacity choices, cost and the regular test that proves a restore works. Treat each managed service as a dependency with its own failure modes and its own exit cost.

Cloud foundations: landing zones and the account hierarchy

Before the first workload moves in, a cloud estate needs foundations, and they are much easier to lay early than to retrofit. A **landing zone** is the prepared environment that workloads deploy into. AWS describes it as a well-architected, multi-account environment that is scalable and secure. Microsoft's Azure landing zone separates a *platform landing zone*, the central foundation of governance, security and shared resources, from *workload landing zones*, where product teams deploy within those guardrails. Either way it should be built and changed as code (Section 10.4), and it is often the first thing a platform team owns (Section 10.7).

The decision that is hardest to change later is the hierarchy. Each provider has a basic unit of isolation and a tree that groups those units: AWS accounts in organisational units; Azure subscriptions under management groups; Google Cloud projects, optionally in folders, under an organisation. The tree matters because policy attaches to it. Google Cloud policies are inherited down the tree, and Azure subscriptions inherit the policies set on their management groups. AWS service control policies set the maximum permissions available in member accounts; they grant nothing themselves, and they do not affect the management account, which is one reason to keep workloads out of it.

An account or subscription is a far stronger boundary than a tag or a naming convention. In AWS, one account has no access to another by default, quotas apply per account and costs map directly to it. So separate by workload and by environment from the start, because moving a live system between accounts later is a migration project. AWS charges for the resources used, not for the number of accounts, although many accounts do need automation.

For the food-delivery platform, a sensible starting tree, adapted from AWS's sample structure, has four branches. *Security* holds a log archive account that product teams cannot alter and an audit account. *Infrastructure* holds networking and shared services. *Workloads* holds separate production and non-production accounts for ordering, dispatch and payments, with stricter policies on the payments branch. *Sandbox* is for safe experiments. Guardrails at the top of the tree might deny regions outside those approved and stop anyone switching off logging.

Azure's list of design areas makes a useful checklist even on other clouds: billing and tenancy, identity, resource organisation, network, security, management, governance and automation. You need not build everything at once; AWS notes that you can start with a subset of its sample units and add more later. The usual failure is the opposite of too little structure: guardrails so tight, and new accounts so slow to obtain, that teams work around them. Offer new accounts as self-service through the platform, with the guardrails already in place.

Network foundations: VPCs, private connectivity and egress

A **virtual private cloud** (VPC; a virtual network on Azure) is your private network inside a provider. Plan its addresses first. On AWS the range a VPC is created with cannot be resized or removed, only added to, and two VPCs with overlapping ranges cannot be peered. Azure gives the same advice for its networks and for on-premises sites. So allocate every network a non-overlapping range from one central plan, with room to grow.

Subnets divide a VPC, and on AWS each one sits in a single availability zone, so resilience means repeating them per zone. What makes a subnet public or private is its routing: a public subnet has a route to an internet gateway, a private subnet has none and needs a NAT device to reach out, and an isolated subnet has no route beyond the VPC. The food-delivery platform might run three tiers across three zones: load balancers in public subnets, services in private ones and databases in isolated ones. Subnets are only coarse zones; finer control comes from the identity and segmentation methods of Section 8.5.

Next, connectivity. Peering links two networks, but it is not transitive: if A peers with B and with C, B still cannot reach C through A. As networks multiply, teams move to a hub such as AWS Transit Gateway or Azure's hub-spoke design, where a central network holds shared gateways, firewall and DNS. The hub then becomes a shared dependency, so Microsoft recommends one per region. To reach a data centre, a site-to-site VPN runs encrypted tunnels over the internet,

while AWS Direct Connect or Azure ExpressRoute gives a dedicated link; AWS notes that Direct Connect is not encrypted by default. To reach one service rather than a whole network, use AWS PrivateLink, Azure Private Link or Google Cloud Private Service Connect, which expose it at a private address without peering or internet access.

Finally, control what leaves. Many estates filter inbound traffic carefully and let anything go out, yet outbound access is how stolen data leaves, including through injected AI instructions (Section 12.6). A NAT gateway lets private services start outbound connections and stops outsiders starting them, but it only translates addresses and does not choose destinations. Choosing them takes an egress firewall, such as AWS Network Firewall or Azure Firewall with domain rules, placed so that all outbound traffic passes through it. The platform's order service then reaches only its payment, maps and messaging providers, and everything else is refused and logged.

Egress paths have reliability and cost sides too. A standard AWS NAT gateway lives in one zone, so if workloads in other zones share it, they lose internet access when that zone fails; AWS advises one per zone, and has added a regional option, so check the current version. NAT gateways are charged by the hour and by the gigabyte, while AWS gateway endpoints for S3 have no hourly or data-processing charge, so sending storage traffic that way is an easy saving (Section 10.8).

What cloud native means

The Cloud Native Computing Foundation (CNCF), part of the Linux Foundation and home of Kubernetes, publishes the most cited definition of cloud native. Version 1.0 (2018) described technologies that let organisations build and run scalable applications in dynamic environments such as public, private and hybrid clouds, and named containers, service meshes, microservices, immutable infrastructure and declarative APIs as typical examples. Version 1.1, approved in 2024, shifts the emphasis from technologies to practices: developing, building and deploying workloads in a programmatic, repeatable way at scale, characterised by “loosely coupled systems that interoperate in a manner that is secure, resilient, manageable, sustainable, and observable”.

Two consequences follow. Cloud native describes *how* a system is built and operated, not *where*: a Kubernetes cluster in a private data centre, provisioned from code and deployed many times a day, is cloud native; a hand-configured virtual machine copied unchanged into a public cloud is not. And cloud native is a means rather than an end. Its practices (automation, disposable components, declarative configuration, elasticity, observability) pay off when a system needs frequent change, variable scale and high availability; they add cost and complexity for a stable internal tool used by twenty people.

Migrations are often described with the seven Rs (see Section 11.6), which are worth knowing because they set expectations: *rehost* (lift and shift, unchanged), *relocate* (move whole virtual machines or platforms to the cloud without changing them), *replatform* (small changes such as moving to a managed database), *refactor* or re-architect (redesign for cloud-native operation), *repurchase* (replace with SaaS), *retain* (leave where it is) and *retire* (switch it off). Rehosting is fast but typically delivers few of the cloud's benefits and, run on always-on machines sized for peak, can cost more than the data centre it left.

When the cloud is not cheaper

The best-known counter-example is 37signals, the company behind Basecamp and HEY. In October 2022 its chief technology officer, David Heinemeier Hansson, announced that it was leaving the cloud, citing an annual AWS bill of more than USD 3 million for workloads with steady, predictable load. It bought roughly USD 700,000 of Dell servers, placed them in rented colocation space and moved its applications off AWS compute during 2023; it reported that this cut its cloud bill by about USD 2 million a year. In 2025 it announced the second stage: moving about 18 petabytes of files from Amazon S3 to roughly USD 1.5 million of storage arrays to replace an S3 bill of about USD 1.5 million a year, with the stated aim of closing its AWS account. The company has projected savings of about USD 10 million over five years. Moving workloads out of the cloud in this way is called cloud repatriation.

Both sides of the argument deserve a hearing. In favour of 37signals' reasoning: its load is steady, so it gained little from elasticity; it already employed experienced operations engineers and reported needing no extra staff; colocation providers handle power, cooling and physical security; and servers can be used for five years or more, so their cost per year of computing is far below on-demand rental. On the other side: 37signals is a mature business with predictable demand, which most new products are not; the comparison depends heavily on what the cloud bill would have been with commitment discounts and right-sizing; owning hardware means capital spending, lead times of weeks or months and capacity bought in advance; and a smaller team without deep operations skills would have to hire them, which can erase the saving. Critics also point out that the savings figures come from the company itself.

WORKED EXAMPLE

Worked example: renting or owning compute for the food-delivery platform

All prices are illustrative assumptions chosen to show the method, not real quotations. On-demand cloud capacity costs £0.040 per vCPU-hour; a three-year commitment costs £0.024 (a 40% discount); owned hardware in colocation costs £0.012 per vCPU-hour including power and space, but needs 30% headroom and staff time. The platform needs 40 vCPUs all day, rising to 120 vCPUs for the four-hour evening rush (about 122 hours a month). A month is 730 hours.

A: cloud, sized for peak, always on	120 vCPU × 730 h × £0.040 = **£3,504 a month**
B: cloud, committed baseline plus autoscaled peak	40 × 730 × £0.024 = £701; plus 80 × 122 × £0.040 = £390; total **£1,091 a month**
C: owned hardware sized for peak	120 × 1.3 × 730 × £0.012 = £1,367; plus a quarter of an engineer at £80,000 a year = £1,667; total **£3,034 a month**
Now scale up: a steady 2,000 vCPUs, no rush hour	Cloud committed: 2,000 × 730 × £0.024 = £35,040. Owned: 2,000 × 1.3 × 730 × £0.012 = £22,776 plus one engineer (£6,667) = **£29,443**, about 16% cheaper

For a small, spiky workload, elasticity dominates: option B costs less than a third of option A and about a third of option C, because it pays for peak capacity only during the rush.

For a large, flat workload, the fixed costs of ownership are spread over more capacity and ownership starts to win, which is the 37signals situation. The saving is real but smaller than headline comparisons suggest once headroom, staff and migration effort are counted.

The method matters more than the numbers: model the load shape, include people and headroom, compare against the best cloud price (with commitments), not the list price, and repeat the calculation as the business changes.

DECISION GUIDE

Decision guide: choosing a hosting model for a workload

Apply it per workload, not once for the whole organisation: a single company sensibly uses SaaS for payroll, functions for image resizing and containers for its core services.

Q1 **Is the capability generic, with a product on the market that fits well enough (email, payroll, CRM, payments)?**

- **Yes** Buy SaaS and integrate through its API; spend design effort on the integration and the exit plan.
- **No** Continue: you need to run your own software.

Q2 **Do law, contract or physical location rule out the public cloud for this workload (strict sovereignty rules, equipment on site, millisecond control loops)?**

- **Yes** Use a private cloud, colocation or an on-site edge deployment, connected to public cloud services as a hybrid where allowed.
- **No** Continue with public cloud options.

Q3 **Is the work event-driven, short-lived and spiky or low in volume (webhooks, file processing, scheduled jobs)?**

- **Yes** Use functions as a service or a serverless container platform; check cold-start, duration and cost-at-volume limits.
- **No** Continue: this is a long-running service.

Q4 **Does a PaaS or serverless container platform's model fit (stateless HTTP service, standard runtime, few special needs)?**

- **Yes** Use it. It removes the most operational work for a small team.
- **No** Continue: you need more control.

Q5 **Do you run many services across several teams, and have (or will fund) a team able to operate Kubernetes?**

- **Yes** Use managed Kubernetes (CaaS), ideally behind an internal platform (Section 10.7).
- **No** Use virtual machines with managed databases and autoscaling groups, built from infrastructure code.

Q6 **Is the load large, steady and predictable for years, with operations skills in house?**

Yes Price owned or colocated hardware against committed cloud capacity; repatriation may pay off.

No Stay on rented capacity and use commitment discounts for the steady baseline.

WATCH OUT

Common mistakes when choosing cloud models

- Rehosting everything unchanged onto always-on virtual machines sized for peak, then concluding that the cloud is expensive.
- Adopting multi-cloud as a general hedge, paying the portability cost on every workload instead of planning an exit for the few that need one.
- Assuming the provider backs up, encrypts and restricts access to your data by default; in every model these are your decisions.
- Choosing Kubernetes before there is a team to run it or enough services to justify it.
- Ignoring data transfer charges, especially between zones, regions and providers, when drawing the deployment diagram.

CHECK YOUR UNDERSTANDING

▼ **A team moves its database from a self-run VM to a managed database service. Name two responsibilities that move to the provider and two that stay with the team.**

The provider takes over engine patching, hardware and usually automated backups and zone failover. The team keeps schema and query design, access permissions, encryption choices, capacity and cost, testing restores and planning major-version upgrades.

▼ **Why does a quorum-based system such as etcd prefer three availability zones to two?**

A quorum needs a majority. With two zones, losing the zone that holds the majority stops the system; with three, any single zone can fail and a majority survives.

▼ **Is a hand-built virtual machine copied unchanged into a public cloud cloud native?**

No. Cloud native describes practices such as automation, declarative configuration, disposable components and observability, not location. Rehosting moves the system without adopting those practices.

▼ **Why might the same organisation sensibly use SaaS, functions and Kubernetes at once?**

The hosting model should be chosen per workload: generic capabilities are bought as SaaS, spiky event-driven tasks fit functions, and long-running core services may justify containers on Kubernetes.

▼ **Try it: Illustrative: on-demand is £0.05 per vCPU-hour, a commitment £0.03. You need 20 vCPUs always, plus 60 more for five hours a day, in a 30-day month. Compare all on-demand, baseline committed, and committing everything for peak.**

Baseline: 20 x 720 = 14,400 vCPU-hours; peak extra: 60 x 150 = 9,000. All on-demand: 23,400 x £0.05 = £1,170. Commit the baseline: 14,400 x £0.03 + 9,000 x £0.05 = £882. Commit 80 vCPUs all month: 80 x 720 x £0.03 = £1,728. Commit only to the certain baseline and let on-demand cover the rush. Owned hardware would only make sense for large, steady load.

SOURCES

- NIST SP 800-145: The NIST Definition of Cloud Computing (2011): https://csrc.nist.gov/pubs/sp/800/145/final
- CNCF Technical Oversight Committee: Cloud Native Definition: https://github.com/cncf/toc/blob/main/DEFINITION.md
- AWS: Shared Responsibility Model: https://aws.amazon.com/compliance/shared-responsibility-model/
- Microsoft Learn: Shared responsibility in the cloud: https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility
- The Register (2025): 37signals is completing its on-prem move: https://www.theregister.com/2025/05/09/37signals_cloud_repatriation_storage_savings
- Designing an AWS Control Tower landing zone (AWS Prescriptive Guidance): https://docs.aws.amazon.com/prescriptive-guidance/latest/designing-control-tower-landing-zone/introduction.html
- Sample account structure and OUs (AWS Prescriptive Guidance): https://docs.aws.amazon.com/prescriptive-guidance/latest/designing-control-tower-landing-zone/account-structure.html
- What is an Azure landing zone? (Microsoft Cloud Adoption Framework): https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/
- Azure landing zone design areas (Microsoft): https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/design-areas

- Organizing your AWS environment using multiple accounts (AWS whitepaper): https://docs.aws.amazon.com/whitepapers/latest/organizing-your-aws-environment/organizing-your-aws-environment.html
- Service control policies (AWS Organizations): https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_scps.html
- Resource hierarchy (Google Cloud): https://docs.cloud.google.com/resource-manager/docs/cloud-platform-resource-hierarchy
- VPC CIDR blocks (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/vpc-cidr-blocks.html
- Subnets for your VPC (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/configure-subnets.html
- VPC peering basics (AWS): https://docs.aws.amazon.com/vpc/latest/peering/vpc-peering-basics.html
- What is AWS Transit Gateway?: https://docs.aws.amazon.com/vpc/latest/tgw/what-is-transit-gateway.html
- Hub-spoke network topology in Azure (Microsoft): https://learn.microsoft.com/en-us/azure/architecture/networking/architecture/hub-spoke
- AWS Direct Connect (Amazon VPC connectivity options whitepaper): https://docs.aws.amazon.com/whitepapers/latest/aws-vpc-connectivity-options/aws-direct-connect.html
- What is AWS PrivateLink?: https://docs.aws.amazon.com/vpc/latest/privatelink/what-is-privatelink.html
- Private Service Connect (Google Cloud): https://docs.cloud.google.com/vpc/docs/private-service-connect
- NAT gateways (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html
- NAT gateway basics (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/nat-gateway-basics.html
- Amazon VPC pricing: https://aws.amazon.com/vpc/pricing/
- What is AWS Network Firewall?: https://docs.aws.amazon.com/network-firewall/latest/developerguide/what-is-aws-network-firewall.html

SECTION 10.2 OF 10.8 · ADVANCED · 26 MIN READ · LAST REVIEWED NOVEMBER 2026

Containers and orchestration with Kubernetes

BEFORE YOU READ THIS

This section builds on:

- 0.7 Servers and the cloud
- 9.1 Scaling
- 9.2 Load balancing

Terms it uses from earlier: Container, Autoscaling, Availability zone, PaaS (platform as a service)

THE IDEA IN BRIEF

Think of a large port. Goods arrive in standard shipping containers, so every crane, ship and lorry can handle them without knowing what is inside. The harbour master does not move boxes personally; he keeps a board saying what should be where, and his staff constantly check the quay against the board and fix any difference: a box missing, a crane broken, a ship late. Software containers are the standard boxes, and Kubernetes is the harbour master's office: you write down what should be running, and it keeps checking and correcting until reality matches.

WHAT YOU NEED TO KNOW

- A container image is a stack of content-addressed layers plus configuration and a manifest; tags can move, digests cannot, so production should run images pinned by digest.
- The OCI runtime, image and distribution specifications let images built by any tool be stored in any registry and run by any compliant runtime.
- Kubernetes has a control plane (API server, etcd, scheduler, controller manager) and worker nodes (kubelet, container runtime, kube-proxy); only the API server talks to etcd, and nodes pull their work.
- Controllers run level-triggered reconciliation loops that compare actual with desired state and act on the difference, which makes the system robust to failures and missed events.
- Pods are disposable; Deployments roll them out, Services give them stable names, Gateway API routes traffic in, StatefulSets give identity and storage, Jobs run to completion and the HPA scales replicas.

- Requests drive scheduling and autoscaling, limits cap use (CPU is throttled, memory overuse is killed); liveness probes must not check shared dependencies.
- Kubernetes pays off for many services, several teams and a platform team; for a few stateless services a PaaS or serverless container platform is usually the better choice.

Images, layers and registries

Chapter 0 introduced containers as isolated processes that share the host's kernel, packaged as images built from a Dockerfile (servers and the cloud). An architect needs a little more detail, because image design affects build speed, security and how reliably the same software reaches every environment.

A container image is a stack of read-only *layers*, each a compressed set of file changes, plus a small configuration document (the command to run, environment variables, the user) and a *manifest* that lists them. Every layer and manifest is identified by a *digest*, a SHA-256 hash of its content. Identical layers are stored and downloaded once, so a hundred images built on the same base share its layers. Because each instruction in a Dockerfile produces a layer and builders reuse cached layers until something changes, instructions are ordered from least to most frequently changed: base image, then dependencies, then application code. *Multi-stage builds* compile in one stage and copy only the result into a small final image, leaving compilers and build tools behind.

Images live in a *registry*: Docker Hub, GitHub Container Registry, Amazon ECR, Azure Container Registry, Google Artifact Registry, or a self-hosted one such as Harbor. A *tag* such as `orders:2.4.1` is a movable label; anyone with push rights can point it at a different image tomorrow. A digest such as `orders@sha256:3f1c...` can never change meaning. Production deployments should therefore resolve tags to digests, so that what was tested is exactly what runs, and should never use `latest`.

Small images are faster to pull and have fewer components to patch. Common practice is a minimal base (a slim distribution or a distroless image containing only the runtime), running as a non-root user, rebuilding regularly to pick up security fixes, and scanning images for known vulnerabilities in the pipeline. Signing images and attaching a software bill of materials belong to supply-chain security, covered in Chapter 8 (software supply-chain security).

The OCI standards

Docker popularised containers from 2013, and in 2015 Docker and others founded the Open Container Initiative (OCI) under the Linux Foundation to stop the format fragmenting. The OCI maintains three specifications. The *runtime specification* says how to run an unpacked

container (namespaces, control groups, mounts, the process to start); `runc` is its reference implementation. The *image specification* defines layers, manifests and configuration. The *distribution specification* defines the HTTP API that registries expose for pushing and pulling. Version 1.1 of the image and distribution specifications (2024) added standard ways to store other artefacts in registries and to link them to an image, which is how signatures and software bills of materials now travel alongside the images they describe.

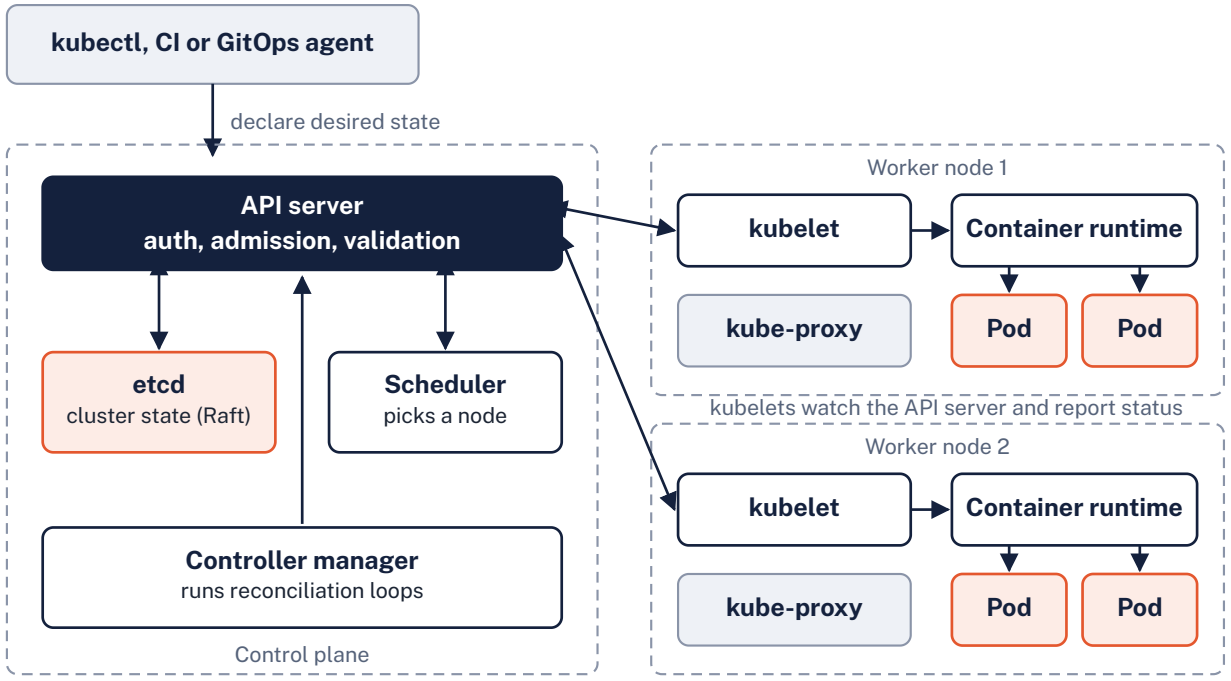
The practical result is interchangeability. An image built by Docker, Podman, Buildah or a cloud build service can be stored in any compliant registry and run by any compliant runtime: containerd and CRI-O on a Kubernetes node, a serverless container platform or a developer's laptop. That is why containers, unlike most packaging formats before them, became a lingua franca between development, operations and every major cloud.

Why orchestration is needed

Running one container is a single command. Running the food-delivery platform is not: forty services, three to thirty copies each, spread over dozens of machines in three availability zones, with a new version of something going out every hour. Someone has to decide which machine each copy runs on, restart copies that crash, replace machines that die, add copies for the evening rush and remove them afterwards, roll out new versions without dropping requests, let services find each other as copies come and go, hand them configuration and secrets, and attach storage to the few that need it.

Container orchestration is software that does all of this from a declaration of what should be running. Google ran an internal system called Borg for this for more than a decade; in 2014 it announced Kubernetes, an open-source system drawing on that experience, released version 1.0 in July 2015 and donated it to the newly founded Cloud Native Computing Foundation, where in 2018 it became the first project to graduate. Alternatives exist (HashiCorp Nomad, Amazon ECS, serverless container platforms), but Kubernetes is now the common substrate offered by every major provider and by most on-premises platforms.

Kubernetes architecture



Every component talks to the API server; only the API server talks to etcd, and nodes pull their work rather than being pushed commands.

How Kubernetes works: control plane and nodes

A Kubernetes cluster has a control plane, which makes decisions, and worker nodes, which run the workloads. The **API server** is the front door: every client, from a developer's `kubectl` to the cluster's own components, reads and writes objects through its REST API, and it authenticates the caller, checks authorisation and runs *admission* controls that can validate or modify each request (for example, rejecting containers that run as root). It is the only component that talks to **etcd**, a strongly consistent key-value store that replicates the cluster state with the Raft consensus protocol across, typically, three or five members, so that a majority survives the loss of one or two (consistency, CAP and PACELC).

The **scheduler** watches for new pods with no node assigned, filters out nodes that cannot run them (not enough unreserved CPU or memory, wrong zone, missing hardware, taints the pod does not tolerate), scores the rest (spreading copies across zones, packing efficiently) and records its choice through the API server. The **controller manager** runs dozens of controllers, each responsible for one kind of object: the ReplicaSet controller keeps the right number of pods, the Node controller notices nodes that stop reporting, the Job controller runs tasks to completion. In a cloud, a cloud controller manager also creates load balancers and attaches disks through the provider's API.

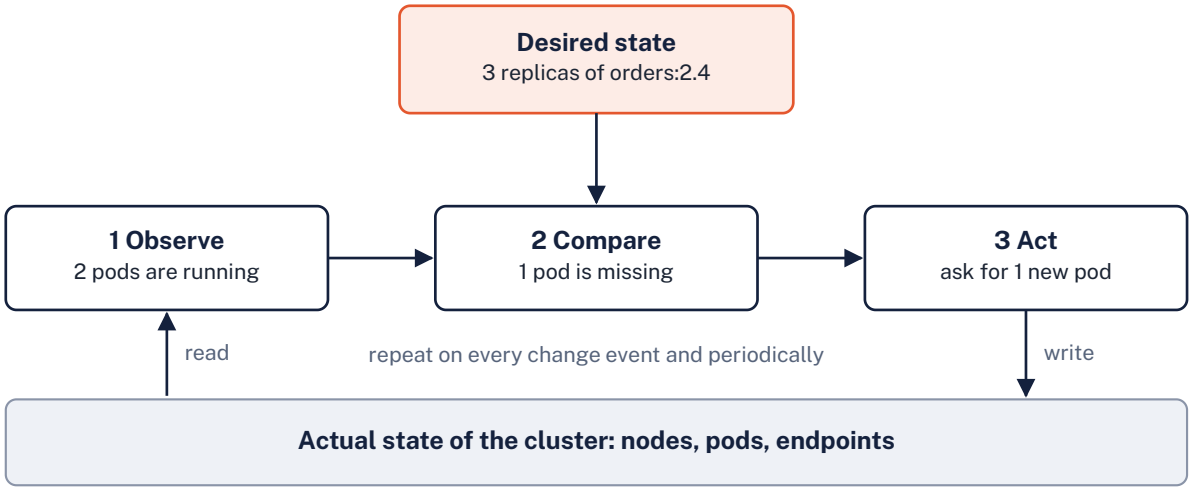
Each worker node runs a **kubelet**, an agent that watches the API server for pods assigned to its node, asks the **container runtime** (usually containerd or CRI-O, through the Container Runtime Interface; Docker Engine's built-in shim was removed in version 1.24 in 2022) to start them, runs their health probes and reports status back. **kube-proxy** programs the node's packet-forwarding rules so that traffic sent to a Service's virtual address reaches one of its pods; some network plugins, such as Cilium, replace it with an eBPF data plane. A network plugin gives every pod its own IP address, and a cluster DNS service (usually CoreDNS) lets pods find Services by name.

Notice the direction of the arrows. The control plane does not log in to a node to start workloads; nodes watch for the state they should be in and converge on it. (On-demand commands are the exception: for `kubectl exec`, `logs` and `port-forward`, the API server calls the kubelet on the node.) This pull-based, watch-and-converge design is what makes Kubernetes robust: a node that was disconnected for ten minutes simply catches up when it returns. With managed Kubernetes (Amazon EKS, Azure Kubernetes Service, Google Kubernetes Engine and others) the provider runs and upgrades the control plane; the customer still chooses node types, upgrades node pools, and owns everything deployed into the cluster.

Declarative desired state and reconciliation loops

The central idea of Kubernetes is declarative configuration. You do not tell the cluster to start three containers; you store an object saying that three copies of a given image should exist, the desired state. Controllers then run reconciliation loops: observe the actual state, compare it with the desired state and act to remove the difference, forever.

The reconciliation loop



Controllers are level-triggered: they act on the current difference, not on a remembered sequence of events, so missed events and restarts do no lasting harm.

Follow what happens when a node in zone B fails during the evening rush. The Node controller stops receiving heartbeats and marks the node not ready; after a timeout its pods are marked for eviction. The ReplicaSet controller for the orders service observes two running pods where three are desired and creates a new pod object. The scheduler sees an unscheduled pod and assigns it to a healthy node, preferring zone B's survivors or another zone according to the spreading rules. That node's kubelet sees the assignment, pulls the image and starts the container. When its readiness probe passes, the endpoints behind the orders Service are updated and traffic flows to it. No component told any other what to do; each reacted to the state it observed.

Because controllers are *level-triggered* (they react to the current difference) rather than *edge-triggered* (reacting to individual events), a controller that crashes, misses a notification or runs twice still converges to the right result. The same pattern can be extended: *custom resource definitions* add new object types, and the *operator* pattern, introduced by CoreOS in 2016, packages a controller that encodes operational knowledge, for example creating, backing up and failing over a PostgreSQL cluster from a single declared object. Section 10.4 shows the same loop applied to whole environments through GitOps.

Declarative control has a cost: you must think in terms of end states and let go of exact sequencing. When two controllers want different desired states for the same field, they fight. A classic example is a Deployment manifest in Git that says `replicas: 3` while an autoscaler sets it to nine; every GitOps sync scales the service back down in the middle of the rush. The fix is to let exactly one owner manage each field, here by removing `replicas` from the manifest.

The core Kubernetes objects

Object	What it is	Food-delivery example
Pod	One or more containers scheduled together on one node, sharing an IP address and volumes. Pods are disposable and replaced, never repaired.	One copy of the orders service, perhaps with a small log-shipping sidecar.
ReplicaSet	Keeps a stated number of identical pods running. Rarely written by hand.	Created and managed by the orders Deployment.
Deployment	Declares a stateless application's pod template and replica count, and rolls out changes by moving pods from an old ReplicaSet to a new one, within surge and unavailability limits.	Orders, menu and pricing services.
Service	A stable name and virtual IP for a changing set of pods selected by label, with load spreading across them.	`orders.default.svc`, used by the checkout service.
Ingress and Gateway API	Route external HTTP(S) traffic into Services by host and path. Gateway API (generally available since version 1.0 in October 2023) is the more expressive, role-oriented successor: the platform team owns the Gateway, application teams own their HTTPRoutes.	`api.example.com/orders` routed to the orders Service, with TLS.
ConfigMap	Non-secret configuration delivered to pods as environment variables or files.	Feature settings, timeouts, the URL of the menu service.
Secret	Sensitive values delivered like ConfigMaps. Stored only base64-encoded in etcd unless encryption at rest is configured; many teams sync them from an external secrets manager.	The payment provider's API key, the database password.
StatefulSet	Pods with stable names, stable network identities and their own persistent volumes, started and stopped in order.	A self-managed Kafka or Redis cluster, if not using a managed service.
DaemonSet	Exactly one pod on every (or every selected) node.	Log collector or node monitoring agent.
Job and CronJob	Pods that run to completion, once or on a schedule, with retries.	Database migration before a release; nightly restaurant payout report.

Object	What it is	Food-delivery example
HorizontalPodAutoscaler	Adjusts a Deployment's replica count to keep a metric near a target.	Scale orders from 3 to 12 replicas as CPU use rises in the evening.

NOTE

Ingress controllers and the move to Gateway API

The Ingress object only declares routing rules; an *ingress controller* implements them. In November 2025 the Kubernetes project announced the retirement of the widely used community controller, ingress-nginx, with best-effort maintenance ending in March 2026 and no security fixes after that, and recommended Gateway API for new work. The Ingress API itself still works with other controllers, but teams running ingress-nginx have had to plan a migration, a reminder that even a de facto standard component of a platform has a lifecycle that must be tracked.

A Deployment and a Service in YAML

The manifests below declare the orders service: configuration from a Secret, resource requests and probes, a stable Service in front and an autoscaler that keeps between three and twelve replicas. The Deployment deliberately has no `replicas` field, so that the autoscaler is the only owner of the count. They are applied with `kubectl apply` or, better, by a pipeline or GitOps agent from a repository.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders
spec:
  # no replicas field: the autoscaler below owns the count
  selector:
    matchLabels: {app: orders}
  template:
    metadata:
      labels: {app: orders}
    spec:
      containers:
      - name: orders
        image: registry.example.com/orders:2.4.1
        ports: [{containerPort: 8080}]
        env:
        - name: DATABASE_URL
          valueFrom: {secretKeyRef: {name: orders-db, key: url}}
        resources:
          requests: {cpu: 500m, memory: 512Mi}
          limits: {memory: 512Mi}  # no CPU limit: may burst
        readinessProbe:
          httpGet: {path: /ready, port: 8080}
          periodSeconds: 5
        livenessProbe:
          httpGet: {path: /live, port: 8080}
          periodSeconds: 10
          failureThreshold: 3
```

A Deployment for the orders service. In production the image would be pinned by digest rather than tag.

```
apiVersion: v1
kind: Service
metadata:
  name: orders
spec:
  selector: {app: orders}
  ports: [{port: 80, targetPort: 8080}]
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: orders
spec:
  scaleTargetRef: {apiVersion: apps/v1, kind: Deployment, name: orders}
  minReplicas: 3
  maxReplicas: 12
  metrics:
  - type: Resource
    resource:
      name: cpu
      target: {type: Utilization, averageUtilization: 60}
```

A Service that gives the pods one stable address, and an autoscaler that targets 60% average CPU use of the requested CPU.

Resource requests, limits and quality of service

Requests and limits are the most consequential lines in a manifest. A *request* is what the scheduler reserves: a pod requesting 500 millicores (half a CPU) and 512 MiB of memory is placed only on a node with that much unreserved capacity, and it is guaranteed that share under contention. A *limit* is a ceiling enforced by the kernel: a container that exceeds its CPU limit is throttled (slowed down), and one that exceeds its memory limit is killed and restarted, the familiar `OOMKilled` status. Kubernetes derives a quality-of-service class from these settings: *Guaranteed* (requests equal limits for every resource), *Burstable* (requests set, limits higher or absent) and *BestEffort* (nothing set, first to be evicted when a node runs short).

Practitioners disagree about CPU limits. One camp sets them everywhere for predictability and fairness between teams; another sets accurate CPU requests and no CPU limits, because throttling causes latency spikes even when the node has idle CPU, and requests already guarantee each pod its share. For memory there is little debate: set the limit equal to the request, because memory cannot be reclaimed from a process without killing it, and a pod using more than it requested is the first candidate for eviction. Either way, requests must be based on measurements; inflated requests waste money (the node is reserved but idle), and understated ones cause noisy-neighbour problems and misleading autoscaling.

The HorizontalPodAutoscaler computes *desired replicas = ceiling(current replicas × current metric ÷ target)*, measuring CPU use as a percentage of the request. It scales up quickly but, by default, waits for a five-minute stabilisation window before scaling down, to avoid flapping. When new pods cannot fit on existing nodes they stay *Pending*, and a node autoscaler (the Cluster Autoscaler, or Karpenter on some providers) adds machines, which takes minutes rather than seconds. Pod autoscaling and node autoscaling are separate loops with separate delays, and both must be planned for the rush.

WORKED EXAMPLE

Worked example: sizing the orders service for the evening rush

Each orders pod requests 500m CPU and 512 MiB, with a memory limit of 512 MiB and no CPU limit. Nodes have 4 vCPUs and 16 GiB, of which about 3.8 vCPUs and 14.5 GiB are allocatable after system reservations (illustrative figures). The HPA targets 60% CPU, with 3 to 12 replicas.

Pods per node	CPU: 3.8 ÷ 0.5 = 7 pods. Memory: 14.5 GiB ÷ 0.5 GiB = 29 pods. CPU is the binding constraint: **7 orders pods per node** if nothing else runs there.
18:00, rush begins	4 replicas averaging 90% of their CPU request. Desired = ceiling(4 × 90 ÷ 60) = **6 replicas**.
18:20, still rising	6 replicas at 85%. Desired = ceiling(6 × 85 ÷ 60) = ceiling(8.5) = **9 replicas**, requesting 4.5 vCPUs in total.
Node capacity	If other services already fill the nodes, three of the new pods stay Pending; the node autoscaler adds a node, which takes perhaps two to four minutes to join and pull the image.
Design response	Raise `minReplicas` before the predictable rush (scheduled scaling), keep some spare node capacity, and keep the image small so that new pods start quickly.
Effect of a wrong request	If each pod really needed only 250m but the request stayed at 500m, the same rush load (225m per pod) would show as 45% instead of 90%: the HPA would not scale until use passed 300m per pod, and the cluster would reserve twice the CPU it needs.

Autoscaling reacts to load that has already arrived; for a predictable daily peak, scheduled pre-scaling is cheaper than lost orders.

Requests drive both the scheduler and the autoscaler, so measuring them is a first-order architectural task, not tuning trivia.

Liveness, readiness and startup probes

Probes are how Kubernetes learns about health. A **readiness** probe answers whether this pod should receive traffic now; while it fails, the pod is removed from its Service's endpoints but left running. A **liveness** probe answers whether the process is stuck beyond recovery; when it fails repeatedly, the kubelet restarts the container. A **startup** probe holds off the other two until a slow-starting application has finished initialising.

The most damaging mistake is a liveness probe that checks dependencies. If the orders service's liveness endpoint queries the database, a thirty-second database failover makes every orders pod fail liveness at once, Kubernetes restarts them all, they all reconnect at the same moment, and a brief blip becomes an outage. Liveness should check only that the process itself can respond; readiness may consider whether the pod can do useful work, though even there many teams avoid making all pods unready because of a shared dependency, preferring to fail requests fast with a circuit breaker (see resilience patterns).

Probes work together with graceful shutdown. When a pod is deleted during a rollout or scale-down, Kubernetes removes it from endpoints and sends the process `SIGTERM`; the process should stop accepting new work, finish requests in flight and exit within the grace period (30 seconds by default) before it is killed. Because endpoint removal propagates asynchronously, a short `preStop` delay avoids sending requests to a pod that has already begun to stop. Section 10.3 returns to this as the twelve-factor principle of disposability.

When Kubernetes is overkill

Kubernetes is a general-purpose platform for building platforms, and it charges for that generality. Someone must design networking, ingress, identity, secrets, policies, observability and multi-tenancy; keep up with three minor releases a year, each supported for roughly fourteen months, plus the add-ons that depend on them; and be on call for the cluster as well as the applications. Managed control planes remove some of this work, not most of it.

For a small team running a handful of stateless web services, a PaaS or a serverless container platform (Google Cloud Run, Azure Container Apps, Amazon ECS with Fargate and similar) gives most of the benefits (containers, autoscaling, rolling deployments, managed TLS) with a fraction of the operational load. Kubernetes starts to pay when there are many services and several teams, a platform team to run it, workloads with unusual needs (stateful systems, GPUs, batch jobs, custom controllers), a requirement to run the same platform on premises and in several clouds, or an ecosystem of tools the organisation wants to use. Many organisations have adopted it because it was fashionable and then spent years building, in effect, a worse PaaS on top of it. Section 10.7 discusses how platform teams avoid that fate (platform engineering).

IF YOU ARE NEW TO THIS

If you are new to this

You do not need to memorise every Kubernetes object. Remember three ideas: you describe what you want in files and the system makes it so; small disposable units (pods) are replaced rather than repaired; and stable names (Services) hide the fact that the copies behind them keep changing. Everything else is detail built on those three.

CHECK YOUR UNDERSTANDING

▼ Why should a production deployment refer to an image by digest rather than by tag?

A tag is a movable label that can be repointed at different content; a digest is a hash of the content and always identifies exactly the image that was tested.

▼ A node dies. Which components act, in what order, to restore the third replica?

The Node controller marks the node not ready and its pods are evicted; the ReplicaSet controller sees too few pods and creates one; the scheduler assigns it to a healthy node; that node's kubelet starts it through the runtime; when readiness passes it joins the Service's endpoints.

▼ What happens to a container that exceeds its memory limit, and to one that exceeds its CPU limit?

Exceeding the memory limit gets the container killed (OOMKilled) and restarted; exceeding the CPU limit only throttles it.

▼ Why is checking the database in a liveness probe dangerous?

A brief database problem makes every pod fail liveness together, so Kubernetes restarts all of them at once, turning a short dependency blip into a full outage.

▼ Try it: Each pod requests 750m CPU and 1 GiB. Nodes offer 7.6 vCPU and 28 GiB allocatable. The rush needs 24 pods. How many pods fit per node, and how many nodes do you need if you must survive losing one?

CPU allows 7.6 / 0.75 = 10.1, so 10 pods; memory allows 28, so CPU is the binding limit. Twenty-four pods need 2.4, so 3 nodes. Surviving a node failure takes one more: 4 nodes. Requests drive scheduling, so keep them near real use. Spread pods across zones. Check the autoscaler's maximum allows 24 replicas, and that the cluster can add nodes before the rush.

SOURCES

- Kubernetes documentation: Cluster components: https://kubernetes.io/docs/concepts/overview/components/
- Kubernetes documentation: Resource management for pods and containers: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
- Kubernetes documentation: Configure liveness, readiness and startup probes: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
- Open Container Initiative: https://opencontainers.org/
- Kubernetes blog (2025): Ingress NGINX retirement: https://kubernetes.io/blog/2025/11/11/ingress-nginx-retirement/

SECTION 10.3 OF 10.8 · INTERMEDIATE · 25 MIN READ · LAST REVIEWED NOVEMBER 2026

The twelve-factor app and cloud-native principles

BEFORE YOU READ THIS

This section builds on:

- 0.8 How software gets built
- 9.2 Load balancing
- 10.2 Containers and Kubernetes

Terms it uses from earlier: Container image, Continuous delivery, Idempotent, Scaling out (horizontal scaling), Observability

THE IDEA IN BRIEF

Think of a chain of takeaway kitchens. A good recipe card works in any branch: it lists every ingredient, says nothing about which branch it is in, and any cook can pick it up mid-shift because nothing important is kept in one cook's head. Deliveries arrive in labelled boxes, and when the menu changes a new card is printed and numbered rather than scribbled on. The twelve-factor app is that kind of recipe card for software: rules that let the same application be started, copied, moved, replaced and scaled by machines without anyone needing to know its quirks.

WHAT YOU NEED TO KNOW

- The Twelve-Factor App (Adam Wiggins, Heroku, 2011, open-sourced in 2024) describes the contract between an application and a modern platform, and applies to monoliths and microservices alike.
- Config (III) and backing services (IV) keep everything that varies between deploys outside the artefact, so one artefact runs in every environment; secrets now belong in a secrets manager or workload identity rather than plain environment variables.
- Build, release, run (V) makes releases immutable and numbered, so that rollback means running an earlier release and nobody edits a running server.
- Stateless processes (VI), concurrency by process type (VIII) and disposability (IX) are what make horizontal scaling, rolling updates and replacement of failed instances safe.
- Dev/prod parity (X), logs as streams (XI) and admin processes run as separate one-off processes (XII) reduce production surprises; migrations must be compatible with old and new code.

- Beyond the twelve factors, cloud-native applications are designed for disposability, immutability, observability and security, and Hoffman's 2016 extension adds API first, telemetry and authentication and authorisation.
- An audit against the factors should rank gaps by risk: exposed secrets first, then anything that blocks the target platform such as local state, then reliability and convenience.

Where the twelve factors came from

In 2011 Adam Wiggins, a co-founder of Heroku, published *The Twelve-Factor App*, a short web document distilling what the Heroku team had learned from watching a very large number of applications being built, deployed and scaled on its platform. The twelve-factor app methodology describes applications that set themselves up from declarative, automated steps, have a clean contract with the platform beneath them so that they can run anywhere, keep development and production close enough for continuous deployment, and can scale out without redesign.

The document predates Docker (2013) and Kubernetes (2014), yet it describes almost exactly the contract those platforms later enforced: an application is a set of stateless processes, built into an immutable artefact, configured from outside, writing logs to a stream and talking to everything else over the network. That is why it remains the common vocabulary of cloud-native application design. In November 2024 Heroku announced that the methodology was now open source, maintained by a community of contributors from several companies, with the aim of refreshing it for current practice; its original twelve factors remain the reference point.

The factors are about the application's relationship with its runtime platform, not about its internal architecture. A modular monolith can be perfectly twelve-factor; a set of microservices can violate every factor. They complement the deployability and operability concerns of Chapter 2 and the scaling and statelessness principles of Chapter 9.

The twelve factors, with modern caveats

Factor	What it means	Why it matters	Modern caveat
I. Codebase	One codebase in version control per app, deployed to many environments.	Every environment runs some version of the same code; no forks per customer or per environment.	A monorepo holding several apps is fine if each app has a clear boundary; the anti-pattern is copy-and-modify forks.
II. Dependencies	Declare every dependency explicitly and isolate the app from the host system.	Builds are reproducible on any machine; nothing relies on a package someone installed by hand.	Lock files and images make this the default; pinned dependencies still need regular updating and vulnerability scanning.
III. Config	Keep everything that varies between deploys (URLs, credentials, feature settings) out of the code, in the environment.	One artefact runs in every environment; secrets never enter the repository.	Environment variables leak into logs, crash reports and child processes; secrets are better read from a secrets manager or mounted files.
IV. Backing services	Treat databases, queues, caches and external APIs as attached resources located by configuration.	A local database can be replaced by a managed one, or a failed one by a restored copy, by changing a URL.	Swaps are rarely seamless when semantics differ; isolate differences behind ports and adapters.
V. Build, release, run	Separate building an artefact, combining it with config into a release, and running it.	Releases are immutable and numbered; any release can be rerun; nobody edits code on a server.	Today: build once in CI, store the image by digest, record releases in Git or the deployment tool.
VI. Processes	Run the app as one or more stateless processes that share nothing; keep state in backing services.	Any copy can serve any request, so copies can be added, removed and replaced freely.	Sticky sessions hide the problem rather than fix it; local disk is only a scratch area.
VII. Port binding	The app is self-contained and exports its service by listening on a port.	The platform routes traffic to it without injecting it into a separate web server.	Universal now; queue consumers have no port, and service meshes may sit in front of it.

Factor	What it means	Why it matters	Modern caveat
VIII. Concurrency	Scale out by running more processes, with different process types for different work (web, worker).	Each kind of work scales independently to its own load.	Autoscalers do this automatically, including on queue length; threads within a process still matter.
IX. Disposability	Start fast, shut down gracefully on SIGTERM, survive sudden death.	Fast scaling, safe rolling deployments and robustness to crashes.	Slow-starting runtimes need startup probes or faster start techniques; background jobs must be idempotent and resumable.
X. Dev/prod parity	Keep development, staging and production as similar as possible in time, people and tools.	Fewer surprises that only appear in production; continuous deployment becomes safe.	Full parity is too costly; prioritise the same types and versions of backing services, using containers locally.
XI. Logs	Write logs as an unbuffered event stream to standard output; let the platform route them.	The app does not manage files; logs from all copies are collected centrally.	Use structured logs plus metrics and traces (for example through OpenTelemetry); watch log volume and cost.
XII. Admin processes	Run one-off tasks (migrations, scripts) as separate processes using the same code and config.	Admin tasks run against exactly the release in production, not a stale copy.	Run them as pipeline steps or Kubernetes Jobs; never let every replica run migrations at start-up without a lock.

Config, secrets and backing services

The config factor has a simple test: could the codebase be made public today without exposing a single credential? If not, configuration has leaked into the code. Separating configuration from code lets one artefact move through every environment unchanged, which is the foundation of the build-once principle in Section 10.5 (CI/CD and deployment). Configuration includes database and service URLs, credentials, and per-environment settings; it excludes internal wiring that is the same everywhere, such as which class handles which route, which belongs in the code.

The original recommendation to use environment variables has aged unevenly. They are universal and language-neutral, and Kubernetes, PaaS platforms and functions all support them. But a process's environment is visible to child processes, is often dumped in crash reports and debugging pages, and cannot be rotated without a restart. Current practice for

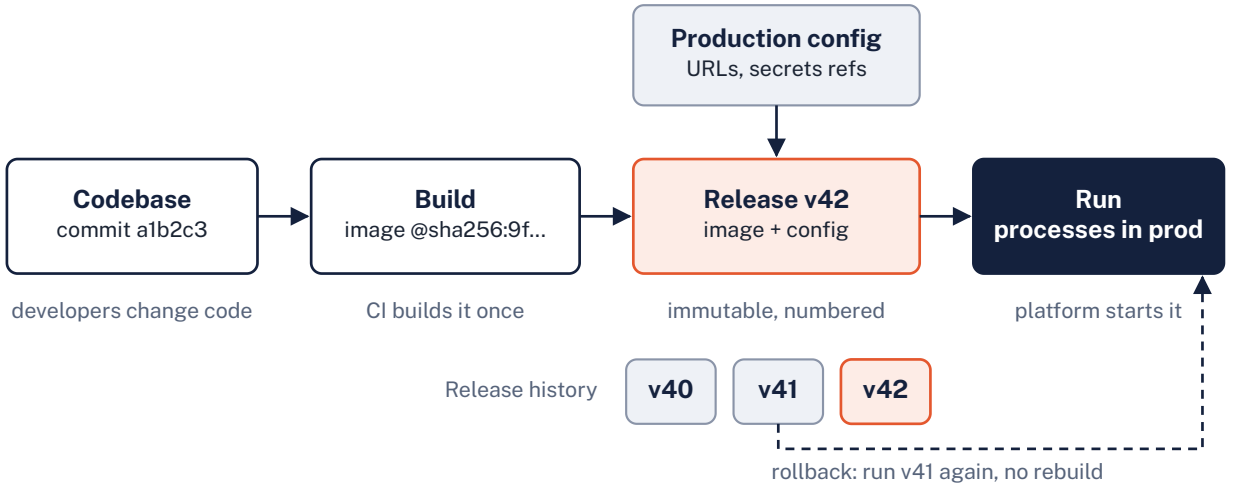
secrets is to fetch them at start-up from a secrets manager (AWS Secrets Manager, Azure Key Vault, Google Secret Manager, HashiCorp Vault or OpenBao) or mount them as files that the platform can update, and better still to replace long-lived secrets with workload identity, where the platform gives each service short-lived credentials automatically. The factor's intent, keeping configuration outside the artefact, is unchanged.

Treating a backing service as an attached resource means that the code reaches the orders database, the payment provider and the message broker the same way: through a URL and credentials supplied at deploy time. That makes environments composable and recovery quicker. It does not make backing services interchangeable by magic; replacing one message broker with another changes delivery semantics. The hexagonal style of Chapter 5 complements the factor: configuration says where the resource is, and an adapter behind a port hides how it behaves.

Build, release, run

The build, release, run factor separates three stages that are often blurred. The *build* stage turns a commit into an executable artefact, today usually a container image identified by its digest. The *release* stage combines that artefact with the configuration of one environment and gives the result an identity, such as release v42 of the orders service in production. The *run* stage starts processes from a release. Releases are immutable: a change of code or configuration produces a new release, never an edit to an existing one.

Build once, release per environment, run anywhere



Because releases are immutable and numbered, rolling back means running an earlier release again, not rebuilding or editing anything.

This separation is what makes deployment boring. Nobody edits files on a server, so servers do not drift; the same image that passed testing in staging runs in production, so a test result means something; and rollback is a fast, rehearsed operation instead of an emergency rebuild. It is also an audit trail: for any moment you can say exactly which code and which configuration were running. Kubernetes keeps a revision history of each Deployment's ReplicaSets, PaaS platforms number releases, and GitOps (Section 10.4) makes the Git history of the release repository the record.

Processes, concurrency and disposability

Stateless, share-nothing processes are the precondition for almost everything else in this chapter: horizontal scaling, rolling deployments, autoscaling, replacing failed nodes and spreading across zones. Anything a process keeps only in its memory or on its local disk (a user's session, an uploaded file, a partly built report) is lost when that process is replaced, and is invisible to the other copies. Chapter 9 discusses session handling in detail (load balancing, statelessness and sessions). State does not disappear; it moves to backing services designed to keep it.

The concurrency factor adds that different kinds of work should run as different process types that scale separately. In the food-delivery platform, the web process serving the customer app scales with request rate, while a worker process sending order confirmations scales with queue length and a courier-location consumer with event throughput. Mixing them in one process type means scaling all of them for the needs of one.

Disposability asks that processes start in seconds and stop cleanly. On `SIGTERM` a web process stops accepting connections and finishes requests in flight; a worker returns its current message to the queue or finishes it within the grace period. Because processes can also die without warning (a node failure, an out-of-memory kill), work must be safe to repeat: idempotent message handling, as in event-driven architecture, and jobs that resume from checkpoints. George Candea and Armando Fox pushed the idea to its limit in 2003 with *crash-only software*, which has only one way to stop (crashing) and one way to start (recovery), so that the recovery path is exercised constantly rather than discovered during an incident.

IF YOU ARE NEW TO THIS

Step by step: a worker that is told to stop

The idea that needs the most care here is disposability: a process must expect to be stopped at any moment, politely or not, and lose nothing when it is. Follow one worker process through both kinds of stop.

- The food-delivery platform runs a worker process that takes messages from a queue and sends order confirmations. During a routine deployment, every old copy of the worker is replaced by a new one, so each old copy must stop.
- The polite way is a signal called SIGTERM, meaning 'please finish up and exit', as Section 10.2 described. The platform waits for a grace period before forcing the process to end.
- A disposable worker reacts at once. It stops taking new messages, either finishes the confirmation it is sending or hands that message back to the queue for another copy to pick up, and exits. Nothing is lost.
- But a process can also die with no warning: its machine fails, or it is killed for using too much memory. There is no chance to tidy up.
- Suppose the worker dies just after sending a confirmation but before telling the queue it had finished. The message is offered again and another copy handles it again. So handling must be idempotent, as Chapter 5 put it: doing it twice has the same effect as doing it once. This usually works by giving each message a unique key, passed on with the confirmation, so that a repeat is recognised and skipped.
- **The lesson:** treat stopping and crashing as everyday events. A process that starts quickly, stops cleanly when asked and loses nothing when it dies without warning can be replaced, moved and scaled freely.

Dev/prod parity, logs and admin processes

The original text identifies three gaps between development and production: time (code written weeks before it is deployed), personnel (developers write it, operators deploy it) and tools (SQLite on a laptop, PostgreSQL in production). Continuous delivery closes the first, teams that run what they build close the second, and containers close much of the third: a developer can run the same PostgreSQL version, message broker and cache locally, and tools such as Testcontainers start real dependencies in integration tests. Perfect parity is neither affordable nor necessary; what matters most is using the same kinds and versions of backing services, because that is where the surprising differences live.

Logs as event streams means that the application writes each event to standard output and leaves storage, rotation and shipping to the platform. That remains right, with two modern additions: logs should be structured (for example JSON with consistent field names and a

request or trace identifier), and logs are one of three signals, alongside metrics and traces, that make a system observable (observability). Log volume is also a real cost driver; Section 10.8 returns to it.

Admin processes, such as database migrations, data repairs and one-off reports, should run with the same release (code and configuration) as the application, as separate processes. In Kubernetes that typically means a Job run by the pipeline before the new version rolls out. Migrations need particular care: they must be compatible with both the old and the new version of the code running side by side, which is the expand-and-contract discipline of schema evolution.

Beyond twelve factors

The twelve factors were never meant to be complete, and later authors extended them. Kevin Hoffman's *Beyond the Twelve-Factor App* (O'Reilly, 2016) revised the list to fifteen, adding *API first* (design the contract before the implementation), *telemetry* (build in metrics, health and tracing) and *authentication and authorisation* (security as a design concern from the start). Current practice adds four themes that every cloud-native application should meet.

Disposability, taken seriously. Design for being killed at any moment: idempotent operations, short request timeouts, bounded start-up time, graceful drain, and no single process whose loss matters. **Immutability.** Never patch a running container or server; build a new image and replace it (Section 10.4 on immutable infrastructure). This makes every running instance traceable to a commit and removes drift. **Observability.** Emit structured logs, metrics and traces through a standard such as OpenTelemetry, expose health endpoints that reflect the process's own state, and propagate trace context across calls, so that operators can answer questions they did not anticipate. **Security.** Run as a non-root user with a read-only file system where possible; obtain credentials through workload identity rather than long-lived keys; apply least privilege to every backing service; and build from minimal, scanned, signed images (supply-chain security).

WORKED EXAMPLE

Worked example: auditing the restaurant-onboarding service

The food-delivery platform's restaurant-onboarding service is a five-year-old web application on two hand-built virtual machines behind a load balancer with sticky sessions. The team wants to move it to the company's container platform. An audit against the factors finds nine problems; each is given a risk, a fix and a priority.

1. Production database password in a properties file in the repository (III)	Risk: anyone with read access to the code, including former contractors, has the credential. Fix: rotate the password now, move it to the secrets manager, inject at deploy time. **Priority 1**.
2. Uploaded menus and logos written to the VM's local disk (VI, IV)	Risk: files vanish when a request lands on the other VM or a VM is replaced; blocks any move to disposable containers. Fix: object storage, with a one-off copy of existing files. **Priority 1**.
3. Sessions in process memory, sticky load balancing (VI)	Risk: every deployment logs restaurant staff out mid-form. Fix: signed session tokens or a shared session store. **Priority 2**.
4. Deployed by SSH, `git pull` and restart on each VM (V, II)	Risk: the two VMs run different code after a failed pull; no rollback. Fix: CI builds one image per commit; numbered releases. **Priority 2**.
5. Migrations run automatically at start-up by both instances (XII)	Risk: two instances race on the same migration. Fix: a separate migration step in the pipeline, with expand-and-contract changes. **Priority 2** (Priority 1 if a race has already happened).
6. Killed with `kill -9` during deploys (IX)	Risk: half-saved onboarding forms. Fix: handle SIGTERM, drain requests, add readiness and a pre-stop delay. **Priority 2**.
7. Logs written to rotating files on each VM (XI)	Risk: logs lost with the VM; hard to search across instances. Fix: structured JSON to standard output, collected centrally. **Priority 3**, cheap to do alongside item 4.
8. Nightly payout report as a cron entry on VM 1 only (XII, VIII)	Risk: the job silently stops when VM 1 is rebuilt, or runs twice if copied. Fix: a scheduled job on the platform, made idempotent. **Priority 3**.
9. Developers use SQLite, production uses PostgreSQL (X)	Risk: SQL that works locally fails in production. Fix: PostgreSQL in a container for development and tests. **Priority 3**.

Priorities follow risk first (an exposed credential is a live incident), then whatever blocks the target platform (local state), then reliability, then convenience.

The team does not need to fix everything before migrating: items 1 and 2 are prerequisites; items 3 to 6 can be done during the move; items 7 to 9 can follow.

An audit like this is also good material for an architecture decision record: it explains why each change is being made, in what order, and what was deliberately deferred.

WATCH OUT

Common misreadings of the twelve factors

- Stateless does not mean the system has no state; it means processes do not keep it. Moving state to a single, unreplicated cache just moves the problem.
- Twelve-factor is not a definition of microservices. It applies equally to a well-built monolith, and a set of services can violate it.
- Putting secrets in environment variables satisfies the letter of factor III but not current security practice; prefer a secrets manager and workload identity.
- Dev/prod parity does not mean a full copy of production for every developer; it means the same kinds and versions of dependencies, and short gaps in time and people.
- Port binding and logs to standard output are easy wins; teams sometimes do these and claim compliance while still keeping files on local disk.

Questions people ask

Does every application need to satisfy all twelve factors?
No. They are a checklist of properties that make deployment, scaling and recovery cheap on a modern platform. Treat each gap as a risk with a cost, and fix those that block your platform or threaten security first, as in the audit above.

Does the methodology apply to serverless functions?
Largely, and the platform enforces most of it: functions are stateless, configured from outside, disposable and logged by the platform. The factors that still need attention are config and secrets, backing services, idempotency and dev/prod parity, since local emulation of a provider's event sources is imperfect.

Is twelve-factor still relevant now that Kubernetes exists?
Yes, because Kubernetes assumes it. A pod that keeps sessions in memory, writes files locally or takes two minutes to start fights the platform's rolling updates, autoscaling and rescheduling. The factors describe what an application must do so that the platform's automation helps rather than hurts.

CHECK YOUR UNDERSTANDING

▼ What simple test shows whether configuration has leaked into the code?

Ask whether the codebase could be made public today without exposing any credential. If not, configuration (at least the secrets) lives in the code.

▼ Why does build, release, run make rollback fast?

Each release is an immutable combination of a built artefact and configuration with its own identity, so rolling back means running the previous release again, without rebuilding or editing anything.

▼ A service stores uploaded files on the local disk of its server. Which factor does this violate and what breaks?

Processes (VI), and arguably backing services (IV): files are missing when requests reach another copy, and are lost when the instance is replaced, which blocks scaling out and disposable containers.

▼ Why should migrations not run automatically in every replica at start-up?

Several replicas may run the same migration at once and race, and start-up becomes slow and fragile. Migrations should run once, as a separate admin step in the pipeline, using the same release.

▼ Try it: Auditing a service, you find: a production password in the repository; uploaded menus saved on local disk; logs written to a file; one-off admin scripts run by logging into a server. Rank these for a container platform move.

First, the password: it is exposed now, so rotate it and move it to a secrets manager or workload identity. Second, local files, because they block the move: containers are disposable, so uploads go to object storage as a backing service. Third, one-off scripts: run them as separate admin processes from the same release, so they are repeatable and audited. Fourth, logs: write to stdout as a stream. Risk comes first, then blockers, then convenience.

SOURCES

- The Twelve-Factor App: https://12factor.net/
- The Twelve-Factor App: III. Config: https://12factor.net/config
- Twelve-Factor App methodology is now open source (2024): https://www.12factor.net/blog/open-source-announcement

- Kubernetes documentation: Configure liveness, readiness and startup probes: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

SECTION 10.4 OF 10.8 · ADVANCED · 26 MIN READ · LAST REVIEWED NOVEMBER 2026

Infrastructure as code and immutable infrastructure

BEFORE YOU READ THIS

This section builds on:

- 0.7 Servers and the cloud
- 0.8 How software gets built
- 10.2 Containers and Kubernetes

Terms it uses from earlier: Desired state, Reconciliation loop, Pull request, Blast radius

THE IDEA IN BRIEF

Imagine furnishing a hotel with two hundred identical rooms. You could tell each decorator what to do and hope they remember, or you could write one precise specification for a room, keep it in a binder, and have every change approved in writing. With the specification, any room can be checked against it, a damaged room can be rebuilt exactly, and a new wing can be fitted out in days. Infrastructure as code is that specification for the computers, networks and databases a system runs on: written down, reviewed, versioned, and applied by machines rather than from memory.

WHAT YOU NEED TO KNOW

- Infrastructure as code defines infrastructure in version-controlled files applied by tools, making environments reproducible, reviewable, auditable and quick to create.
- Declarative tools describe the desired end state and compute the steps, which makes repeated runs idempotent; imperative scripts list steps and are hard to rerun safely.
- HashiCorp moved Terraform to the Business Source License in August 2023; the OpenTofu fork followed under the Linux Foundation (stable in January 2024, a CNCF sandbox project since 2025); CloudFormation, Bicep, Pulumi and Crossplane are the main alternatives.
- State maps code to real resources; it must be remote, locked, versioned, access-controlled and split to limit blast radius, and drift must be detected by scheduled plans and then codified or reverted.
- Versioned modules with small interfaces, reused across environments with different inputs, prevent copy-and-paste environments from diverging.
- Immutable infrastructure replaces servers and containers from new images instead of changing them in place; configuration management remains useful for long-lived estates and for building images.

- GitOps keeps desired state in Git and lets in-cluster agents (Argo CD, Flux) pull and continuously reconcile it; policy as code and layered tests make changes safe.

What infrastructure as code is

Infrastructure as code (IaC) means defining networks, virtual machines, clusters, databases, queues, DNS records, permissions and every other piece of infrastructure in text files kept in version control, and letting tools create and change the real thing from those files. Instead of an engineer clicking through a cloud console, a change is a commit: reviewed in a pull request, tested in a pipeline, applied automatically and recorded for ever.

The idea is older than the public cloud. Mark Burgess's CFEngine (1993) managed Unix machines from declarative policies; Puppet (2005), Chef (2009) and Ansible (2012) brought configuration management to the mainstream. The cloud made it essential, because infrastructure itself became available through APIs: AWS CloudFormation (2011) and HashiCorp's Terraform (2014) describe whole environments, not just the software on servers. Kief Morris's book *Infrastructure as Code*, now in its third edition (2025), is the standard account of the practice and its principles.

The benefits are those of software engineering applied to infrastructure. Environments become reproducible: staging can be built from the same definitions as production, and a region can be rebuilt after a disaster from code rather than from memory. Changes become reviewable and auditable, which regulators and security teams value. Knowledge stops living in one engineer's head. And speed rises, because creating an environment for a new food-delivery service becomes a pull request rather than a ticket that waits two weeks.

Declarative versus imperative

An *imperative* approach lists the steps: create a network, then a subnet, then a database, then open this port. Running the script twice tries to create everything twice. A *declarative* approach describes the end state, the desired state of Section 10.2: a network with these subnets, a database of this size, a firewall allowing this port. The tool compares the description with what exists and works out the steps, so applying it twice changes nothing the second time. That property, *idempotence*, is what makes infrastructure code safe to run repeatedly and in automation.

Most mature tools are declarative at their core, but they differ in the language used to write the declaration and in who keeps track of what exists. The table summarises the main families.

Approaches to infrastructure code

Approach	You write	Examples	Strengths	Weaknesses
Imperative scripts	Sequences of CLI or SDK calls	Shell scripts calling cloud CLIs, SDK programs	Familiar, flexible, quick for one-off tasks	Not idempotent without great care; no record of what exists; hard to change safely
Declarative, domain-specific language	Resources and their relationships in a configuration language	Terraform and OpenTofu (HCL), AWS CloudFormation (YAML or JSON), Azure Bicep, Kubernetes manifests	Readable, reviewable plans; idempotent; large provider ecosystems	Loops, conditions and abstraction can be awkward; a language to learn
Declarative model, general-purpose language	Code in TypeScript, Python, Go, C# or Java that builds a declarative model	Pulumi, AWS Cloud Development Kit (CDK)	Real abstractions, testing tools and editors; familiar to developers	Easy to write clever, hard-to-review infrastructure; the plan is less visible in the code
Continuous reconciliation	Kubernetes-style resources describing cloud infrastructure	Crossplane, cloud providers' Kubernetes controllers	Same reconciliation loop and tooling as applications; drift corrected continuously	Needs a Kubernetes control plane; debugging spans two systems
Configuration management	Desired configuration of software on existing servers	Ansible, Chef, Puppet	Manages long-lived machines, network devices and on-premises estates	Servers accumulate history and drift; slower and less predictable than replacing them

The tools and the OpenTofu fork

Terraform became the most widely used multi-cloud tool because one language and workflow covers thousands of providers: AWS, Azure, Google Cloud, Cloudflare, Kubernetes, GitHub, monitoring services and more. On 10 August 2023 HashiCorp moved Terraform from the open-

source Mozilla Public License 2.0 to the Business Source License 1.1, which permits internal and commercial use but forbids offering it in products that compete with HashiCorp's own. Version 1.5.7 was the last release under the old licence.

The community's response was a fork. A manifesto calling for a return to open source appeared within days; in September 2023 the Linux Foundation accepted the fork as OpenTofu; its first stable release, 1.6.0, followed in January 2024; and in 2025 it was accepted into the CNCF as a sandbox project. IBM completed its acquisition of HashiCorp in February 2025. The two tools remain largely compatible in language and providers but are diverging in features: OpenTofu added client-side encryption of state, for example, while Terraform's commercial platform added its own capabilities. For an architect, the episode is a reminder that a tool's licence and governance are part of its risk profile, and that open standards and a credible alternative keep that risk manageable.

Among the single-cloud tools, AWS CloudFormation (2011) and Microsoft's Bicep (introduced in 2020, compiling to Azure Resource Manager templates) are run by the provider, which keeps track of what exists, so there is no state file to manage, at the price of being tied to one cloud. Pulumi (2018) and the AWS CDK let developers write infrastructure in general-purpose languages; Crossplane extends Kubernetes to manage cloud resources by reconciliation. The right choice depends on which clouds you use, your team's skills, the maturity of the providers you need, licence policy and what the rest of the organisation already runs; consistency across teams is usually worth more than the marginal merits of any one tool.

```
terraform {
  required_version = "≥ 1.6"
  backend "gcs" {                 # remote state; this backend locks natively
    bucket = "fd-infra-state"
    prefix = "orders/prod"
  }
}

variable "environment" { type = string }

module "orders_db" {
  source      = "git::https://git.example.com/modules/postgres.git?ref=v3.2.0"
  name        = "orders-${var.environment}"
  size        = var.environment == "prod" ? "large" : "small"
  high_avail  = var.environment == "prod"   # standby in a second zone
  backup_days = 14
}

output "orders_db_host" {
  value = module.orders_db.host
}
```

OpenTofu or Terraform (HCL): remote state, and a versioned in-house module that hides provider details behind a small interface.

Running `tofu plan` (or `terraform plan`) shows what would change before anything happens. The plan is the most important artefact in the workflow: reviewers read it in the pull request, alongside the code, and it is what gets applied.

```
  # module.orders_db.google_sql_database_instance.this will be updated in-place
  ~ resource "google_sql_database_instance" "this" {
      ~ settings {
          ~ tier = "db-custom-2-8192" → "db-custom-4-16384"
        }
    }

Plan: 0 to add, 1 to change, 0 to destroy.
```

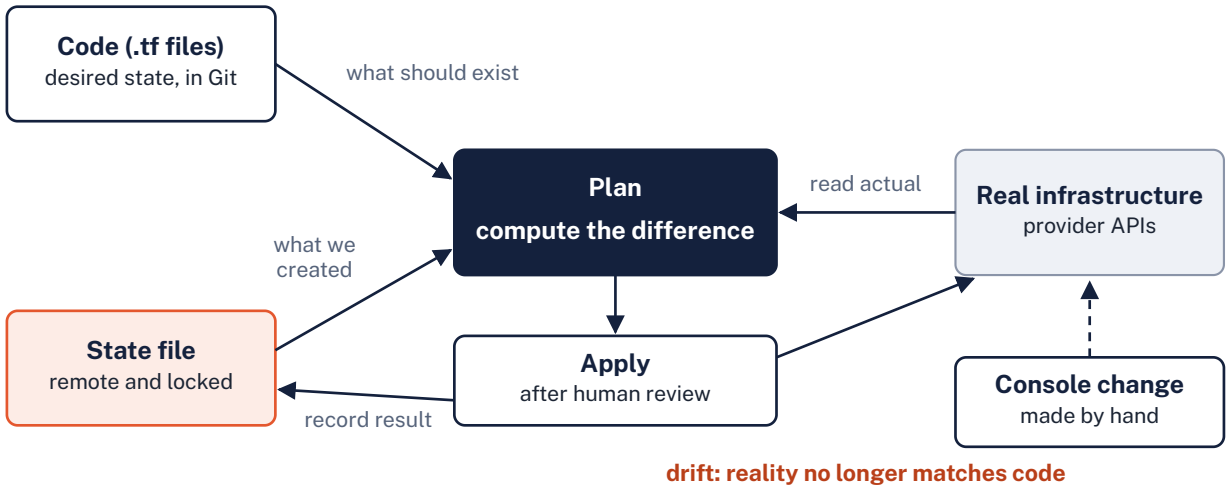
An abbreviated plan for resizing the production orders database. The tilde marks an in-place update; a -/+ would mean destroy and recreate, which for a database is a red flag.

State, locking and drift

Declarative tools need to know which real resources belong to which definitions. Terraform and OpenTofu keep this in state: a file mapping each resource in the code to the identifier of the real object, with its last known attributes. Planning compares three things: the code (what

should exist), the state (what the tool created) and the real infrastructure, read from the provider's API (what actually exists).

Plan, apply, state and drift



A plan compares code, state and reality; a manual change in the console makes reality diverge from both, and the next plan will try to undo it.

State needs care because it is critical and sensitive. It must be stored remotely (in object storage or a managed service), not on a laptop, so that the whole team and the pipeline share one version. It must be *locked* during changes, so that two applies cannot interleave and corrupt it. It must be backed up and versioned, because a lost state file leaves the tool unaware of resources it created. And it is a secret: state often contains database passwords and keys in plain text, so access to it must be restricted, and encrypted at rest (OpenTofu can also encrypt it on the client side).

Drift is any difference between the code and reality, usually from manual changes made in a hurry, from another tool touching the same resource, or from a partial failure during apply. Drift is dangerous because the next apply may silently revert a change someone depended on, or fail in production on a difference nobody knew existed. Teams detect it by running a plan on a schedule (for example nightly) and alerting on any proposed change, and prevent it by restricting console write access to a recorded break-glass procedure.

State also defines *blast radius*. One giant state for a whole company means every plan is slow, every apply risks everything, and one lock blocks every team. Splitting state by component and environment (network, shared cluster, each service's resources; production separate from staging) keeps plans fast and failures contained, at the price of passing outputs such as

network identifiers between states. Refactoring is supported: `moved` blocks rename resources without destroying them, and `import` brings existing hand-made resources under management.

WORKED EXAMPLE

Worked example: a drifted firewall rule

During a Friday-night incident, an engineer opens the production orders database's firewall to a contractor's IP address through the cloud console, so that the contractor can help diagnose a slow query. Nobody updates the code.

Detection	The nightly drift job runs `tofu plan` for every production state. On Saturday it reports *1 to change* for the orders database firewall and posts the plan to the platform team's channel.
Diagnosis	The plan shows an extra allowed source address. The cloud audit log identifies who added it, when and from which incident.
Decision	The access was temporary and the incident is closed, so the code is right and reality is wrong. Applying the plan removes the rule. Had the change been needed permanently, the fix would be to add it to the code through a reviewed pull request instead.
Prevention	Console write access in production is replaced by a break-glass role that requires a reason and alerts on use. A policy rule (see below) rejects any database firewall rule open to the whole internet.
What it shows	Drift is not only a tidiness problem: an undocumented hole in a firewall is a security exposure, and without detection the next unrelated apply would have removed it without anyone knowing why the contractor lost access.

The rule of thumb: reality may differ from the code for minutes during an emergency, but the difference must be either codified or reverted by the next working day.

IF YOU ARE NEW TO THIS

Step by step: why the tool keeps a record

The step most people trip over here is why tools such as OpenTofu and Terraform keep state. The code says what should exist, but not which real resource in the cloud account it refers to. Follow the production orders database through its life.

- The team describes orders-prod in code, using the in-house database module. Nothing exists yet, so the state is empty and the first plan proposes adding it.
- On apply, the tool asks the provider to create the database. The provider gives it an identifier, and the tool records in the state that this resource in the code is the real database with that identifier, along with its settings.
- Months later the team changes the size in the code. The plan reads the code, finds in the state which real database that is, asks the provider for its current settings and proposes one in-place change.
- Without the state, the tool would have no record that this database belongs to this code, and would propose creating it again.
- The state also reveals what the code no longer mentions. If the team deletes the module from the code, the plan proposes destroying the database that the state still lists.
- That is why state is shared, locked, backed up and kept private: two applies at once could corrupt it, losing it would leave the tool unaware of what it created, and it often holds passwords.
- **The lesson:** the code says what should exist, the real infrastructure is what does exist, and the state links the two. Read every plan with all three in mind, and treat a plan to destroy a database as a warning sign.

Modules and environments

A module is a reusable unit of infrastructure code with inputs and outputs: a PostgreSQL database with backups and monitoring, a service's network and load balancer, a standard Kubernetes namespace with its quotas and policies. A good module applies the information hiding of Chapter 3: a small interface (name, size, high availability or not) in front of the many provider-specific details a service team should not have to know. Modules are versioned and released like libraries, and environments refer to a specific version, as the `ref=v3.2.0` in the example does, so that a module change reaches production only when someone deliberately upgrades it.

Environments should be built from the same modules with different inputs, not from copied code. A common layout is one directory (or repository) per environment and component, each with its own state, calling shared, versioned modules; a change is promoted by updating the module version or inputs in staging, then in production. Copy-and-paste environments diverge within months, until staging no longer predicts production. Workspaces, which give one configuration several states, suit short-lived copies such as per-branch test environments better than long-lived production and staging.

Beware of over-abstraction. A module with forty boolean inputs that tries to serve every team is as hard to change as the code it replaced. Prefer a few opinionated modules that encode the organisation's standards, which is exactly what a platform team offers as a golden path in Section 10.7.

Immutable infrastructure versus configuration management

Configuration management tools keep long-lived servers in a described state: Ansible connects over SSH and pushes changes without an agent; Chef and Puppet run agents that pull their configuration and converge on it periodically. They work, but every server accumulates history: packages installed during an incident, files edited by hand, updates that half-applied. Martin Fowler called the result a *snowflake server*, unique and impossible to reproduce, and proposed the *phoenix server*, regularly burnt down and rebuilt from scratch, as the cure.

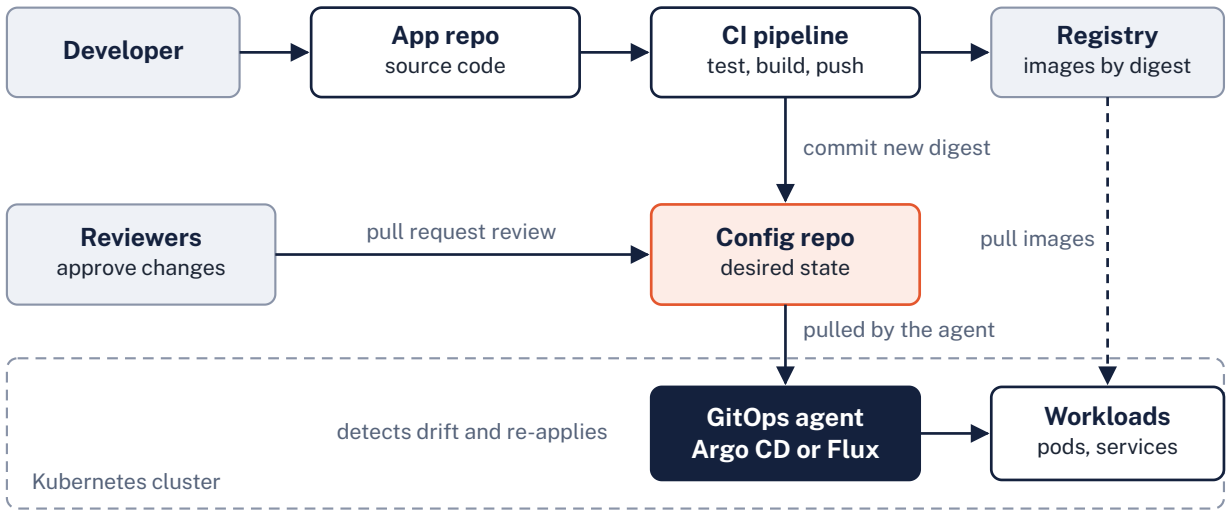
Immutable infrastructure, a term popularised by Chad Fowler in 2013, takes the cure to its conclusion: never change a running server. Every change, whether an application release or an operating system patch, is made by building a new image (a container image, or a machine image with a tool such as Packer), testing it and replacing the old instances, typically by a rolling or blue-green deployment. Containers on Kubernetes are immutable by default. The benefits are no drift, identical instances, trivially repeatable environments and rollback by redeploying the previous image; the costs are an image pipeline, the need for disposable, stateless processes (Section 10.3) and slower turnaround for a single urgent change.

Configuration management has not disappeared. It remains the right tool for estates that cannot be replaced wholesale: on-premises servers, network equipment, databases on dedicated hardware, and the base images themselves, which are often built by running Ansible inside the image build. The architectural rule is to push mutability as far down and as early as possible: configure while building images, then treat what runs as read-only.

GitOps

GitOps applies the reconciliation loop of Section 10.2 to whole environments. The term was coined in 2017 by Alexis Richardson of Weaveworks, the company behind the Flux tool. The OpenGitOps project, a CNCF working group, defines four principles: the desired state is *declarative*; it is *versioned and immutable*, with a complete history; it is *pulled automatically* by software agents; and it is *continuously reconciled*, with agents observing the actual state and applying the desired one. The two leading agents, Argo CD and Flux, are both graduated CNCF projects; Flux continued as a community project after Weaveworks closed in 2024.

A GitOps delivery flow



CI builds and publishes images; Git holds the desired state; an agent inside the cluster pulls it, applies it and keeps correcting drift.

The pull model has a security advantage over a pipeline that pushes changes into the cluster: the CI system needs no credentials for production, because the agent inside the cluster fetches what it should run. Git becomes the audit log and the interface for change, including for operations: scaling a service or rolling back is a reviewed commit or a revert. Drift is corrected automatically, so a manual `kubectl edit` lasts only until the next sync.

GitOps brings its own design questions. Secrets must not sit in Git in plain text; teams encrypt them (SOPS, Sealed Secrets) or store references that the External Secrets Operator resolves from a secrets manager. Promotion between environments needs a convention, such as one directory per environment with pull requests that copy a new image digest from staging to production. Automatic drift correction must be paused deliberately during some emergencies, and the repository structure must reflect ownership so that teams can change their own

services without touching others'. GitOps is most natural for Kubernetes, but the same principles apply to cloud infrastructure through Crossplane or through pipelines that plan and apply infrastructure code from merged commits.

Policy as code

Policy as code expresses rules about infrastructure as machine-checkable code instead of a wiki page: storage buckets must not be public, databases must be encrypted and backed up, every resource must carry an owner and cost-centre tag, only approved regions may be used, containers must not run as root, production images must come from the company registry. Checks run at several points: on the plan in the pull request (Open Policy Agent with Conftest, Checkov, HashiCorp Sentinel in its commercial products), at admission into a Kubernetes cluster (OPA Gatekeeper, Kyverno) and continuously against running resources (cloud providers' policy services).

Policy as code turns governance from a review meeting into a fast, consistent guardrail that explains itself in the pull request. It works best when policies are few, clearly justified, versioned and tested like any code, when they start in warning mode before they block, and when there is a recorded exception process. Chapter 11 discusses how such automated checks fit into architecture governance (governance in agile organisations).

Testing infrastructure code, from fast to slow

1. **Static checks on every commit**
   Formatting, `validate`, linting (for example TFLint) and security scanners catch syntax errors, deprecated arguments and obvious misconfigurations in seconds.

2. **Policy checks on the plan**
   Render the plan as JSON and evaluate it against the organisation's policies; reviewers see violations in the pull request.

3. **Module tests**
   `terraform test` and `tofu test` run assertions against a module, with mocked providers for fast unit-style tests or real ones for integration tests.

4. **Integration tests in ephemeral environments**
   Apply the module or environment to a throwaway account, check that it works (a database accepts connections, a service answers), then destroy it. Tools such as Terratest script this; it is slow and costs money, so reserve it for shared modules.

5 **Plan review and staged rollout**
Apply to staging first, then production, from the same versions; a human reads every production plan, especially destroys and replacements.

6 **Drift detection and restore drills**
Scheduled plans catch drift; periodically rebuilding an environment from code proves that the code, not memory, is the source of truth.

Checklist: infrastructure code ready for production

Use it when reviewing a team's infrastructure repository or a new platform module.

- [ ] **Everything is in code**
  No production resource exists only because someone clicked; existing hand-made resources have been imported.
- [ ] **Remote, locked, encrypted state**
  Shared backend with locking, versioning and restricted access; state split by component and environment to limit blast radius.
- [ ] **Reviewed plans**
  Every change is a pull request showing the plan; destroys and replacements of stateful resources require explicit approval.
- [ ] **Versioned modules**
  Environments pin module versions; staging and production use the same modules with different inputs, not copies.
- [ ] **Policies enforced automatically**
  Encryption, network exposure, tagging and region rules are checked on every plan and at cluster admission.
- [ ] **Secrets kept out of code and Git**
  Credentials come from a secrets manager or workload identity; state access is treated as secret access.
- [ ] **Tests in the pipeline**
  Static checks and policy checks on every change; module tests for shared modules; integration tests for critical ones.
- [ ] **Drift detected and handled**
  Scheduled plans alert on differences; console write access is a recorded break-glass procedure.
- [ ] **Immutable where possible**
  Servers and containers are replaced from images rather than patched in place; configuration happens at image build time.
- [ ] **Recovery rehearsed**
  An environment has been rebuilt from code within the agreed recovery time, including data restore (see Section 10.6).

CHECK YOUR UNDERSTANDING

▼ Why is idempotence important for infrastructure code?

Because automation reruns code often. A declarative, idempotent tool changes nothing when reality already matches the code, so rerunning it is safe; a non-idempotent script may try to create everything again.

▼ Name three things a plan compares, and what each represents.

The code (what should exist), the state (what the tool created and tracks) and the real infrastructure read from the provider's API (what actually exists).

▼ Why should state not be kept as one file for a whole organisation?

One state means slow plans, a single lock that blocks every team and a blast radius covering everything; splitting by component and environment keeps changes fast and failures contained.

▼ What security advantage does the GitOps pull model have over a pipeline that pushes to the cluster?

The CI system needs no production credentials, because an agent inside the cluster pulls the desired state from Git and applies it.

▼ When is configuration management still a good choice?

For long-lived machines that cannot simply be replaced, such as on-premises servers and network devices, and for configuring base images during the image build.

▼ Try it: A team keeps its infrastructure state file on one engineer's laptop, with all environments in one state, and created staging by copying the production code and editing it. List the risks, and say how you would restructure this.

The state could be lost, nobody else can safely run changes, and two people could collide. One state also means one mistake can damage every environment. Copied code will drift. Move state to a remote, locked, versioned, access-controlled store, split by environment and component to limit blast radius. Build versioned modules with small interfaces, and use them for each environment with different inputs. Add scheduled plans to detect drift.

SOURCES

- OpenTofu: https://opentofu.org/
- Terraform documentation: State: https://developer.hashicorp.com/terraform/language/state

- Terraform documentation: Tests: https://developer.hashicorp.com/terraform/language/tests
- OpenGitOps: GitOps principles: https://opengitops.dev/
- Martin Fowler: PhoenixServer: https://martinfowler.com/bliki/PhoenixServer.html

SECTION 10.5 OF 10.8 · INTERMEDIATE · 25 MIN READ · LAST REVIEWED NOVEMBER 2026

CI/CD pipelines and deployment strategies

BEFORE YOU READ THIS

This section builds on:

- 0.8 How software gets built
- 2.6 Modifiability and deployability
- 7.7 Schema evolution

Terms it uses from earlier: Container image, Expand and contract, Service level objective (SLO), Distributed monolith

THE IDEA IN BRIEF

Think of a bakery chain launching a new loaf. It bakes one test batch, checks it in the bakery's own kitchen, then sells it in one shop for a morning before sending it to every shop. If customers complain, only one shop's customers were affected and the old loaf is still on the shelves. A software pipeline is the same routine, automated: each change is built once, checked in stages, tried on a few users, and spread further only when the evidence says it is good. A separate switch decides when customers actually see a new feature.

WHAT YOU NEED TO KNOW

- A deployment pipeline (Humble and Farley, 2010) runs every commit through automated stages, fastest first, and a failure at any stage stops that change and notifies its author.
- Build once, deploy many: package each change into one immutable artefact, store it in an artefact repository and promote that same artefact through every environment, changing only configuration.
- Trunk-based development keeps branches short-lived and changes small, which makes continuous integration real and every change easier to review, test and undo.
- Recreate, rolling, blue-green, canary and shadow deployments trade extra infrastructure and complexity for less user exposure and faster rollback; all but recreate require old and new versions to be compatible.
- Feature flags separate deployment from release; Hodgson's categories are release, experiment, ops and permissioning toggles, and every flag carries a cost until it is removed.
- Database changes must be backward-compatible (expand and contract) and run as separate pipeline steps, because data cannot be rolled back like code.

- Progressive delivery automates canary decisions by comparing the canary statistically with a fresh baseline; the default response to a bad release is to roll back first and diagnose later.

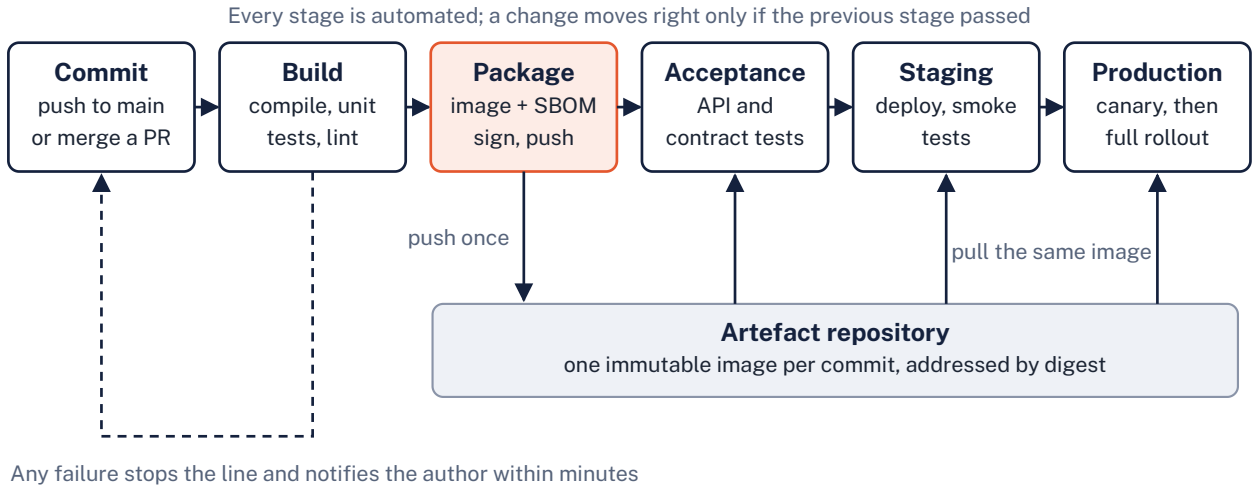
From commit to production: what a pipeline is for

Section 0.8 (how software gets built) introduced continuous integration (every change is merged and verified automatically), continuous delivery (the main branch is always releasable) and continuous deployment (every change that passes the checks goes to production without a human decision). This section treats them as architecture: the design of the path a change takes, and the strategies that move it into production without hurting users.

The central idea is the deployment pipeline, described by Jez Humble and David Farley in *Continuous Delivery* (2010). Every commit triggers an automated sequence of stages. Early stages are fast and cheap and catch most problems; later stages are slower, more realistic and more expensive. A change moves to the next stage only if the previous stage passed, so the further a build gets, the more confidence it has earned. A failure anywhere stops that build and goes straight back to the person who made the change, while it is still fresh in their mind.

The pipeline matters to architects for three reasons. First, it is where deployability, a quality attribute from section 2.6, is actually achieved or lost. Second, its speed bounds how quickly the organisation can respond to a market change or a security vulnerability: if a one-line fix takes two days to reach production, the architecture is two days slow, whatever the code looks like. Third, the shape of the system constrains the shape of the pipeline: independently deployable services can have independent pipelines, but a distributed monolith forces coordinated releases however clever the tooling.

The stages of a typical pipeline



One artefact is built and pushed once; every later stage deploys exactly that artefact, and any failure stops the line.

What each stage does and how long it should take

1. **Commit stage (target: under 10 minutes)**
   Compile, run unit tests, linters, static analysis and dependency vulnerability scans. This is the stage developers wait for, so it must be fast; slow tests belong later.

2. **Package**
   Build the deployable artefact (usually a container image), generate a software bill of materials (SBOM), sign it and push it to the artefact repository. Supply-chain controls are covered in section 8.6.

3. **Acceptance and integration**
   Deploy the artefact to a temporary or shared test environment and run API tests, consumer-driven contract tests and a few end-to-end journeys. Performance and security tests may run here in parallel or on a schedule.

4. **Staging**
   Deploy to an environment configured like production, apply database migrations, run smoke tests. Its value depends on how closely it matches production in configuration, data shape and dependencies.

5. **Production**
   Deploy with a strategy that limits exposure (rolling, blue-green or canary), verify with real metrics against service level objectives, then complete or roll back automatically.

Build once, deploy many, and the artefact repository

A rule from *Continuous Delivery* that is still broken surprisingly often is build once, deploy many: compile and package each change exactly once, then promote that same artefact through every environment. If a team rebuilds for staging and again for production, the thing it tested is not the thing it shipped. A different compiler patch, a dependency resolved to a newer version overnight or a changed base image can make the two builds differ, and the tests prove nothing about the second one.

What differs between environments is configuration, supplied at deployment or start-up time (database addresses, credentials, feature-flag settings), never baked into the artefact. This is the configuration factor of the twelve-factor app. A container image tagged `orders:4f1c2ab` (the commit hash) is deployed to test, staging and production; only its environment variables and mounted secrets change.

The artefact repository is the store between build and deploy: a container registry (Amazon ECR, Azure Container Registry, Google Artifact Registry, GitHub Container Registry, Harbor) or a universal repository such as JFrog Artifactory or Sonatype Nexus for packages and binaries. Treat it as critical infrastructure. Deploy by immutable digest (`sha256:...`) rather than by a mutable tag such as `latest`, which can be moved to point at a different image; keep artefacts that are or were in production for as long as you might need to roll back to them or investigate them; and record provenance, meaning which commit, pipeline run and dependencies produced each artefact.

Trunk-based development and small batches

How code is branched decides how often it is really integrated. In trunk-based development everyone integrates small changes into one main branch (the trunk) at least daily, either by committing directly or through short-lived branches and pull requests that live for hours, not weeks. The DORA research programme has repeatedly associated short-lived branches and frequent merging with higher delivery performance, and it is a precondition for continuous integration in the literal sense: code on a branch for three weeks is not being integrated continuously.

Long-lived feature branches feel safer because unfinished work is kept apart, but the risk is merely postponed: the longer two branches diverge, the larger and riskier the eventual merge, and the longer the pipeline sees code that is not the code that will ship. GitFlow, a branching model with long-lived develop and release branches published by Vincent Driessen in 2010, was widely adopted; in 2020 Driessen added a note that teams delivering continuously, such as web applications, should consider a simpler workflow. Teams that do need to work on large

changes in the trunk use two techniques: feature flags, which hide unfinished behaviour, and branch by abstraction, which introduces an interface in front of the old implementation, builds the new one behind it and switches over when ready.

Practitioners genuinely disagree at the edges. Open-source projects with many untrusted contributors rely on longer-lived pull requests and review, and regulated teams sometimes need explicit release branches for audit. The principle that survives the disagreement is the batch size: the smaller each change, the easier it is to review, test, deploy, diagnose and undo.

```yaml
name: delivery
on:
  push:
    branches: [main]
env:
  IMAGE: registry.example.com/food/orders
jobs:
  build:
    runs-on: ubuntu-latest
    outputs:
      digest: ${{ steps.push.outputs.digest }}
    steps:
      - uses: actions/checkout@v4
      - run: make lint test              # fast feedback first
      - id: push
        run: |
          docker build -t "$IMAGE:${{ github.sha }}" .
          docker push "$IMAGE:${{ github.sha }}"
          ref=$(docker inspect --format '{{index .RepoDigests 0}}' "$IMAGE:${{ github.sha
}}")
          echo "digest=${ref#*@}" >> "$GITHUB_OUTPUT"   # the image's sha256 digest
  staging:
    needs: build
    runs-on: ubuntu-latest
    environment: staging
    steps:
      - uses: actions/checkout@v4
      - run: ./deploy.sh staging "$IMAGE@${{ needs.build.outputs.digest }}" && ./smoke-
test.sh staging
  production:
    needs: [build, staging]
    runs-on: ubuntu-latest
    environment: production            # can require an approval
    steps:
      - uses: actions/checkout@v4
      - run: ./deploy.sh production "$IMAGE@${{ needs.build.outputs.digest }}" --canary 5
```

A minimal continuous delivery workflow in GitHub Actions syntax: build and test once, push one image tagged with the commit, then deploy that exact image by its digest to staging and then to production as a canary. GitLab CI, Azure Pipelines, Jenkins and others express the same stages. Registry login and the deploy scripts are omitted.

Deployment strategies

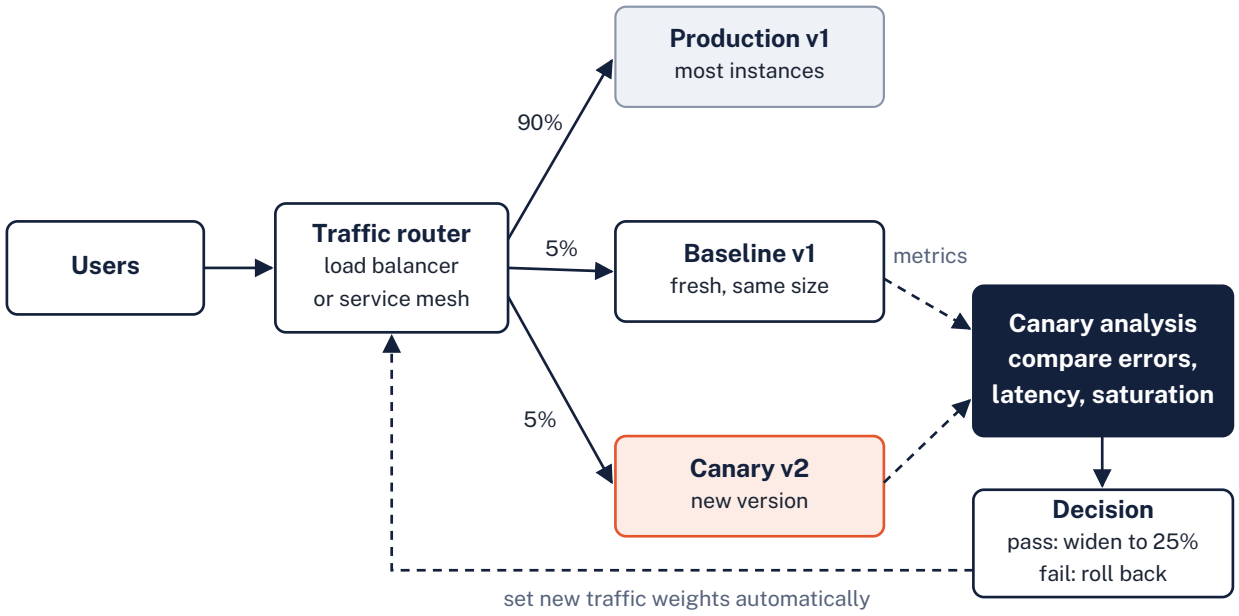
A deployment strategy decides how the new version replaces the old one in production, and therefore how many users a bad version can hurt before it is stopped. The strategies differ in the extra infrastructure they need, how fast they roll back and whether two versions run at once. Whenever two versions run at once, they must be compatible with each other's data and messages, which is the price of every strategy except recreate.

Deployment strategies compared

Strategy	How it works	Risk to users	Rollback	Extra cost and complexity
Recreate	Stop all old instances, then start the new ones.	Downtime during the switch; every user meets a bad version at once.	Redeploy the old version, with more downtime.	None. Acceptable for internal tools, batch jobs and maintenance windows.
Rolling	Replace instances a few at a time (Kubernetes' default for Deployments, with maxSurge and maxUnavailable both defaulting to 25%).	Exposure grows as the rollout proceeds; health checks catch crashes but not subtle errors.	Roll back by rolling the old version out again, which takes as long as the rollout.	Low. Old and new versions run side by side, so they must be compatible.
Blue-green	Run a full new environment (green) beside the live one (blue), test it, then switch all traffic at the router.	All users move at once, but the new environment was verified before the switch.	Fast: switch the router back, while blue is still running.	Up to double capacity during the switch; shared databases must work with both versions.
Canary	Send a small share of traffic (say 1-5%) to the new version, compare its metrics, then widen in steps.	Only the canary's share of users meets a bad version, and only until analysis stops it.	Fast: set the canary's weight to zero.	Needs weighted routing, good metrics and enough traffic for a meaningful comparison.
Shadow (dark launch)	Copy real requests to the new version; its responses are compared or discarded, never returned.	None directly, provided side effects (payments, emails, writes) are suppressed or faked.	Nothing to roll back; users never depended on it.	Double load on the shadowed path and careful handling of side effects; best for read paths and rewrites.

Blue-green deployment was described by Martin Fowler in 2010, crediting Dan North and Jez Humble; the canary release, named after the canaries miners carried to detect gas, was written up by Danilo Sato on the same site in 2014. In Kubernetes, rolling updates are built in; blue-green and canary releases usually come from a progressive delivery controller (Argo Rollouts, Flagger) or a service mesh and the Gateway API's weighted routing. On managed platforms, the equivalent features are called traffic splitting or deployment slots.

Canary release with automated analysis



Comparing the canary with a fresh baseline of the old version, rather than with the long-running fleet, removes differences caused only by uptime and warm caches.

IF YOU ARE NEW TO THIS

Step by step: one bad version, four ways to deploy it

This section's hardest idea is that the deployment strategy decides how many customers a bad version reaches before it is stopped, and how quickly it can be undone. Follow one faulty version of the order service through four strategies.

- The new version has a subtle bug: it does not crash, but charges the wrong delivery fee on some orders. Health checks pass, because the process itself is healthy.
- Recreate: all old copies stop, then the new ones start. After a short outage, every customer meets the faulty version at once. Undoing it means redeploying the old version, with another outage.
- Rolling: copies are replaced a few at a time. Since nothing crashes, the health checks let the rollout continue until every copy is faulty. Rolling back takes as long as rolling out.
- Blue-green: the new version runs as a complete second environment and is tested before all traffic is switched to it. If the tests miss the bug, every customer moves at once, but undoing it is fast: switch the router back to the old environment, still running.
- Canary: 5% of traffic goes to the new version, and its metrics are compared with a fresh copy of the old version. If the comparison includes a business signal such as the fees charged, it can spot the difference, and the canary's share is set to zero. Only customers in that 5% met the bug, and only until the analysis stopped it.
- **The lesson:** choose a strategy by how many customers you can afford to expose and how fast you must undo a mistake, and make sure your checks can see the failures you fear, not only crashes.

Separating deployment from release with feature flags

Deployment installs a new version of the software in an environment; **release** makes a behaviour available to users. Treating them as one event means every deployment is a business event, with all the nervousness that brings. Feature flags (feature toggles) separate them: the new code is deployed switched off, and the behaviour is released later, gradually and reversibly, by changing configuration. Pete Hodgson's article *Feature Toggles (aka Feature Flags)* on martinfowler.com (2017) distinguishes four categories by how long they live and how dynamically they decide.

Hodgson's four categories of feature toggle

Category	Purpose	Typical lifetime	How dynamic
Release toggles	Ship incomplete or unreleased features in the trunk, switched off.	Days to a few weeks; remove after release.	Mostly static per release.
Experiment toggles	A/B or multivariate tests that route each user to a cohort.	Long enough for statistically sound results, then removed.	Per request, by user cohort.
Ops toggles	Operational controls, such as switching off an expensive recommendation panel under load (kill switches).	Mostly short; a few kill switches are kept long-term.	Changed at run time, often during an incident.
Permissioning toggles	Features for particular users: premium tiers, beta testers, internal staff.	Can live for years.	Per request, by user.

Flags have a carrying cost. Every flag doubles the paths through the code it guards, adds test combinations and leaves dead code once its decision is settled. Hodgson recommends treating flags as inventory: add a removal task when a release toggle is created, give flags expiry dates, add tests that fail once a flag is past its date, and limit how many may exist at once. The Knight Capital loss of 2012, described in section 1.3, involved a repurposed flag that reactivated old code on a server missed by a manual deployment: a reminder never to reuse flag names and to delete dead code behind settled flags. At scale, flags are served by a dedicated service (open-source options such as Unleash and Flagsmith, commercial ones such as LaunchDarkly, or the vendor-neutral OpenFeature API) so that a change propagates in seconds and is audited.

Database changes in the pipeline

Code can be rolled back in seconds; data cannot. A deployment that renames a column and is then rolled back leaves the old code facing a schema it does not understand. The answer, covered in depth in schema evolution, is to make every schema change backward-compatible with the version of the code currently running, using expand and contract: add the new structure first, deploy code that writes both and reads either, backfill, switch readers, and remove the old structure only in a later release once nothing uses it.

In the pipeline this means migrations are versioned artefacts that run as their own step, before the code that needs them is deployed, and are tested on a production-sized copy of the data so that a lock on a large table is discovered in staging rather than at lunchtime. Destructive steps

(dropping columns or tables) are delayed and run separately, because they are the one kind of change a rollback cannot undo. A useful pipeline check fails the build when a migration is not additive unless it is explicitly marked as a contract step.

Progressive delivery and automated canary analysis

Progressive delivery, a term popularised by James Governor of the analyst firm RedMonk in 2018, combines canaries, feature flags and automated verification: a change reaches progressively larger audiences, and at each step the system, not a person watching dashboards, decides whether to continue. The decision uses automated canary analysis: the canary's metrics (error rate, latency percentiles, saturation, business signals such as orders placed) are compared statistically with a baseline. Netflix and Google open-sourced their analysis engine, Kayenta, in 2018; Argo Rollouts and Flagger provide similar analysis steps in Kubernetes.

Three details decide whether the analysis can be trusted. Compare like with like: a fresh baseline of the old version receiving the same share of traffic avoids blaming the new version for differences caused by cold caches or recent restarts. Have enough traffic: at 5% of a service handling ten requests a minute, an hour of canary sees thirty requests, too few to detect a 1% error rate, so low-traffic services need longer steps, synthetic load or a different strategy. Choose metrics tied to the service level objectives and to business outcomes, because a version can be fast and error-free while silently charging the wrong delivery fee.

Rollback or roll forward

When a release goes wrong, there are two ways out. Rollback returns to the previous artefact, which is fast and needs no new code, provided the previous version still works with the current data and with the versions of other services. Roll forward ships a fix through the pipeline, which is slower but sometimes the only option, for example after a data migration that cannot be reversed or a change that other services already depend on.

A sound default is “roll back first, diagnose later”: restore service with the fastest safe action, then investigate without pressure. Rolling forward under pressure tends to bypass the pipeline's checks, and a rushed fix to a production incident is a classic source of the next incident. Rollback is only a real option if it is designed in: previous artefacts kept, schema changes compatible in both directions for at least one version, messages readable by old and new consumers, and the rollback path itself exercised regularly. A team that has never rolled back should assume it cannot.

WORKED EXAMPLE

Worked example: releasing a new delivery-fee rule

The food-delivery platform's order service is changing how delivery fees are calculated, using distance and demand instead of a flat fee. The team uses trunk-based development, a canary deployment and a release flag.

Change	New fee calculator behind a release flag `dynamic-delivery-fee`, off by default; an expand-only migration adds a nullable column `fee_rule_version`.
Commit to artefact	Merged to main at 10:02. Lint, unit tests and scans take 6 minutes; image `orders:4f1c2ab` with its SBOM and signature is pushed at 10:09.
Migration	The additive migration runs at 10:15 as its own pipeline step. The running version ignores the new column, so nothing breaks.
Staging	The same image is deployed at 10:16; contract tests with the payment and restaurant services and smoke tests pass by 10:31.
Canary	At 10:35, 5% of production traffic goes to the new version, still with the flag off. Analysis over 20 minutes compares p99 latency, 5xx rate and order success with a fresh baseline: pass.
Rollout	Weights move to 25%, 50% and 100% at 15-minute steps; complete at 11:40. The deployment is finished, yet no customer has seen the new fee.
Release	Next day the product manager enables the flag for 10% of customers in one city and watches order conversion and complaints, then widens to all cities over a week.
Clean-up	The flag and the flat-fee code are deleted the following sprint; a later contract migration makes the column mandatory.

Deployment risk (does the code work?) and release risk (do customers like the new fees?) were handled separately, each with its own small blast radius.

At every step until the contract migration, recovery was a configuration change or a router weight, not an emergency fix. Commit to full production deployment took about 1 hour 40 minutes.

Measuring the pipeline

The DORA software delivery metrics introduced in section 2.6 (change lead time, deployment frequency, change fail rate, failed deployment recovery time and deployment rework rate) are largely properties of the pipeline and the architecture together. Small batches and trunk-based development shorten lead time and raise frequency; automated testing and canary analysis lower the change fail rate; fast rollback and flags shorten recovery time. Measure them per service, watch trends rather than single values, and never turn them into targets for individuals, which simply invites gaming.

WATCH OUT

Common pipeline mistakes

- Rebuilding the artefact for each environment, so production runs something that was never tested.
- A commit stage that takes 40 minutes, so developers batch changes to avoid waiting, which defeats the purpose.
- Flaky tests that are re-run until green, teaching everyone to ignore red builds.
- Manual approval gates at every stage that add days of waiting without adding information.
- Migrations bundled into application start-up, so every instance races to apply them and a rollback meets an incompatible schema.
- Feature flags that are never removed, leaving hundreds of untested combinations.

CHECK YOUR UNDERSTANDING

▼ **Why is rebuilding the artefact separately for production a problem even if the source code is identical?**

Dependencies, base images or build tools may have changed between builds, so the production artefact is not the one that was tested; build once and promote the same artefact.

▼ **What is the difference between deploying a feature and releasing it?**

Deployment installs the code in an environment; release makes the behaviour available to users. Feature flags let the code be deployed switched off and released later, gradually and reversibly.

▼ **Why does a canary on a low-traffic service often give unreliable results?**

A small share of little traffic yields too few requests to detect differences in error rate or latency with statistical confidence, so steps must be longer, traffic synthetic or another strategy chosen.

▼ **When might rolling forward be the only option?**

When the release included an irreversible change, such as a destructive migration, or other services already depend on the new behaviour, so the previous version can no longer run correctly.

▼ **Try it: A team rebuilds the image for each environment, deploys by stopping everything and restarting at lunchtime, and the new version drops a database column in the same release. Find three problems and give a better approach.**

Rebuilding means production runs something never tested, so build once and promote one immutable artefact. Recreate at lunchtime causes downtime at peak, so use a rolling or canary deployment, with automatic comparison against a baseline. Dropping the column in the release makes rollback impossible, and old and new versions cannot run together, so use expand and contract, removing it in a later release. Add a feature flag to separate deployment from release.

SOURCES

- Pete Hodgson: Feature Toggles (aka Feature Flags), martinfowler.com: https://martinfowler.com/articles/feature-toggles.html
- Martin Fowler: BlueGreenDeployment: https://martinfowler.com/bliki/BlueGreenDeployment.html
- Danilo Sato: CanaryRelease: https://martinfowler.com/bliki/CanaryRelease.html
- Trunk Based Development: https://trunkbaseddevelopment.com/
- DORA: software delivery performance metrics: https://dora.dev/guides/dora-metrics-four-keys/

SECTION 10.6 OF 10.8 · ADVANCED · 24 MIN READ · LAST REVIEWED NOVEMBER 2026

Multi-region, disaster recovery and edge computing

BEFORE YOU READ THIS

This section builds on:

- 2.5 Availability and reliability
- 6.6 CAP and consistency
- 7.4 Replication and partitioning

Terms it uses from earlier: Failover, Replication lag, PACELC, Blast radius

THE IDEA IN BRIEF

A shop chain that keeps all its stock in one warehouse is in trouble if that warehouse floods. It could keep only an inventory list and a spare key to an empty building elsewhere (cheap but slow to recover), keep a small second warehouse stocked and staffed (dearer but quick), or run two full warehouses all the time (dearest, almost no interruption). It can also put small kiosks near customers for the most popular items. Multi-region and edge design make the same choices for software, deciding how much to pay in advance against how long customers can wait.

WHAT YOU NEED TO KNOW

- Run across several regions only for latency, disaster recovery or data residency; multiple availability zones in one region are the normal baseline and much cheaper.
- RTO (how long you may be down) and RPO (how much data you may lose) come from the business and decide the disaster recovery strategy and its cost.
- The AWS Well-Architected strategies, backup and restore, pilot light, warm standby and multi-site active-active, buy progressively lower RTO and RPO with higher continuous cost and complexity.
- Cross-region data is the hard part: synchronous replication adds long round trips to writes, asynchronous replication lag becomes the effective RPO, and replication copies deletions and corruption, so backups remain essential.
- Failover is steered by DNS or anycast global load balancers; statically stable designs keep pre-provisioned capacity and avoid depending on the failed region's control plane.
- Cell-based architecture and shuffle sharding limit the blast radius of bad deployments, overloads and poison inputs that a second region would suffer too.

- CDNs and edge functions, such as Cloudflare Workers running in V8 isolates, serve reads and lightweight logic near users, while systems of record usually stay in a region.

Why run in more than one region

Section 0.7 (servers and the cloud) introduced regions (separate geographic areas) and availability zones (independent data-centre groups inside a region), and section 10.1 showed how providers organise them. Running across several availability zones in one region is the normal baseline for any production system: it survives the loss of a data centre at little extra cost, because zones are close enough for synchronous replication. Running across several *regions* is a much bigger step, and there are only three good reasons to take it.

Latency. Light in optical fibre travels at roughly 200,000 km per second, so a round trip between London and Sydney (about 17,000 km in a straight line) cannot take less than about 170 ms, and real routes take considerably longer. A request that needs several round trips to a distant region feels slow however fast the servers are. Serving users from a nearby region, or from the edge, is the only fix.

Disaster recovery. Whole regions do fail, usually through a shared dependency rather than a physical disaster. On 19 and 20 October 2025, a race condition in the automation that manages DNS records for Amazon DynamoDB left its US-EAST-1 endpoint with an empty DNS record; DynamoDB errors lasted about three hours, but knock-on failures in launching new virtual machines and in load balancer health checks lasted into the following afternoon. Every large provider has had comparable regional incidents. Disaster recovery plans for them.

Data residency. Data residency requirements, from law, regulators or customer contracts, may require certain data to be stored or processed in particular jurisdictions. The GDPR itself restricts transfers of personal data out of the European Economic Area rather than demanding local storage, but sector rules, national laws and enterprise contracts often go further (see privacy by design). Residency often forces multi-region designs in which data is deliberately *not* replicated everywhere.

Multi-region is expensive: duplicated infrastructure, cross-region data transfer charges, harder data consistency and more complex operations and testing. Many teams that believe they need it actually need good multi-zone design, tested backups and a CDN.

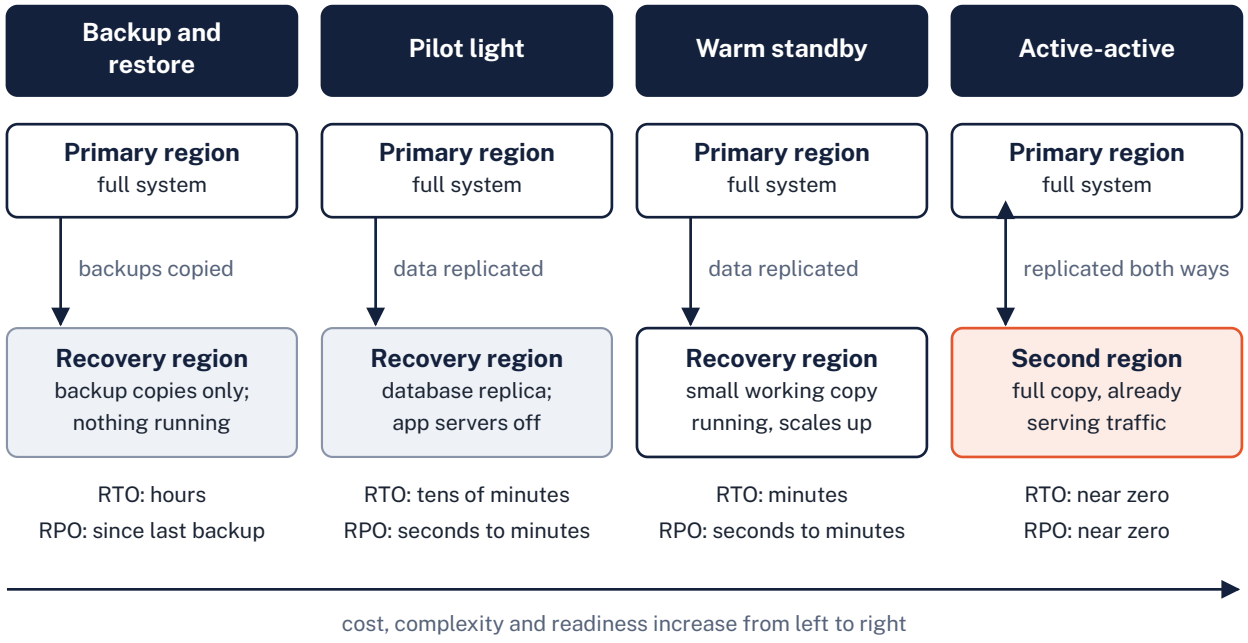
Recovery objectives and the four strategies

Section 2.5 (availability and reliability) defined the two numbers that drive every disaster recovery design. The recovery time objective (RTO) is how long the service may be unavailable; the recovery point objective (RPO) is how much recent data may be lost, expressed as time.

Both come from the business, ideally from the cost of an hour of outage and of a lost order, not from engineers' preferences. Lower values cost more, and the cost rises steeply near zero.

The AWS Well-Architected whitepaper *Disaster Recovery of Workloads on AWS: Recovery in the Cloud* describes four strategies, in order of increasing cost and complexity. The names are widely used beyond AWS and apply equally to Azure, Google Cloud or a pair of private data centres.

Disaster recovery strategies



Each step to the right keeps more of the system running in the second region, buying lower RTO and RPO with higher continuous cost.

Disaster recovery strategies compared

Strategy	What runs in the second region	Typical RTO	Typical RPO	Relative cost and main risk
Backup and restore	Nothing but backup copies (database snapshots, object storage) and the infrastructure code to rebuild.	Hours, depending on data volume and automation.	The backup interval (minutes with continuous log backups, hours with daily snapshots).	Lowest. Restores are slow and often untested; without infrastructure as code the RTO is fiction.
Pilot light	Replicated databases and core infrastructure; application servers defined but not running.	Tens of minutes: start and scale the application tier, then redirect traffic.	Seconds to minutes (replication lag).	Low. Scaling up during a regional crisis competes with everyone else for capacity.
Warm standby	A scaled-down but fully working copy of the whole system.	Minutes: redirect traffic, then scale up.	Seconds to minutes.	Moderate. The standby can be tested with real traffic, but drifts if not deployed with every release.
Multi-site active-active	A full copy serving live traffic in every region.	Near zero: traffic is redistributed, no failover needed.	Near zero for replicated data; corruption still needs backups.	Highest. Requires a data design for concurrent writes in several regions.

The whitepaper itself gives one rough figure covering both RTO and RPO for each strategy; the table separates them because, with continuous replication, the RPO of pilot light and warm standby is set by replication lag rather than by how much of the application is running. Two warnings apply to every column. Replication is not a backup: an accidental deletion, a bad migration or ransomware is replicated faithfully within seconds, so point-in-time backups, ideally immutable and in a separate account, are needed even in an active-active design. And a strategy that has not been rehearsed is a hope, not a plan. Teams that run a regional failover exercise at least a few times a year discover missing permissions, hard-coded endpoints, expired certificates and quotas that are too small in the recovery region, while it is still cheap to find them.

Replicating data across regions

Stateless application servers are easy to run in two places; the data is the hard part, and the trade-offs are those of consistency, CAP and PACELC at continental distances. Synchronous replication across regions means every write waits for at least one long round trip, perhaps 70 to 150 ms or more, which is acceptable for a few critical writes and ruinous for chatty ones. Asynchronous replication keeps writes fast but means the replica lags, and whatever has not arrived when the primary region fails is lost: the replication lag *is* the effective RPO.

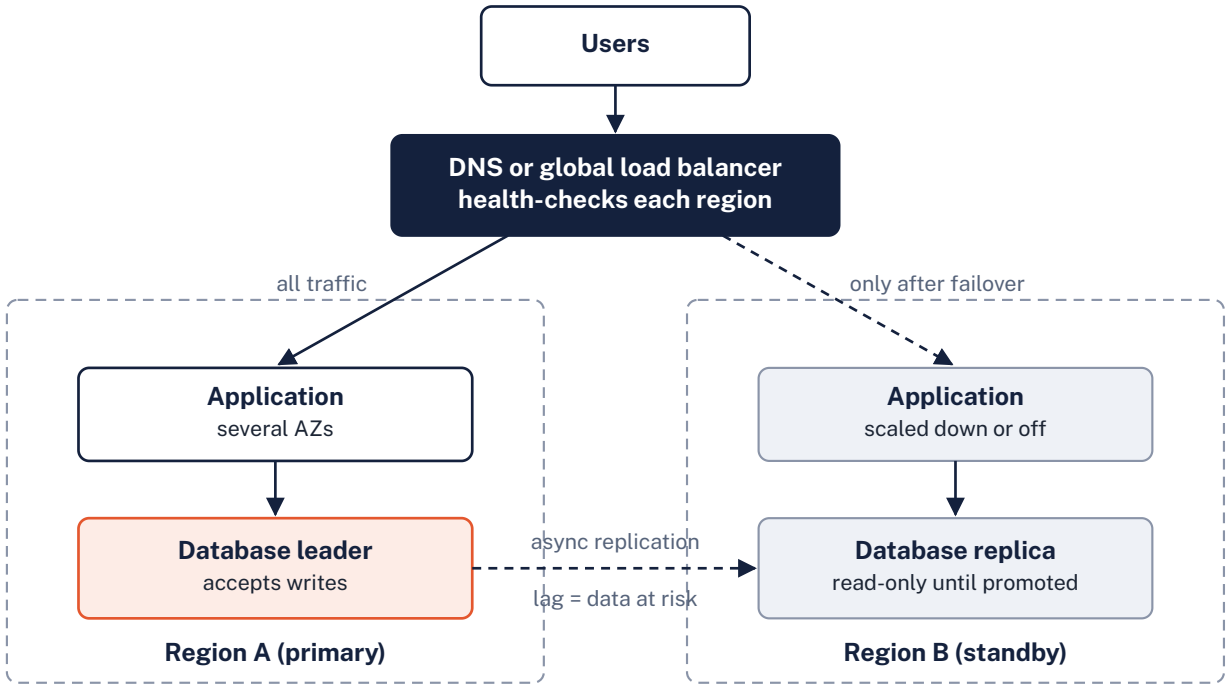
There are four common data designs. **Single primary region**: all writes go to one region and replicas elsewhere serve reads and recovery; simple, but distant users pay write latency and failover means promoting a replica (with the split-brain risks described in replication and partitioning). **Home region per entity**: each customer, restaurant or tenant has a home region where its data is written, which keeps writes local and supports residency, at the cost of routing and of rare cross-region operations. **Multi-writer with conflict resolution**: every region accepts writes and conflicts are merged, for example by last-writer-wins (the default in DynamoDB global tables, which since 2025 also offer multi-Region strong consistency) or by conflict-free replicated data types; safe only for data where merging is meaningful, such as a basket, and dangerous for money or stock. **Globally consistent databases** such as Google Spanner or CockroachDB pay consensus latency on writes to give strong consistency across regions; Azure Cosmos DB lets each application choose among five consistency levels, from strong to eventual.

Most real systems mix these per data set, exactly as section 6.6 recommends: payments and order state in a single primary or home region, menus and restaurant details replicated everywhere for reads, and analytics shipped asynchronously.

Failover, DNS and static stability

Traffic is steered between regions by DNS or by a global load balancer. With DNS-based failover, the DNS service health-checks each region and returns the address of a healthy one; its weakness is caching, because resolvers and clients may hold old answers for longer than the record's TTL. Global load balancers (for example AWS Global Accelerator, Azure Front Door, Google Cloud's global external load balancer, Cloudflare) use anycast addresses announced from many locations, so clients keep one address and the provider shifts traffic within seconds.

Failover between a primary and a standby region



Whatever has not been replicated when region A fails is lost, so the replication lag sets the real recovery point.

Deciding *when* to fail over is harder than the mechanism. Automatic failover reacts fastest but can be triggered by a monitoring fault or a brief network blip, and failing over a database risks two primaries accepting writes. Many organisations therefore automate detection and the failover steps but keep a human decision to start a regional failover, with a runbook and a time limit. Failing back after the incident, including re-synchronising data written in the recovery region, needs a plan too.

The October 2025 incident illustrates a principle that the AWS Builders' Library calls static stability: a system should keep working during a failure without having to make changes at that moment. Systems that already had enough capacity running rode out the problem; systems that needed to launch new instances, or whose recovery depended on control-plane actions in the affected region, could not recover until the region did. For disaster recovery this argues for pre-provisioned capacity (warm standby rather than pilot light, for critical paths), for failover that depends only on the healthy region's data plane, and for DNS, identity and secrets that do not live solely in the region you are escaping.

IF YOU ARE NEW TO THIS

Step by step: recovering without asking for anything

The idea that needs the most care here is static stability: a recovery plan is only as good as the things it needs at the moment of failure, and the safest plan needs nothing new. Compare two ways of preparing a second region for the food-delivery platform's ordering core.

- With a pilot light, the second region holds a continuously updated copy of the data and the core infrastructure, but the application servers are off. With a warm standby, a small working copy of the ordering core already runs there, receiving a little real traffic.
- At dinner time the primary region fails. Under pilot light, recovery means making changes now: start the application servers, scale them up, then redirect traffic. Each start is a request to the provider's control plane, the machinery that creates and changes resources.
- That is the worst moment to ask. Other customers want capacity at the same time, and any step that relies on something kept only in the failed region, such as DNS, identity or secrets, must wait until that region recovers.
- Under warm standby, the copy is already running and proven by real traffic. Traffic is redirected to it at once; it serves at reduced capacity while scaling up. If scaling is slow, customers get a reduced service rather than none.
- Static stability costs money: a copy that runs all the time costs more each month than pilot light's switched-off servers, as the worked example later in this section shows.
- **The lesson:** for critical paths, prefer recovery that relies only on what is already running in the healthy region. Anything that must be created or changed during the disaster may not be available when needed.

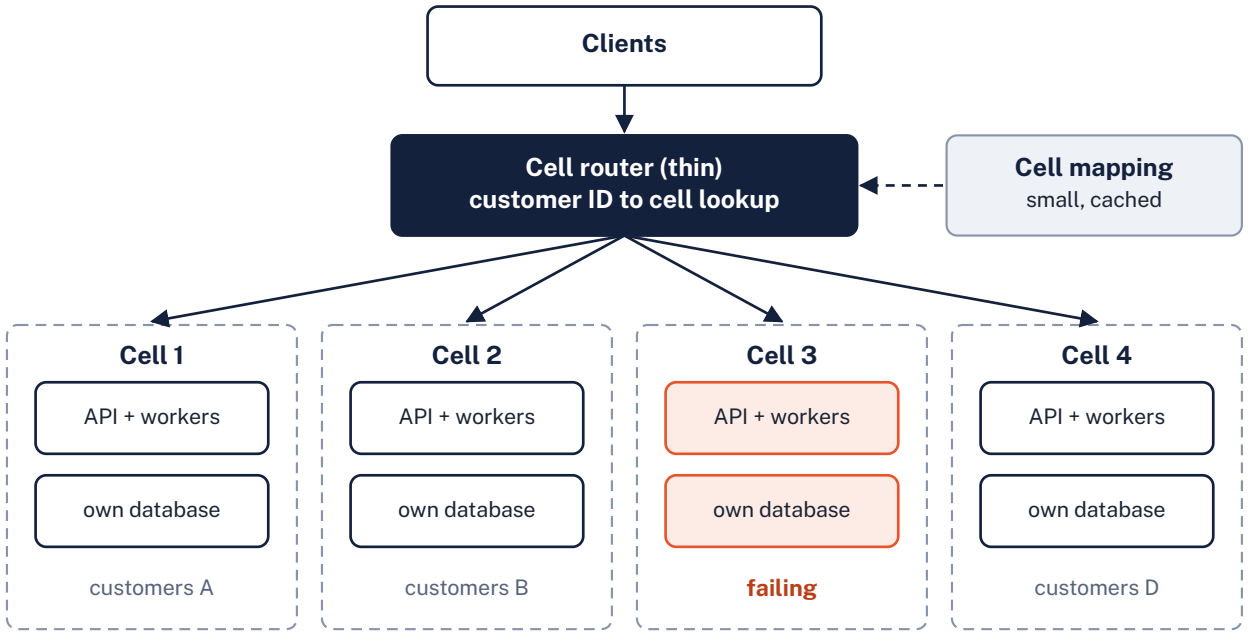
Cells: limiting the blast radius

Regions protect against regional disasters; most outages, however, are caused by the system's own changes, overloads and poison inputs, which a second region running the same code and configuration will suffer too. Cell-based architecture limits the blast radius of such failures by running many complete, independent copies of the system, called cells, each serving a fixed subset of customers. A thin router maps each request to its cell, usually by customer or tenant identifier. Cells share nothing at run time: each has its own application instances, queues and databases.

If one cell suffers a bad deployment, a hot customer or data corruption, only that cell's customers are affected: with 20 cells, about 5% of them. Deployments roll out cell by cell in waves, so a bad release is caught in the first cell. The design is described in the AWS Well-

Architected whitepaper *Reducing the Scope of Impact with Cell-Based Architecture* (2023) and is used at large scale by AWS's own services and by companies such as Slack. A related technique, shuffle sharding, gives each customer a random combination of a few workers out of many (with 8 workers and 2 per customer there are 28 combinations), so a customer whose requests crash workers takes down only its own combination, and most other customers share at most one worker with it.

Cell-based architecture



A fault, bad deployment or noisy customer in cell 3 affects about a quarter of customers, not all

The router must be the simplest, most reliable component in the system, because it is the one thing all cells share.

Cells cost something: more instances and databases to run, cross-cell operations (reports across all customers, moving a customer between cells) that need separate machinery, and a router and mapping service that must be extremely simple and reliable. They suit large multi-tenant systems with natural partition keys; for a single-tenant internal application they are over-engineering.

The edge: CDNs and edge functions

Between users and regions sits the edge: the hundreds of locations operated by content delivery networks such as Akamai, Cloudflare, Fastly, Amazon CloudFront and Azure Front Door. A CDN caches static and cacheable content (images, scripts, menu pages) near users, following the HTTP caching rules described in caching, and absorbs traffic spikes and many denial-of-service attacks before they reach the origin.

Edge functions go further by running code at those locations: Cloudflare Workers, Fastly Compute, CloudFront Functions and Lambda@Edge, Akamai EdgeWorkers, and the edge functions of platforms such as Vercel and Netlify. Cloudflare Workers runs code in lightweight V8 isolates, the sandboxes JavaScript engines use for browser tabs, rather than in a container per function, so start-up is very fast and many tenants share one process safely. This course itself runs on Cloudflare Workers. Good uses are request routing and A/B assignment, authentication checks and token validation, header and redirect rules, image resizing, simple personalisation, and API proxies that keep secret keys out of the browser.

The limits are architectural. Edge functions have tight CPU-time and memory limits; edge storage is usually eventually consistent (Cloudflare's Workers KV can take some time to propagate a write globally) or offers strong consistency only for a single coordinating object, as Durable Objects do; and the system of record normally stays in a region. A common pattern is therefore *read at the edge, write at the origin*: serve menus, restaurant pages and session checks from the edge, and send orders and payments to the home region.

DECISION GUIDE

Decision guide: how much multi-region do you need?

Answer for each important user journey, not for the whole system at once.

Q1 **Can the business tolerate this journey being unavailable for several hours while a region is rebuilt from backups?**

- **Yes** Use several availability zones in one region plus tested, automated, cross-region backup and restore. Revisit if the business changes.
- **No** Continue: a standby in another region is needed.

Q2 **Is a recovery time of tens of minutes acceptable, and can you count on obtaining capacity during a regional incident?**

- **Yes** Pilot light: replicate data continuously, keep infrastructure as code ready and automate the scale-up and traffic switch.
- **No** Continue: capacity must already be running.

Q3 **Is a recovery time of a few minutes acceptable, with reduced capacity at first?**

- **Yes** Warm standby: run a small working copy, deploy to it with every release and send it a little real traffic to prove it works.
- **No** Active-active, with a data design for multi-region writes (home regions or a globally consistent database); budget for the engineering, not just the infrastructure.

Q4 **Do distant users need fast writes, or must some data stay in their jurisdiction?**

- **Yes** Partition by geography: give customers a home region and keep their data there, rather than replicating everything everywhere.
- **No** Serve distant users' reads through a CDN and edge functions and keep writes in one primary region.

WORKED EXAMPLE

Worked example: choosing a DR strategy for the food-delivery platform

The platform runs in one region across three availability zones. The ordering core (ordering, payment, dispatch) costs about 60,000 a month in infrastructure. It handles about 40,000 orders a day, peaking at 6,000 an hour at dinner, with about 5 in platform revenue per order. Leadership asks for a recovery plan for a regional outage. All figures are illustrative, in pounds.

Cost of an outage	Revenue averages about 8,000 an hour over the day and reaches 30,000 at peak. Customer, restaurant and courier goodwill roughly doubles the true cost. Contracts with large restaurant chains promise ordering is restored within one hour.
Requirement	RTO of 1 hour or less and RPO of under 1 minute for the ordering core; RTO of 24 hours for analytics, recommendations and marketing.
Backup and restore	Extra about 2,000 a month. Measured rebuild: 4 to 6 hours (2 TB database restore dominates). Fails the 1-hour RTO.
Pilot light	Extra about 8,000 a month (continuous database replica, networking, images ready). Rehearsed failover: 35 to 50 minutes, mostly scaling the application tier. Meets the RTO only if capacity can be obtained during a regional event.
Warm standby	Extra about 18,000 a month (replica plus application at 20% scale). Rehearsed failover: about 10 minutes, then scale-up while serving. Meets the RTO with margin.
Active-active	Extra about 45,000 a month plus an estimated six engineer-months to redesign order and payment data for multi-region writes. Near-zero RTO, far beyond the requirement.
Decision	Warm standby for the ordering core; backup and restore for analytics, recommendations and marketing; menus and restaurant pages served from the CDN so browsing continues during failover. Total extra cost about 20,000 a month, about a third of the core's infrastructure.
Conditions	Standby is deployed by the same pipeline with every release, receives 2% of real traffic to prove it works, and a full failover is rehearsed each quarter. The decision is recorded in an ADR.

The expected loss from a rare regional outage alone would not justify warm standby; the contractual RTO and the reputational damage of a long outage at dinner time do. DR is bought like insurance, and the business sets the premium it is willing to pay.

Tiering by journey cut the cost substantially: only the part of the system that must recover quickly pays for fast recovery.

WATCH OUT

Multi-region pitfalls

- A recovery region that has never received traffic: configuration drift, missing secrets and small quotas surface only during the disaster.
- Failover tooling, DNS, identity or the deployment pipeline hosted only in the region that failed.
- Assuming replication protects against deletion or corruption; it copies them.
- Synchronous cross-region calls on the user's critical path, which add long round trips to every request.
- Treating active-active as an infrastructure setting rather than a data design problem.

CHECK YOUR UNDERSTANDING

▼ Why does an active-active design still need point-in-time backups?

Replication copies deletions, bad migrations and corruption to every region within seconds; only backups let you restore the data as it was before the mistake.

▼ In a design with asynchronous replication to a standby region, what determines how much data is lost in a regional failure?

The replication lag at the moment of failure: writes that had not yet reached the replica are lost, so the lag is the effective RPO.

▼ What failures does a cell-based architecture protect against that a second region running the same code does not?

Failures caused by the system itself, such as a bad deployment, a poison input or an overloaded customer, which would affect every region but only one cell.

▼ Why is pilot light riskier than warm standby during a large regional incident?

Pilot light must launch and scale the application tier at the moment of failure, when capacity and control-plane operations may be scarce; warm standby already has working capacity running.

▼ Try it: Illustrative: backup and restore means an 8-hour regional outage at £8,000 lost per hour. A warm standby costs £6,000 a month and cuts it to 20 minutes. One regional outage is expected every five years. Compare yearly costs and advise.

Backup: 8 x £8,000 = £64,000 per outage, or £12,800 a year. Standby: £72,000 a year, plus about £533 of expected loss. On revenue alone it is not justified. Check RTO and RPO with the business: contracts, safety or lost customers could change the answer, and a cheaper pilot light may fit. Keep tested backups whichever you choose, because replication copies deletions and corruption.

SOURCES

- AWS Whitepaper: Disaster Recovery of Workloads on AWS, disaster recovery options in the cloud: https://docs.aws.amazon.com/whitepapers/latest/disaster-recovery-workloads-on-aws/disaster-recovery-options-in-the-cloud.html
- AWS Whitepaper: Reducing the Scope of Impact with Cell-Based Architecture: https://docs.aws.amazon.com/wellarchitected/latest/reducing-scope-of-impact-with-cell-based-architecture/reducing-scope-of-impact-with-cell-based-architecture.html
- AWS: Summary of the Amazon DynamoDB service disruption in the Northern Virginia (US-EAST-1) Region: https://aws.amazon.com/message/101925/
- Amazon Builders' Library: Workload isolation using shuffle-sharding: https://aws.amazon.com/builders-library/workload-isolation-using-shuffle-sharding/
- Cloudflare Workers documentation: How Workers works: https://developers.cloudflare.com/workers/reference/how-workers-works/

SECTION 10.7 OF 10.8 · INTERMEDIATE · 24 MIN READ · LAST REVIEWED NOVEMBER 2026

Platform engineering and internal developer platforms

BEFORE YOU READ THIS

This section builds on:

- 3.7 Conway's law
- 10.4 Infrastructure as code
- 10.5 CI/CD and deployment

Terms it uses from earlier: Kubernetes, Infrastructure as code (IaC), Policy as code, DevOps, Observability

THE IDEA IN BRIEF

Imagine a large office where every department has to buy its own printers, wire its own network and hire its own cleaner. Work gets done, but everyone wastes time on chores and some departments do them badly. A good facilities team instead offers ready-made services: book a meeting room online, order a laptop from a short list, request a desk in a few clicks. Nobody is forced to use them, but they are the easiest way. Platform engineering is that facilities team for software teams: it makes the common, necessary chores of building and running software quick, safe and self-service.

WHAT YOU NEED TO KNOW

- Platform engineering builds an internal developer platform: self-service tools, services, knowledge and support that let product teams build and run software without tickets, reducing duplicated effort and uneven quality.
- A golden path (paved road) is an opinionated, supported, end-to-end and optional way to do a common task, such as creating a new service and deploying it to production.
- Backstage, created at Spotify and open-sourced in 2020, is a CNCF project offering a software catalogue, templates, TechDocs and plugins; a portal is the front door, not the platform itself.
- The platform is a product with internal customers: it needs a product owner, user research, documentation, its own SLOs and adoption that is earned rather than mandated (Team Topologies' X-as-a-Service and thinnest viable platform).
- Self-service is made safe by guardrails, automatic policy checks with clear messages and an exception route, instead of human approval gates.

- Success is measured by other teams' outcomes: adoption, time to first deployment, DORA metrics and developer experience, using several SPACE dimensions rather than one number.
- Typical failures are building a platform nobody wants, renaming a ticket-driven operations team, mandating adoption and abstractions that hide too much to debug.

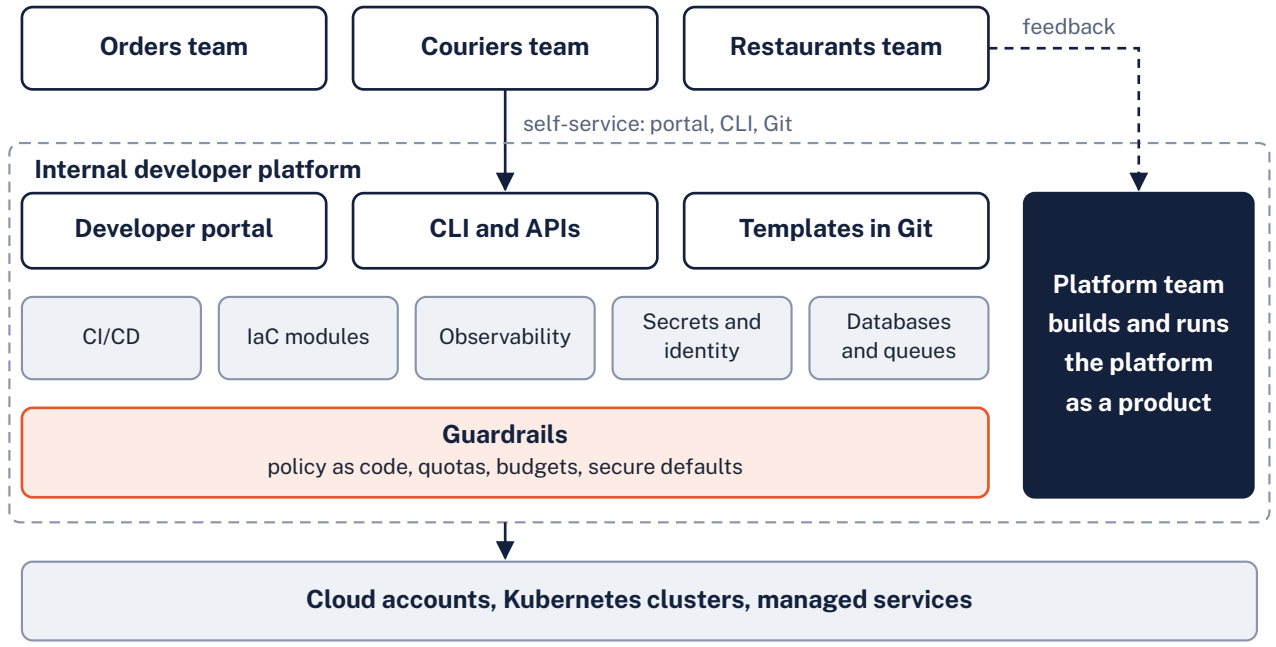
The problem platforms solve

In 2006 Amazon's chief technology officer, Werner Vogels, summed up his company's operating model as “you build it, you run it”: the team that writes a service also deploys, monitors and fixes it. The idea became central to DevOps and to microservices, and it works: teams that run their own software feel the consequences of their design choices and fix them. But the cloud-native toolset described in this chapter is large. A product team that must build, run and secure a service today may need to understand container images, Kubernetes objects, infrastructure as code, pipelines, observability, secrets management, network policies, supply-chain scanning and cost controls, before writing a line of business logic.

When every team solves all of this alone, three things happen. Effort is duplicated, as ten teams write ten slightly different pipelines. Quality varies, as some teams encrypt their databases and configure backups and others forget. And the time product teams spend on infrastructure is time not spent on the product. Platform engineering is the response: a dedicated team builds an internal developer platform (IDP) that packages these capabilities so that product teams can use them through self-service.

Evan Bottcher's 2018 article on martinfowler.com gives a widely quoted description: a digital platform is a foundation of self-service APIs, tools, services, knowledge and support, arranged as a compelling internal product. Each part of that description matters. *Self-service* means no tickets for routine needs. *Knowledge and support* means documentation and people, not only software. *Compelling* means teams choose to use it because it is better than doing the work themselves. *Internal product* means it is managed like a product, with customers, rather than like a project or a cost centre.

What an internal developer platform contains



Product teams consume the platform through self-service interfaces; the platform team runs it as a product and learns from their feedback.

An IDP is not one product. It is usually an integration of existing tools (a CI system, a GitOps controller, a cloud provider's managed services, an observability stack, a secret store) behind consistent interfaces and sensible defaults. The platform team's distinctive work is choosing, integrating, documenting and supporting them, and designing the experience of using them, not writing a new orchestrator.

Golden paths and paved roads

The most valuable thing a young platform can offer is a golden path: an opinionated, supported and documented way to accomplish a common task end to end, such as creating a new service and getting it into production. Spotify popularised the term (its engineering blog described its golden paths in 2020); Netflix speaks of the *paved road*. The metaphor is precise. A paved road is the easiest route, and it is maintained, but you may still drive off-road if you have a good reason; you just do not get the road's support.

A good golden path has five properties. It is **end to end**: it takes a team from an empty repository to a monitored service in production, not just through one step. It is **opinionated**: it makes the common choices (language versions, logging format, deployment strategy, database defaults) so that teams do not have to. It is **optional**: teams with unusual needs can leave it,

which keeps the platform team honest about whether the path is actually better. It is **supported**: when it breaks, the platform team fixes it. And it is **evolving**: services created from last year's template receive updates, rather than being frozen at the version they started with.

What a golden path for a new service typically provides

Element	What the product team gets	What it replaces
Service template	A repository with a working skeleton, tests, Dockerfile, linting and a README, generated in minutes.	Copying an old service and deleting what does not fit.
Pipeline	A ready CI/CD pipeline with scanning, signing and progressive deployment, as in section 10.5.	Each team writing and maintaining its own pipeline.
Runtime configuration	Kubernetes or managed-platform settings with sensible resource requests, probes and autoscaling.	Trial and error in production.
Infrastructure modules	Reviewed infrastructure-as-code modules for databases, queues and buckets, with encryption, backups and tags built in.	Tickets to an operations team, or hand-written modules of varying quality.
Observability	Structured logging, metrics, traces and a starter dashboard with alerts on service level objectives.	Monitoring added after the first incident.
Catalogue entry and docs	The service registered with its owner, dependencies and documentation.	Spreadsheets and tribal knowledge of who owns what.

Backstage and developer portals

A developer portal is the front door of many platforms. The best-known is Backstage, built inside Spotify, open-sourced in March 2020, accepted by the Cloud Native Computing Foundation (CNCF) as a sandbox project in September 2020 and promoted to incubation in March 2022. Its core features are a **software catalogue**, in which every service, website, library, API and resource is registered with its owner and relationships; **software templates**, which generate new projects from golden-path templates; **TechDocs**, which publishes documentation written as Markdown next to the code; and a large ecosystem of plugins that show pipelines, Kubernetes status, costs and incidents alongside each catalogue entry.

Each component describes itself in a small file kept in its own repository, so ownership information lives with the code and is reviewed like code:

```yaml
apiVersion: backstage.io/v1alpha1
kind: Component
metadata:
  name: restaurant-ratings
  description: Stores and serves customer ratings of restaurants
  tags: [python, http]
spec:
  type: service
  lifecycle: production
  owner: team-restaurants
  system: restaurant-discovery
  providesApis: [ratings-api]
  dependsOn: [resource:ratings-db]
```

A Backstage catalogue entry (catalog-info.yaml) for a food-delivery service: its owner, lifecycle, the system it belongs to, the API it provides and a resource it depends on.

Two cautions. Backstage is a framework that an organisation assembles and operates, typically needing engineers who know TypeScript and React, and its plugins vary in quality; managed Backstage services and commercial portals exist for teams that do not want to run it. More importantly, **a portal is not a platform**. A beautiful portal in front of manual, ticket-driven processes merely displays the queue more attractively. Start with the capabilities and the golden path; add a portal when there is enough behind it to be worth finding.

The platform as a product

The single most important idea in platform engineering is that the platform is a product and the developers who use it are its customers. That means a product manager or a platform lead who owns the roadmap; user research (interviews, observing a team create a service, surveys) before building; documentation and onboarding treated as features; published service level objectives for the platform itself; versioning and deprecation policies so that teams can plan; and, above all, adoption that is earned rather than mandated. If teams avoid the platform, that is market feedback, not insubordination.

Team Topologies (Matthew Skelton and Manuel Pais, 2019), covered in section 11.5, gives the organisational model. A platform team exists to reduce the load on stream-aligned teams, those aligned to a flow of business change such as ordering or courier dispatch, and interacts with them mainly in *X-as-a-Service* mode: a clear, self-service offering with little need for collaboration. Collaboration mode is used deliberately and briefly, for example when designing a new capability with one or two early-adopter teams. Skelton and Pais also coined the thinnest viable platform: the smallest platform that genuinely helps, which for a small organisation may be a well-written wiki page of approved services and a shared pipeline template.

This also means the platform must obey Conway's law in its favour: the platform's interfaces become the boundaries between the platform team and every product team, so they should be stable, documented and narrow, like any good abstraction.

Self-service with guardrails

Self-service removes the queue; guardrails keep it safe. The difference from traditional controls is where the check happens. A **gate** is a person who must approve a request, such as a change advisory board or a database administrator reviewing a ticket. A **guardrail** is an automatic limit that lets anything through that is within policy and blocks or flags what is not, at the moment of the request. Guardrails are usually written as policy as code (introduced in section 10.4) and enforced in pipelines, admission controllers and cloud organisation policies.

Self-service capabilities and their guardrails

Capability	Self-service interface	Example guardrails
New service	Template in the portal or CLI	Must have an owner in the catalogue; must use the approved base image.
Database	Infrastructure module or portal form	Encryption at rest on; automated backups on; no public network access; size limited by team quota.
Deployment	Merge to main triggers the pipeline	Only signed images from the internal registry; resource requests and probes required; canary analysis required for tier-1 services.
Cloud spend	Team accounts or namespaces created on request	Mandatory cost tags; budget alerts at 80% and 100%; expensive instance types need a reason.
Access	Role requests through the portal	Production access is time-limited and logged; no long-lived personal keys.

Good guardrails share three properties: they give a clear message explaining what was blocked and how to fix it; they are versioned and tested like code, with a period of warnings before a new rule starts blocking; and they come with a documented route for justified exceptions, because a guardrail with no exception path turns back into a gate the first time it is wrong.

IF YOU ARE NEW TO THIS

Step by step: a gate or a guardrail?

The step most people trip over here is the difference between a gate and a guardrail. Both can enforce the same rules, but a gate puts a person in front of every request, while a guardrail checks automatically at the moment of the request. Follow one account request both ways.

- The restaurants team needs a cloud account for its new restaurant-ratings service. The rules say budget alerts must be on, every resource must carry cost tags, and expensive machine types need a stated reason.
- With a gate, the team raises a ticket. When it reaches the top of the queue, perhaps days later, a cloud operations engineer checks it against the rules and creates the account. The rules are enforced, but at the cost of waiting.
- With a guardrail, the team requests the account through the platform's portal. Automatic checks, written as policy as code, compare it with the same rules. It passes, and the account is created with no ticket or human approval.
- Later the team tries to start a server without cost tags. The check blocks it at once, with a message saying what was blocked, why, and how to fix it. The team adds the tags and tries again.
- If the team had a case the rules did not foresee, it would use the documented exception route. Without one, the first wrong rule leaves the team stuck, and the guardrail has turned back into a gate.
- **The lesson:** a guardrail does not weaken the rules; it moves the check from a person's queue to the moment of the request. It works when it explains itself, is tested like code and leaves a route for justified exceptions.

Reducing cognitive load, not hiding everything

Team Topologies borrows the idea of cognitive load from John Sweller's educational research and distinguishes three kinds. Intrinsic load is inherent in the task, such as understanding how delivery fees work. Extraneous load comes from the environment, such as remembering the twelve steps to request a database. Germane load is the effort of learning that makes future work easier, such as understanding the domain more deeply. A platform should strip out extraneous load so that teams can spend their limited attention on intrinsic and germane load in their own domain.

That does not mean hiding everything. A team that runs its service still needs to understand how it behaves in production: its resource use, its failure modes, its latency and its costs. Abstractions that hide too much leak at the worst moment, during an incident, when the team

discovers it cannot see the logs or does not know which database tier it is on. Good platforms are transparent: they provide sensible defaults and simple interfaces while letting teams look underneath and override when they need to.

Measuring whether the platform works

A platform's value shows up in other teams' outcomes, so its measures must look there. Useful measures fall into four groups, and a balanced set takes at least one from each.

Platform success measures

Group	Example measures	Watch out for
Adoption	Share of services on the golden path; voluntary adoption of new capabilities; teams leaving the path and why.	Adoption that is mandated says nothing about value.
Delivery	DORA metrics for teams on and off the platform; time from creating a service to its first production deployment; onboarding time for a new engineer.	Comparisons between teams doing very different work.
Developer experience	Regular surveys of satisfaction, perceived friction and time lost to tooling; interviews.	Survey fatigue; treat results as signals, not scores.
Platform health	The platform's own SLOs (pipeline availability, provisioning time), support load, cost per team.	A platform that is reliable but unused is still a failure.

The SPACE framework, published in ACM Queue in 2021 by Nicole Forsgren, Margaret-Anne Storey and colleagues from GitHub, Microsoft Research and the University of Victoria, warns that developer productivity cannot be captured by one number. It names five dimensions: satisfaction and well-being, performance, activity, communication and collaboration, and efficiency and flow. It recommends measuring several dimensions at once, including at least one perceptual measure such as a survey, and never judging individuals by activity counts such as commits or lines of code.

Evidence on platforms is encouraging but not unconditional. The 2024 DORA *Accelerate State of DevOps* report found that users of internal developer platforms reported about 8% higher individual productivity and 10% higher team performance, yet software delivery throughput and change stability were 8% and 14% lower. The report offers possible explanations, such as extra hand-offs and the platform layer itself adding steps, and the lesson for architects is to measure delivery outcomes as well as satisfaction, and to expect a dip while a platform matures.

WORKED EXAMPLE

Worked example: the first golden path for a new food-delivery service

The food-delivery company has twelve product teams. A survey and a week of shadowing show that creating a new service takes about three weeks and fourteen tickets to four different teams. A platform team of four is formed. The first product teams to need a new service are the restaurants team, which wants a restaurant-ratings service, and the couriers team.

Choose the path	Inventory of existing services: 70% are stateless HTTP services in Python or Go with a PostgreSQL database and, often, a queue subscription. That shape becomes golden path v1; everything else stays off-road for now.
Design with users	Two engineers from the restaurants and couriers teams join the platform team for three weeks (collaboration mode) to build the path with them, not for them.
Contents	Template repository; the standard pipeline with canary deployment; Kubernetes settings with requests, probes and autoscaling; a database module with encryption, backups and cost tags; logging, tracing and an SLO dashboard; a catalogue entry; a getting-started guide.
Guardrails	Owner and cost tags mandatory; signed images only; database without public access. All checked automatically; warnings for a month before blocking.
First result	The restaurants team creates restaurant-ratings from the portal and deploys a hello-world version to production in 5 hours, against three weeks before. Tickets needed: none.
Measure	Time to first production deployment, survey of the two pilot teams, and DORA metrics for services on the path, reviewed monthly. Three more teams adopt the path voluntarily within a quarter.
Next steps	Teams ask for event-consumer support and preview environments per pull request; the roadmap is ordered by demand. A request to support a niche language is declined for now and documented as off-road.

The platform started with one path for the most common case, built with its first users, and was judged by their outcomes rather than by the number of features it shipped.

Most of the work was integration, defaults and documentation of existing tools, not new software.

Risks and failure modes

Platform initiatives fail in recognisable ways. The commonest is **building a platform nobody wants**: a team spends a year building an ambitious general-purpose platform, launches it, and finds product teams already have workable solutions and no time to migrate. Starting thin, with real users and one golden path, is the antidote. The second is **the renamed operations team**: an infrastructure team is relabelled as a platform team but still works from a ticket queue, so nothing changes except the job titles. If routine requests still need a ticket and a person, it is not yet a platform.

Other risks are mandating adoption, which removes the feedback that keeps a platform useful; over-abstraction, in which a home-grown layer hides the underlying tools so completely that teams cannot debug or extend it and the platform team becomes a bottleneck for every change; neglecting the long tail of services created from old templates; and the platform becoming a single point of failure whose outage stops every deployment. Conversely, a small organisation may need no platform team at all: a few shared templates maintained by volunteers can be the thinnest viable platform for years.

Checklist: starting a platform

- [ ] **Evidence of pain**
  Measure how long common tasks take today (new service, new database, first deployment) and how many tickets and teams they involve.
- [ ] **A product owner**
  Someone owns the roadmap, talks to users regularly and can say no.
- [ ] **One golden path first**
  Pick the most common service shape and make it excellent end to end before adding breadth.
- [ ] **Self-service, not tickets**
  Every routine request on the path completes without a human approval; exceptions have a documented route.
- [ ] **Guardrails as code**
  Security, cost and reliability rules enforced automatically, with clear messages and a warning period.
- [ ] **Transparent abstractions**
  Teams can see logs, metrics, costs and the underlying resources, and override defaults when justified.
- [ ] **Outcome measures**
  Adoption, time to first deployment, DORA metrics and developer surveys reviewed regularly and published.
- [ ] **Upgrade path**
  Services created from older templates receive improvements, and deprecations come with notice and migration help.

CHECK YOUR UNDERSTANDING

▼ **What is the difference between a gate and a guardrail?**

A gate needs a person to approve each request; a guardrail is an automatic check that lets compliant requests through immediately and blocks or flags the rest.

▼ **Why should a golden path be optional?**

So that teams with unusual needs are not blocked, and so that the platform team learns from who leaves the path whether it is actually better than the alternatives.

▼ **A company installs Backstage, but creating a database still needs a ticket and a week's wait. What has it built?**

A portal in front of a manual process, not a platform; self-service capabilities behind the portal are what create value.

▼ **Why does the SPACE framework advise against measuring developer productivity by commits or lines of code alone?**

Activity counts capture one dimension, are easy to game and say nothing about satisfaction, collaboration, flow or the value of the outcome.

▼ **Try it: Leadership renames the operations team "Platform", mandates that all teams use it, and measures success by the number of tickets closed. Requests for a new service still take fourteen tickets. Critique this and say what you would change.**

This is a renamed ticket-driven team, not a platform. Treat it as a product: give it a product owner, research what product teams need, and build a thin golden path for the commonest task, a new stateless service, as self-service with automatic guardrails. Make it optional and earn adoption. Measure others' outcomes, such as time to first deployment, adoption, DORA metrics and developer experience. Fewer tickets is the goal; counting closures rewards the wrong thing.

SOURCES

- Evan Bottcher: What I Talk About When I Talk About Platforms, martinfowler.com: https://martinfowler.com/articles/talk-about-platforms.html
- CNCF TAG App Delivery: Platforms White Paper: https://tag-app-delivery.cncf.io/whitepapers/platforms/
- Backstage documentation: What is Backstage?: https://backstage.io/docs/overview/what-is-backstage
- Forsgren et al.: The SPACE of Developer Productivity, ACM Queue (2021): https://queue.acm.org/detail.cfm?id=3454124

SECTION 10.8 OF 10.8 · INTERMEDIATE · 23 MIN READ · LAST REVIEWED NOVEMBER 2026

FinOps, cost architecture and vendor lock-in

BEFORE YOU READ THIS

This section builds on:

- 2.7 Security, usability and more
- 5.6 Hexagonal and clean
- 10.1 Cloud models

Terms it uses from earlier: Managed service, Multi-cloud, Resource requests and limits, Ports and adapters (hexagonal architecture), Serverless

THE IDEA IN BRIEF

Renting a car for a day is cheap; renting one every day for three years is far dearer than buying. Leaving the engine running in the car park costs fuel for nothing. And if you lease a car with a special charging plug, switching brands later means new chargers too. Cloud computing raises all three questions at once. FinOps is the household budget habit for the cloud: see what each part costs, cut waste, buy wisely. Lock-in is the special-plug question: some dependence is worth it, as long as you chose it on purpose and know how you would leave.

WHAT YOU NEED TO KNOW

- In the cloud every architectural decision is also a purchasing decision, so cost is a quality attribute that changes with each deployment and is best expressed per unit of business value, such as cost per order.
- FinOps, defined by the FinOps Foundation (a Linux Foundation project), shares financial accountability between engineering, finance and the business through three iterative phases: inform, optimise and operate.
- Common cost drivers are data egress and cross-zone transfer, idle and over-provisioned resources, logging volume, managed-service premiums and unmanaged storage growth; most are shaped by architecture.
- Commitment discounts suit only the certain baseline, spot capacity suits interruptible and retriable work, and on-demand autoscaling covers peaks; rightsize before committing.
- Tagging enables allocation; showback reports costs to teams, chargeback charges their budgets and needs accurate allocation and agreed rules for shared costs.

- Lock-in is a trade-off: Hohpe (2019) distinguishes many kinds and weighs switching cost times likelihood against the cost of avoiding it; accept it where it buys unique value.
- Portability has real costs; keep business logic behind ports at likely seams, keep data exportable and maintain a tested exit strategy, which EU financial regulation now requires for critical services.

Cost is an architectural quality

Section 2.7 (security, usability and more) listed cost among the quality attributes and recommended expressing it per unit of work. In a data centre, cost was mostly fixed: hardware was bought every few years and an architect's choices barely moved the monthly bill. In the cloud, every architectural decision is also a purchasing decision, made continuously by engineers rather than once a year by procurement. A chatty service boundary, a debug log level, a forgotten test environment or a synchronous cross-region call each shows up on next month's invoice. Cost has become a quality attribute that changes with every deployment.

Two disciplines help. FinOps is the operating practice that makes cloud spending visible, owned and managed. Cost-aware architecture is the design practice that makes systems cheap to run per unit of value in the first place. This section covers both, and then turns to their awkward neighbour: lock-in, the cost of leaving.

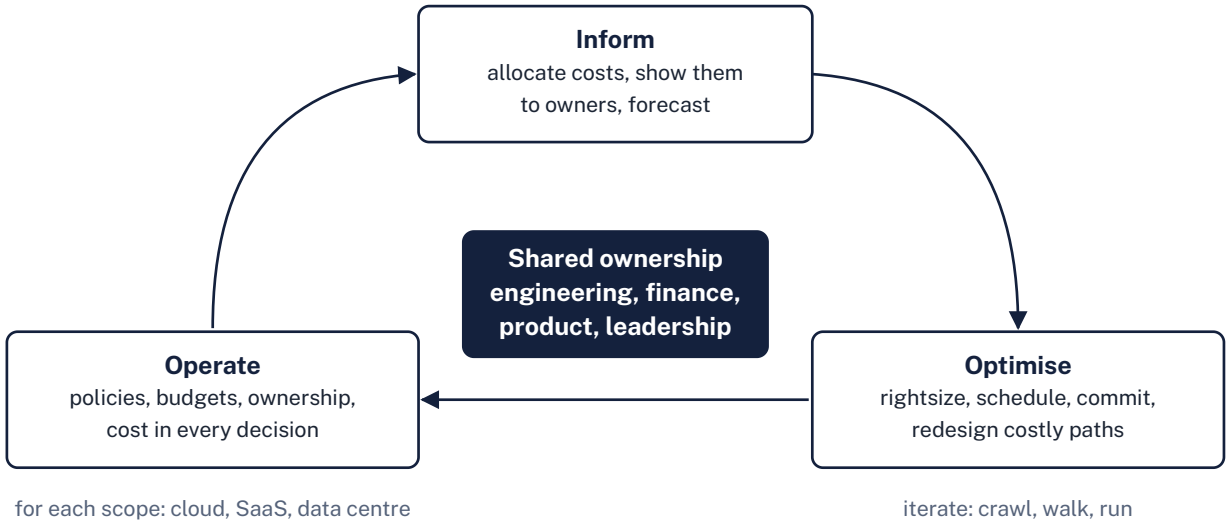
FinOps: phases, principles and who does what

FinOps is defined by the FinOps Foundation, a Linux Foundation project, as an operational framework and cultural practice that maximises the business value of technology through timely, data-driven decisions and financial accountability shared between engineering, finance and the business. The name deliberately echoes DevOps: just as DevOps broke down the wall between building and running software, FinOps breaks down the wall between spending money and deciding how software is built.

The FinOps Framework organises the work into three iterative phases. **Inform**: make costs visible, allocate them to the teams and products that cause them, forecast and benchmark. **Optimise**: reduce waste and improve rates, by rightsizing, scheduling, commitments and design changes. **Operate**: build the habits that keep costs under control, such as budgets, policies, ownership and cost considered in every design decision. Organisations move through the cycle repeatedly, maturing from crawl to walk to run in each capability. The framework's principles include that teams collaborate, that business value drives technology decisions, that everyone takes ownership of their technology use, that cost data is accessible, timely and accurate, that

FinOps is enabled centrally, and that the variable cost model of the cloud should be used to advantage. Recent versions extend the framework beyond public cloud to other technology spending, such as SaaS licences, data centres and AI services.

The FinOps cycle



FinOps is a repeating cycle, not a one-off cost-cutting project, and it rests on shared ownership rather than a central team policing spend.

Unit economics: cost per order

A cloud bill that rises from 80,000 to 100,000 a month might be a disaster or a triumph, depending on whether the business grew by 5% or by 50%. Unit economics expresses cost per unit of business value: cost per order for the food-delivery platform, per active customer for a streaming service, per thousand API calls for a developer platform. It separates efficiency from growth, connects engineering decisions to business outcomes and gives teams a target they can influence.

Computing it well needs three things. First, a **unit that matters to the business**, chosen with finance and product, not invented by engineers. Second, **cost allocation**: direct costs (the order service's compute and database) assigned through tags or namespaces, and shared costs (the Kubernetes control plane, the observability stack, the platform team's tools) split by an agreed rule such as proportional usage. Third, a distinction between **average and marginal cost**: the average cost per order includes fixed costs that do not grow with orders, so the cost of the next thousand orders is usually lower than the average suggests, which matters when pricing a promotion or forecasting growth.

IF YOU ARE NEW TO THIS

Step by step: what the next thousand orders cost

This section's hardest idea is the difference between average and marginal cost. The average spreads the whole bill over every order; the marginal cost is what one more order adds. Because part of the bill is fixed, the two can differ widely. Follow one promotion through both, with illustrative figures.

- The food-delivery platform's monthly cloud bill is £96,000 for 1.2 million orders, so its cost per order, the unit economics measure this section recommends, is £0.080. That is the average.
- Part of the bill does not move with orders: the orders database, sized in advance, the always-on servers for the quiet hours, the Kubernetes control plane and the platform team's tools. Say £36,000 is fixed like this and £60,000 grows with orders: mostly autoscaled compute, data transfer and logs.
- So each extra order adds about £60,000 divided by 1.2 million, or £0.050: the marginal cost, well below the average.
- The business plans a promotion expected to bring 200,000 extra orders a month. Priced at the average, serving them looks like 200,000 x £0.080 = £16,000. Priced at the marginal cost, it is about £10,000, as long as the extra orders fit within the fixed capacity already paid for.
- Marginal cost is not always lower. If the promotion pushed the orders database past its current size, moving to a larger size would add a fixed step, and the first orders over that line could cost far more than the average.
- **The lesson:** track efficiency with the average cost per order, but price extra volume, such as a promotion or a growth forecast, with the marginal cost. First ask which costs move with orders and which do not.

Where the money goes: common cost drivers

Most surprising cloud bills come from a handful of drivers, and nearly all of them are shaped by architecture rather than by negotiating better prices.

Common cost drivers and architectural responses

Driver	Why it grows	Architectural responses
Data egress and transfer	Providers charge per gigabyte for data leaving to the internet, and often for traffic between zones and regions and through managed NAT gateways. Chatty services spread across zones pay on every call.	Serve public content through a CDN; keep chatty services zone-local (topology-aware routing); use private endpoints to provider services; compress; avoid synchronous cross-region traffic.
Idle and over-provisioned resources	Test environments running 168 hours a week for 50 hours of use; container resource requests set far above real usage; orphaned volumes, snapshots and load balancers.	Rightsizing from measured usage; schedules and autoscaling to zero for non-production; time-limited preview environments; automated clean-up of unattached resources.
Logging and observability volume	Ingestion and retention are priced per gigabyte; debug logs in production, verbose request logging and high-cardinality metric labels grow with traffic.	Log levels and sampling for successful requests; tiered retention (hot for days, archive for months); metrics instead of logs for counts; trace sampling.
Managed-service premiums	A managed service costs more per unit of capacity than running the software yourself; per-request pricing grows linearly with volume.	Compare total cost of ownership including people and risk; review per-request services at high steady volume (see serverless); use them where the premium buys real reliability or speed.
Storage growth	Data that is never deleted, object versions, duplicated copies and old backups accumulate quietly.	Lifecycle policies to cheaper tiers and deletion; retention rules agreed with the business and the law; deduplicated, incremental backups.

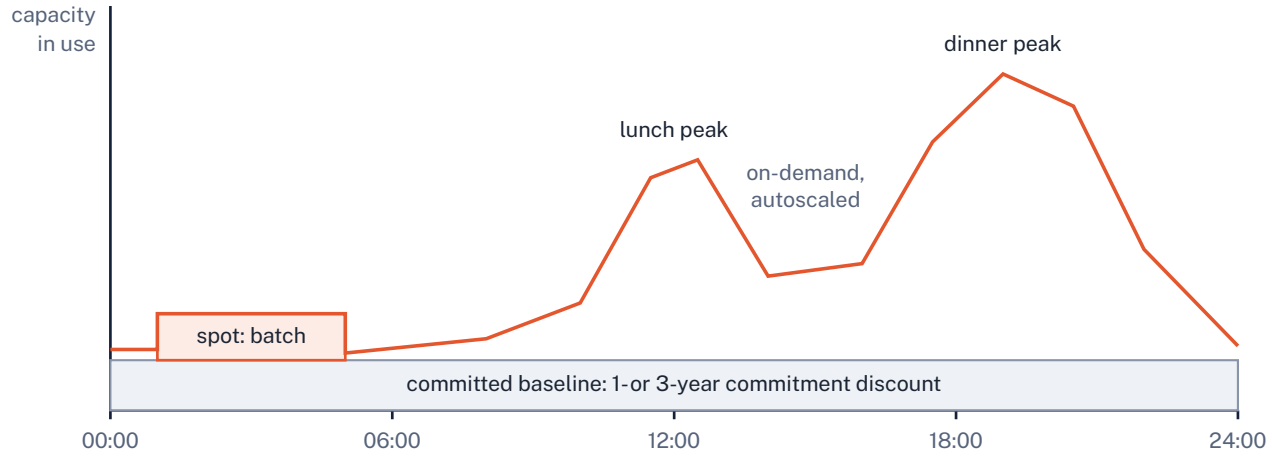
Paying less for the same capacity

Once waste is removed, the rate paid for the remaining capacity can be lowered. On-demand capacity is the most flexible and most expensive. A commitment discount exchanges flexibility for price: AWS Reserved Instances and Savings Plans, Azure reservations and savings plans, and Google Cloud committed use discounts offer substantial reductions (AWS quotes up to 72% for some commitments) in return for committing to a level of usage or spend for one or three years. Commitments should cover only the baseline that will certainly be needed for the term; committing to peak capacity converts a discount into waste.

Spot capacity (AWS Spot Instances, Azure Spot Virtual Machines, Google Cloud Spot VMs) sells spare capacity at large discounts, often 60% to 90%, but the provider can reclaim it at short notice: two minutes on AWS, about 30 seconds on Azure and Google Cloud. It suits work

that is stateless, interruptible and retriable, such as batch processing, CI runners, rendering and some stateless web tiers spread across many instance types. It does not suit databases or anything that cannot be stopped mid-task. Architecture decides how much of a workload can use spot: idempotent, checkpointed jobs can; long, stateful ones cannot.

Matching pricing models to the shape of demand



Commit to the floor, autoscale on demand for the peaks, and push interruptible work onto spot capacity at quiet times.

Tagging, showback and chargeback

Nobody optimises a cost they cannot see. Allocation starts with a tagging (or labelling) policy: every resource carries a small set of mandatory tags, such as owner team, service, environment and cost centre, enforced by guardrails in infrastructure code and cloud organisation policies, because tags added by hand are soon missing. For Kubernetes, costs are allocated by namespace and label with tools such as the open-source OpenCost project or the providers' own cost allocation features. The FinOps Foundation's FOCUS specification defines a common format for billing data so that costs from several providers can be combined.

With costs allocated, an organisation chooses how to use them. Showback reports each team's costs to it without touching its budget; it builds awareness and is the usual starting point. Chargeback charges the costs to each team's budget, which creates stronger incentives but needs accurate allocation, agreed rules for shared costs and a way to handle disputes. Either way, publish the share of spend that cannot be allocated: an untagged share above a few per cent means the numbers are not yet trustworthy enough to act on.

Architectural decisions that drive cost

The largest savings come from design, not discounts, and they are cheapest to obtain before a system is built. Each of the following decisions has a predictable cost profile. **Service granularity**: every network hop costs compute, serialisation and possibly data transfer, so fine-grained services in many zones cost more per request than a modular monolith. **Synchronous fan-out**: a request that calls ten services keeps eleven sets of resources busy for its duration. **Environment count**: full copies of production for every team and branch multiply the bill; shared environments with isolation or short-lived previews cost far less. **Data retention and replication**: every extra copy and every extra region multiplies storage and transfer. **Caching and CDNs**: a cache hit is usually far cheaper than recomputing, and a CDN hit avoids egress from the origin. **Compute model**: per-request serverless pricing is cheaper than always-on capacity below a crossover volume and dearer above it.

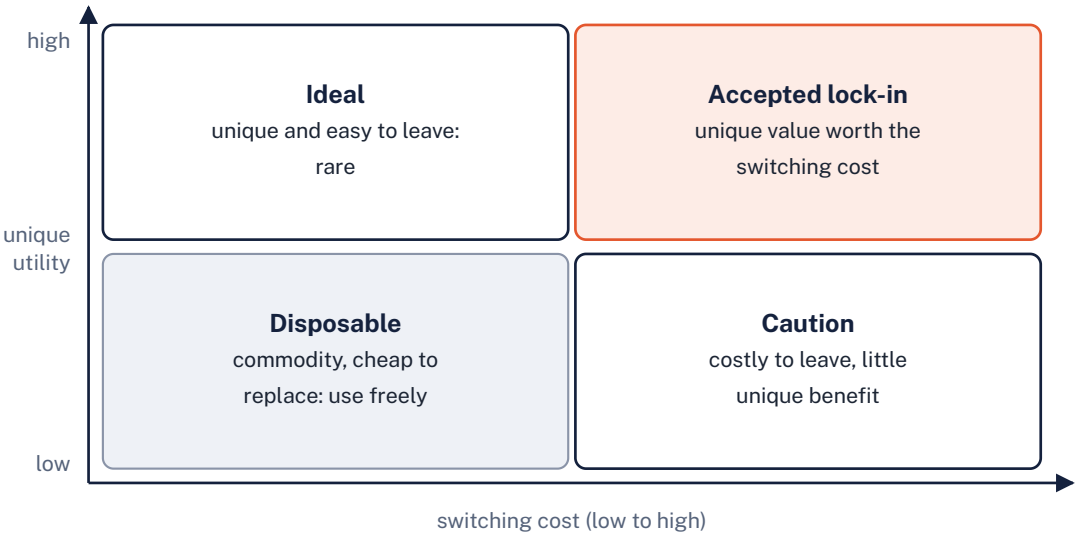
Record significant cost consequences in decision records alongside the other quality trade-offs, and consider adding cost to the fitness functions that guard the architecture: for example, a monthly check that cost per order has not risen by more than 10% without an explanation.

Lock-in is a trade-off, not a sin

Lock-in is the cost and difficulty of leaving something once you depend on it. It is usually discussed as something to avoid at all costs; Gregor Hohpe's 2019 article on martinfowler.com, *Don't get locked up into avoiding lock-in*, argues that this is itself a trap. He distinguishes several kinds of lock-in: vendor (contracts and commercial terms), product (a particular product's APIs and features, even if open source), version, architecture (a style such as serverless), platform (accounts, identity and security models), skills (what your people know), legal (compliance and licensing) and mental (assumptions that stop you considering alternatives). Using open-source software reduces vendor lock-in but leaves most of the others.

The key move is to treat lock-in as an option-pricing problem. The switching cost is what it would cost to move; multiplied by the likelihood of ever moving, it gives the expected cost of being locked in. Reducing the switching cost has a price of its own: extra abstraction, lowest-common-denominator designs and forgone features. A switch that would cost 50,000 and has a 5% chance of happening carries an expected cost of 2,500, so spending far more than that to avoid it is a poor bargain. Hohpe's matrix of unique utility against switching cost turns this into a quick classification.

Unique utility against switching cost



After Gregor Hohpe (2019): lock-in is acceptable when it buys unique value; be wary of costly dependencies that give little in return.

The cost of portability and the exit strategy

Portability is not free. Designing every component to run on several clouds means using only features they all share, building and testing abstraction layers that leak (see information hiding and leaky abstractions), keeping skills in several platforms and giving up managed services that would have saved effort. Many multi-cloud strategies end up with the costs of all their providers and the advantages of none. Portability is cheapest where open standards already exist: OCI container images, Kubernetes manifests, PostgreSQL-compatible databases, S3-compatible object storage APIs, standard protocols such as HTTP, AMQP and the Kafka protocol, and OpenTelemetry for observability. Even the tools used to stay portable, such as Terraform or OpenTofu, are dependencies in their own right.

Practitioners disagree on how far to go. Advocates of portability point to price rises, outages, sanctions, regulation and the cases where moving out of the cloud paid off (see cloud repatriation in section 10.1). Advocates of committing to one provider point to the speed and reliability gained from using managed services fully. A sound middle position is to accept lock-in deliberately where it buys clear value, keep business logic independent of provider APIs behind ports and adapters at the few seams most likely to change, keep data in formats you can export, and write down an exit strategy.

Regulation increasingly requires that last step. The EU's Digital Operational Resilience Act (Regulation (EU) 2022/2554, applying from 17 January 2025; not to be confused with the DORA delivery metrics) requires financial entities to have exit strategies for ICT services that support critical or important functions. The EU Data Act's cloud-switching rules have applied since 12

September 2025: customers may switch provider with at most two months' notice, providers may charge only their direct costs for switching, including data egress, until 12 January 2027, and nothing after that. In 2024 the largest providers also announced free data transfer for customers leaving them, subject to conditions. An exit strategy should name the replacement for each critical service, estimate the time and cost to move, show that data can be exported in a usable format, and be rehearsed at least partly, for example by restoring a backup into another environment.

WORKED EXAMPLE

Worked example: cost per order and three architectural changes

The food-delivery platform handles 1.2 million orders a month. Its monthly cloud bill is 96,000: compute 42,000, managed databases 18,000, observability 14,000, data transfer 12,000, other managed services 6,000 and storage 4,000. All figures are illustrative, in pounds.

Baseline	96,000 / 1,200,000 orders = 0.080 per order. Showback reveals that compute requests are about twice measured usage, half the log volume is debug output, and the order and pricing services exchange about 40 calls per order across availability zones.
Change 1: rightsize, then commit	Requests set from measured p95 usage cut node count by 25%: compute 42,000 to 31,500. The steady 60% of that (18,900) is covered by a one-year commitment at about 30% off, saving 5,670. Compute becomes 25,830, a saving of 16,170.
Change 2: log less, keep it shorter	Debug logging off in production, successful requests sampled at 10%, hot retention cut from 90 to 14 days with an archive tier. Ingestion falls by about 60%: observability 14,000 to 6,500, a saving of 7,500.
Change 3: stop paying for distance	Pricing logic moved into the order service's process for the hot path (removing most of the 40 calls), zone-aware routing for the remaining calls, and restaurant images served from a CDN instead of the origin. Data transfer 12,000 to 6,000; CDN adds 1,000. Net saving 5,000.
Result	96,000 minus 28,670 = 67,330 a month, or 0.056 per order: 30% lower per order, with no change in features or reliability targets.
What did not change	Database and storage costs were left alone this quarter because the savings available there were small relative to the risk; they remain in the backlog.

The largest structural change was architectural (removing a chatty boundary and introducing a CDN), the logging change was a design decision about observability, and the rate discount was applied only after rightsizing, so the company did not commit to waste.

Tracking cost per order rather than the total bill shows whether future growth keeps the improvement: if orders double and cost per order stays near 0.056, efficiency has held.

Questions people ask

Should every team be charged for its cloud costs?
Start with showback, so that teams see their costs and can act, and move to chargeback only when allocation is accurate and shared costs are split by rules everyone accepts; charging inaccurate numbers creates arguments, not savings.

Is multi-cloud the best protection against lock-in?
Rarely on its own. Running the same system on several clouds is expensive and limits you to common features. Using different providers for different, independent workloads, plus a tested exit strategy for critical ones, usually gives most of the protection at a fraction of the cost.

When is cost optimisation premature?
When the product has not found its market and engineering time is the scarcest resource. Even then, tagging from day one and a rough cost per unit are cheap and make later optimisation far easier.

CHECK YOUR UNDERSTANDING

▼ **Why is cost per order more useful than the total monthly bill?**

It separates efficiency from growth: a rising bill with a falling cost per order is healthy, while a flat bill with falling orders is not.

▼ **Why should rightsizing come before buying commitment discounts?**

A commitment locks in a level of spend for one or three years; committing before removing over-provisioning turns waste into a long-term obligation.

▼ **What kind of workload suits spot capacity, and why?**

Stateless, interruptible and retriable work such as batch jobs and CI runners, because the provider can reclaim spot capacity with seconds or minutes of notice.

▼ **According to Hohpe, when is lock-in acceptable?**

When it buys unique utility worth the switching cost, or when switching is so unlikely that the expected cost is lower than the cost of avoiding the lock-in.

▼ **Try it: The bill is £60,000 a month for 1,500,000 orders: compute £30,000, logging £9,000. Rightsizing cuts compute by 30%; dropping debug logs halves logging. Find cost per order before and after. Which should come before any commitment discount, and why?**

Before: £60,000 / 1,500,000 = 4.0 pence. The savings are £9,000 and £4,500, so the bill falls to £46,500, or 3.1 pence. Rightsize first, because a commitment on oversized capacity locks in the waste for its whole term. Then commit only to the measured baseline. Report cost per order to teams through showback, because a rising total may simply mean more business, while a rising unit cost signals a real problem.

SOURCES

- FinOps Foundation: FinOps Framework: https://www.finops.org/framework/
- Gregor Hohpe: Don't get locked up into avoiding lock-in, martinfowler.com: https://martinfowler.com/articles/oss-lockin.html
- AWS Well-Architected Framework: Cost Optimization Pillar: https://docs.aws.amazon.com/wellarchitected/latest/cost-optimization-pillar/welcome.html
- Regulation (EU) 2023/2854 (Data Act), EUR-Lex: https://eur-lex.europa.eu/eli/reg/2023/2854/oj

Further reading

- *The NIST Definition of Cloud Computing (SP 800-145)*, Peter Mell and Timothy Grance (2011), National Institute of Standards and Technology. The short, widely cited definition of the essential characteristics, service models and deployment models.
  https://csrc.nist.gov/pubs/sp/800/145/final
- *The Twelve-Factor App*, Adam Wiggins (2011), 12factor.net (Heroku; community-maintained since 2024). Twelve concise practices for deployable, scalable applications; still the common vocabulary of cloud-native application design.
  https://12factor.net/
- *Kubernetes: Up and Running (3rd edition)*, Brendan Burns, Joe Beda, Kelsey Hightower and Lachlan Evenson (2022), O'Reilly Media. A clear introduction to Kubernetes objects and operations by people who helped create it.
- *Infrastructure as Code (3rd edition)*, Kief Morris (2025), O'Reilly Media. The standard book on designing, testing and delivering infrastructure code, independent of any one tool.
- *Continuous Delivery*, Jez Humble and David Farley (2010), Addison-Wesley. The foundational text on deployment pipelines, build once deploy many, and making releases routine.
- *Accelerate: The Science of Lean Software and DevOps*, Nicole Forsgren, Jez Humble and Gene Kim (2018), IT Revolution. The research behind the DORA metrics, linking delivery practices and loosely coupled architecture to organisational performance.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (eds) (2016), O'Reilly Media (free online). How a large operator runs systems with automation, error budgets and release engineering.
  https://sre.google/books/
- *Don't get locked up into avoiding lock-in*, Gregor Hohpe (2019), martinfowler.com. A balanced analysis of the kinds of lock-in and the real cost of avoiding them.
  https://martinfowler.com/articles/oss-lockin.html

Glossary

Active-active (multi-site)
An arrangement in which two or more regions serve live traffic at the same time, so losing one region needs no failover, only redistribution of traffic.

Artefact repository
A store for built, versioned outputs such as container images and packages, from which every environment deploys the same file.

Automated canary analysis
Statistical comparison of a canary's metrics with those of a baseline running the old version, used by the pipeline to promote or roll back without a human judgement.

Backing service
Any service an application uses over the network, such as a database, queue, cache or email API, treated as an attached resource that configuration can swap.

Blue-green deployment
Running two complete production environments and switching all traffic from the old one (blue) to the new one (green) at once, keeping the old one ready for an instant switch back.

Build once, deploy many
The rule that a change is built into one immutable artefact exactly once, and that same artefact is promoted through every environment with only configuration differing.

Build, release, run
The strict separation of turning code into an artefact (build), combining it with an environment's configuration (release) and executing it (run).

Canary release
Sending a small share of real traffic to a new version first, comparing its behaviour with the old version, and widening the share only if it is healthy.

Cell-based architecture
Running many complete, independent copies (cells) of a system, each serving a fixed subset of customers, behind a thin router, so that a failure affects only one cell.

Chargeback
Charging each team's or product's budget for the costs attributed to it, so that the spending appears in its own accounts.

Cloud native
An approach to building and running software as loosely coupled, automated, observable and resilient systems that are deployed programmatically and repeatably, in public, private or hybrid clouds. It describes how a system is built, not where it runs.

Cloud repatriation
Moving workloads out of a public cloud back to owned or colocated hardware, usually because steady load makes ownership cheaper.

Cognitive load
The amount of mental effort a person or team must spend to understand and work within a system; Team Topologies uses it to size team responsibilities.

Commitment discount
A lower price in exchange for committing to a level of usage or spend for one or three years, such as reserved instances, savings plans or committed use discounts.

Configuration drift
Differences that build up between the intended configuration in code and the real infrastructure, usually from manual changes or partial updates.

Configuration management
Tools such as Ansible, Chef and Puppet that bring existing, long-lived servers into a described configuration by installing packages and editing files on them.

Container orchestration
Automatically scheduling containers onto machines and keeping them running: restarting failures, scaling, rolling out new versions, and connecting them to networks and storage.

Containers as a service (CaaS)
A cloud service that runs container images for you, either as managed Kubernetes or as a serverless container platform where you never see the machines.

Control plane
The components that make decisions for a cluster or system (in Kubernetes: the API server, etcd, scheduler and controllers), as opposed to the nodes that run the workloads.

Data egress
Data leaving a cloud provider's network, or crossing between its zones or regions, which is usually charged per gigabyte.

Deployment (Kubernetes)
A Kubernetes object that declares how many identical pods of a stateless application should run and rolls out new versions gradually through ReplicaSets.

Deployment pipeline
The automated sequence of stages (build, tests, packaging, deployment, verification) that every change passes through on its way from a commit to production.

Desired state
A declarative description of what a system should look like (for example, three copies of version 2.4), which automation compares with the actual state and works to achieve.

Developer portal
A web interface, such as Backstage, that gathers a software catalogue, templates, documentation and links to platform services in one place.

Disaster recovery (DR)
The plans, copies and procedures that restore a service and its data after a large failure, such as the loss of a whole data centre or region.

Disposability
The property of processes that start quickly, shut down gracefully and can be killed at any time without harm, so they can be replaced and scaled freely.

Edge function
A small piece of code run by a CDN or edge provider in locations close to users, to handle or change requests before (or instead of) reaching the origin.

Exit strategy
A documented and ideally rehearsed plan for leaving a provider or product, covering data export, replacement services, contracts, time and cost.

FinOps
A practice in which engineering, finance and business teams share responsibility for cloud and technology spending, using timely cost data to make decisions that maximise business value.

Gateway API
The Kubernetes API for routing external and internal traffic into services, generally available since 2023 and the recommended successor to Ingress.

GitOps
Operating systems by keeping their declarative desired state in Git and letting agents pull it and continuously reconcile the running system to match it.

Golden path (paved road)
An opinionated, supported and documented way to do a common task, such as creating a new service, that is the easiest way but not the only permitted way.

Guardrail
An automatic limit or check that keeps self-service use safe (for example, a policy that blocks public storage buckets) without requiring a person to approve each request.

Horizontal Pod Autoscaler (HPA)
A Kubernetes controller that changes the number of pod replicas to keep a metric, such as average CPU use relative to requests, near a target.

Hybrid cloud
A combination of private and public cloud (or on-premises systems) connected so that data and applications can work across them.

Immutable infrastructure
The practice of never changing a running server or container in place; every change is made by building a new image and replacing the old instances.

Infrastructure as code (IaC)
Defining servers, networks, databases and other infrastructure in version-controlled files that tools apply automatically, instead of configuring them by hand.

Internal developer platform (IDP)
The set of self-service tools, services, templates and documentation that an organisation's platform team provides so that product teams can build and run software without tickets.

Kubernetes
The most widely used open-source container orchestrator, created at Google, announced in 2014 and released as version 1.0 in 2015; it keeps a cluster's actual state matching a declared desired state.

Lock-in
The cost and difficulty of switching away from a vendor, product, platform or technology once a system depends on it.

Multi-cloud
Using services from two or more public cloud providers, for distinctive services, acquisitions, leverage or regulatory exit plans.

Open Container Initiative (OCI)
A Linux Foundation project, founded in 2015, that maintains open standards for container image formats, runtimes and registries, so that tools from different vendors interoperate.

OpenTofu
An open-source fork of Terraform created in 2023 after HashiCorp moved Terraform to the Business Source Licence; hosted by the Linux Foundation and a CNCF project.

Pilot light
A disaster recovery strategy that keeps data replicated and core infrastructure provisioned in a second region, but starts and scales the application servers only during a disaster.

Platform engineering
The discipline of building and running internal platforms that let product teams provision, deploy and operate software through self-service.

Pod
The smallest deployable unit in Kubernetes: one or more containers scheduled together on one node, sharing a network address and storage volumes.

Policy as code
Writing rules about infrastructure and deployments (no public storage, required encryption, approved regions) as code that tools check automatically before or during changes.

Private cloud
Cloud infrastructure dedicated to one organisation, offering self-service, API-driven provisioning; it may be on premises or hosted by a third party.

Probe (liveness, readiness, startup)
A periodic check by Kubernetes on a container: liveness decides whether to restart it, readiness whether to send it traffic, startup whether it has finished starting.

Progressive delivery
Releasing a change to gradually larger audiences with automated checks at each step, so that a bad change reaches as few users as possible before it is stopped.

Reconciliation loop
A controller's endless cycle of observing the actual state, comparing it with the desired state and acting to remove the difference.

Resource requests and limits
Per-container CPU and memory settings in Kubernetes: requests reserve capacity and guide scheduling; limits cap use, throttling CPU and killing containers that exceed memory.

Rightsizing
Matching the size of provisioned resources (machine types, container requests, database tiers) to what a workload actually uses.

Roll forward
Recovering from a bad release by deploying a new fix rather than returning to the previous version.

Rolling deployment
Replacing the instances of a service a few at a time, so that old and new versions run side by side until all instances run the new version.

Service (Kubernetes)
A Kubernetes object that gives a changing set of pods one stable name and virtual address, and spreads traffic across them.

Shadow traffic (dark launch)
Copying real requests to a new version whose responses are compared or discarded but never returned to users, to test it under real load without user impact.

Shared responsibility model
The division of security and operational duties between a cloud provider, which secures the cloud itself, and the customer, which secures its data, identities and configuration in the cloud.

Showback
Reporting to each team or product the costs attributed to it, without actually charging its budget.

Shuffle sharding
Assigning each customer to a small, randomly chosen combination of resources, so that two customers rarely share all of theirs and one bad customer affects few others.

SPACE framework
A 2021 framework for developer productivity covering satisfaction and well-being, performance, activity, communication and collaboration, and efficiency and flow.

Spot capacity
Spare cloud capacity sold at a large discount that the provider can take back at short notice; suitable only for work that tolerates interruption.

State (infrastructure as code)
The record an infrastructure tool keeps of which real resources correspond to which definitions in code, used to work out what to create, change or delete.

Static stability
The property that a system keeps working during a failure without having to make changes, such as launching new servers or updating configuration, at the moment of the failure.

Switching cost
The total effort, money and risk needed to move from one technology or provider to another.

Thinnest viable platform
The smallest set of shared services and documentation that genuinely speeds up product teams; Team Topologies' warning against building more platform than needed.

Trunk-based development
A way of working in which developers integrate small changes into one shared main branch at least daily, instead of keeping long-lived feature branches.

Twelve-factor app
A methodology of twelve practices for building deployable, scalable web applications, written by Adam Wiggins at Heroku in 2011 and open-sourced in 2024.

Unit economics
Expressing cost (and revenue) per unit of business value, such as cost per order or per active customer, so that efficiency can be judged independently of growth.

Warm standby
A disaster recovery strategy that keeps a scaled-down but fully working copy of the system running in a second region, ready to take traffic and scale up.