

BE SOFTWARE ARCHITECTURE WISE · CHAPTER 11

Evaluation, Decision-Making and Governance

This chapter is about judging, governing and evolving architectures over time. It covers formal evaluation with the ATAM, lightweight reviews such as risk-storming and pre-mortems, fitness functions and evolutionary architecture, governance in agile organisations, Team Topologies and socio-technical design, legacy modernisation with the strangler fig, leadership and influence, and the business and ethical case for architecture work. It draws together every earlier chapter, uses the food-delivery platform as a running example, and is written for experienced developers, technical leads and architects who make or guide significant decisions.

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

AT A GLANCE

You will learn to

- Explain when and why to evaluate an architecture, and run the nine steps of the ATAM to produce risks, non-risks, sensitivity points, trade-off points and risk themes.
- Facilitate lightweight reviews such as risk-storming and pre-mortems, and turn their results into a risk register with owners.
- Design fitness functions that turn quality attribute scenarios into automated checks of structure, performance, security, resilience and data.
- Write architecture principles and choose between guardrails, gates and the advice process to govern different kinds of decision.
- Apply Team Topologies' team types, interaction modes and cognitive load to align teams with the intended architecture.
- Plan an incremental legacy modernisation using the strangler fig, branch by abstraction and parallel runs.
- Lead and influence architectural decisions without formal authority, including when communicating with executives.
- Calculate net present value and payback for a business case, and review an architecture for ethical and sustainability concerns.

Before you start

Chapters 1 to 4, especially decisions and trade-offs (Section 1.6), technical debt (Section 1.7), quality attribute scenarios and utility trees (Sections 2.3 and 2.8), Conway's law (Section 3.7) and decision records (Section 4.5). Chapters 5 to 10 help but are not essential.

Time

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

Key terms

ATAM · Utility tree · Sensitivity point · Trade-off point · Risk-storming · Pre-mortem · Fitness function · Evolutionary architecture · Advice process · Technology radar · Team Topologies · Strangler fig · Net present value · Software Carbon Intensity

Judging, governing and evolving architecture

The earlier chapters taught how to describe what a system must achieve, how to structure it, and how to make it fast, reliable, secure and economical. This chapter is about the work that surrounds those decisions over a system's whole life: checking that they are sound before money is committed, keeping them true as hundreds of changes land every month, deciding who may make which decisions in an organisation of many teams, reshaping teams and legacy systems, persuading people, and justifying the cost of architecture work in business and

ethical terms. It is where the course comes together, so it links back often: to quality attribute scenarios and utility trees in Chapter 2, to principles and metrics in Chapter 3, to decision records in Chapter 4, to Conway's law in Section 3.7 and to technical debt in Section 1.7.

The level is advanced, but every section begins with a short explanation for newcomers, and the running example is the food-delivery platform used since Chapter 2: customers, restaurants and couriers, services for ordering, dispatch and notifications, an event broker and an external payment provider. You will see its architecture evaluated, risk-stormed, protected by fitness functions, governed through principles and a technology radar, reorganised around teams and partly modernised.

Evaluating and protecting decisions: Sections 11.1 to 11.3

Section 11.1 covers formal architecture evaluation and the SEI's ATAM: why and when to evaluate, its nine steps across two meetings, who takes part, and its distinctive outputs: risks, non-risks with their assumptions, sensitivity points, trade-off points and risk themes. The utility tree from Section 2.8 returns in its original setting, and the CBAM shows how to weigh the benefit of each fix against its cost. A condensed ATAM of the food-delivery platform shows the tables an evaluation actually produces.

Section 11.2 explains why monthly review boards so often fail and what works instead: Simon Brown's risk-storming, Gary Klein's pre-mortem, Ted Neward's architecture katas, Eoin Woods's TARA for existing systems, written design reviews and checklists. It ends with a step-by-step format for a 90-minute review and a guide to choosing between techniques.

Section 11.3 turns one-off evaluation into continuous protection. A fitness function is an objective check of an architectural characteristic, ideally automated: a dependency rule, a load test with percentile thresholds, a vulnerability scan, a chaos experiment or a schema compatibility check. Together with small, frequent, safe changes, fitness functions make an evolutionary architecture possible. Three scenarios from Chapter 2 become working checks, with code.

Governing, organising and modernising: Sections 11.4 to 11.6

Section 11.4 asks how an organisation of many autonomous teams keeps its architecture coherent. It distinguishes principles, standards, guardrails and gates; shows how to write principles with a name, statement, rationale and implications; and describes decentralised decision-making through Andrew Harmel-Law's advice process, architecture advisory forums and technology radars, with compliance automated through fitness functions and policy as code.

Section 11.5 takes Conway's law seriously as a design tool. It presents Matthew Skelton and Manuel Pais's Team Topologies: four team types, three interaction modes, team cognitive load, team APIs and fracture planes, and shows how to redesign the food-delivery teams around streams of change. Section 11.6 deals with the systems that already exist: why big-bang rewrites so often fail, the range of migration strategies, and the incremental techniques that work, including the strangler fig, branch by abstraction, parallel runs, change data capture and anti-corruption layers, worked through by strangling an order module out of a monolith.

Leading, justifying and acting responsibly: Sections 11.7 and 11.8

Section 11.7 covers the human side of the role: leading without formal authority, building trust, facilitating decisions by consent or consensus, handling conflict and organisational politics, communicating with executives, mentoring, staff-plus engineering paths and the decision traps that catch experienced people, with a contested event-bus decision navigated step by step.

Section 11.8 closes the chapter with the two arguments an architect most needs to make well. The business case: costs, benefits, risks and options, with net present value and payback calculated in a worked example, and technical debt expressed in money and time. And the ethical case: professional codes of conduct, foreseeable harms such as privacy loss, bias, inaccessibility and manipulative design, and sustainability measured with the Software Carbon Intensity specification.

The one idea to take away

Architecture is not finished when the diagram is drawn. Its decisions have to be tested against what the business needs, protected while the system changes, made by the right people with the right advice, carried by teams whose structure fits them, and justified in terms that sponsors and society can accept. The most effective architects make this work light and continuous rather than heavy and occasional: short risk reviews instead of annual audits, automated guardrails instead of approval queues, advice instead of permission, incremental migration instead of rewrites. The discipline that ties it together is the one this course began with: name the quality you need, make it measurable, decide deliberately, write down why, and keep checking.

Architecture evaluation and the ATAM

BEFORE YOU READ THIS

This section builds on:

- 1.6 Decisions and trade-offs
- 2.3 Quality attribute scenarios
- 2.8 Eliciting and prioritising

Terms it uses from earlier: Quality attribute scenario, Utility tree, Quality attribute, Stakeholder

THE IDEA IN BRIEF

Before a city builds a new bridge, independent engineers check the design: they ask what happens in a flood, under the heaviest lorries, in a storm, and they point out where one choice that saves money makes another danger worse. They do not redesign the bridge or give it a mark out of ten; they tell the owners where the real risks are, so the owners can decide what to do while changes are still cheap. Architecture evaluation does the same for software. Its best-known method, the ATAM, brings the system's owners, builders and users together to test the design against the situations that matter most.

WHAT YOU NEED TO KNOW

- Architecture evaluation checks whether a system's significant decisions will let it meet its business goals and quality requirements, and is most valuable before major or irreversible commitments.
- The ATAM (Kazman, Klein and Clements, SEI, 2000) runs nine steps across preparation, two evaluation meetings and follow-up, with an independent evaluation team, the project's decision makers and a wider group of stakeholders.
- Its outputs are risks, non-risks (with their assumptions), sensitivity points, trade-off points and risk themes linked to the business drivers they threaten; it does not award a pass mark.
- The utility tree built by decision makers in step 5 is compared with the scenarios stakeholders brainstorm and vote on in step 7, and gaps between them reveal blind spots.
- CBAM builds on the ATAM by weighting the utility each architectural strategy adds to prioritised scenarios and dividing by its cost to compare value for cost, while constraints can still override the ratio.

- A full ATAM costs tens of person-days, so the method should be scaled to the stakes, and its findings must become decision records, backlog items and fitness functions with owners.

Why evaluate an architecture

Every architecture is a bet. When a team chooses an event broker, a database per service or a single cloud region, it is predicting that those decisions will deliver the performance, availability, security and cost the business needs. Section 1.3 showed why such bets matter: the cost of changing a structural decision grows with everything built on top of it, and some of the most expensive failures in software were failures of arrangement rather than of individual lines of code. Architecture evaluation is the deliberate act of checking the bet before, or while, the money is spent.

An evaluation asks a narrow, practical question: given what this system must achieve, which of its architectural decisions put those goals at risk? It is not a code review, a beauty contest between styles or an audit of documentation quality. It works on decisions and their consequences for quality attributes, so it can be done long before there is code, on a whiteboard design or a set of decision records, and it can be repeated on a system that has been running for years.

Evaluation also has benefits beyond the findings. Stakeholders who rarely meet (product owners, operations, security, finance, developers) have to agree what matters most, which often exposes conflicting assumptions. The architect has to explain the design clearly enough for outsiders to probe it, which improves the documentation. And the organisation gets a record of known risks that it has chosen to accept, which is very different from risks nobody noticed.

When to evaluate

The best time to evaluate is when enough has been decided for the evaluation to have something to examine, but before those decisions become expensive to reverse. In practice there are several natural moments, and mature organisations use different weights of evaluation for each.

Moments for evaluation

Trigger	What is being checked	Typical weight
Candidate architecture exists, before large-scale build	Whether the main styles and tactics can meet the prioritised quality scenarios	Full or reduced ATAM for large or long-lived systems; a lightweight review otherwise
Before a major commitment	A vendor contract, a cloud migration, a funding gate or a re-platforming decision that would be costly to undo	Formal, often with external evaluators, because the decision is a one-way door
Significant change in business drivers	New markets, a merger, a tenfold growth target, new regulation	Re-run the utility tree and analyse the scenarios that changed
Accumulated change and drift	Whether the system as built still matches the intended architecture and its reasons	Periodic lightweight review plus automated fitness functions
Acquisition or due diligence	Whether an existing system can support the buyer's plans	Time-boxed evaluation of an existing system, often combined with code analysis
After a serious incident	Whether the incident exposed an architectural weakness rather than a one-off fault	Focused analysis of the affected scenarios

Evaluating too early wastes effort on decisions that are not yet real; evaluating too late turns findings into regrets. A useful rule from Fairbanks's risk-driven model (Section 1.8) applies here too: invest in evaluation in proportion to the risk. A two-team internal tool does not need a two-day workshop; a national payments platform does.

> **IF YOU ARE NEW TO THIS**
>
> ### If you are new to this: what an evaluation needs as input
>
> An evaluation is only as good as what it is given. At minimum it needs the business goals, a short list of prioritised quality attribute scenarios (Section 2.3), and a description of the architecture: a context and container diagram (Section 4.3), the key runtime flows and the significant decisions. If these do not exist, producing them is the first, and often most valuable, part of the evaluation.

The ATAM: origins, purpose and participants

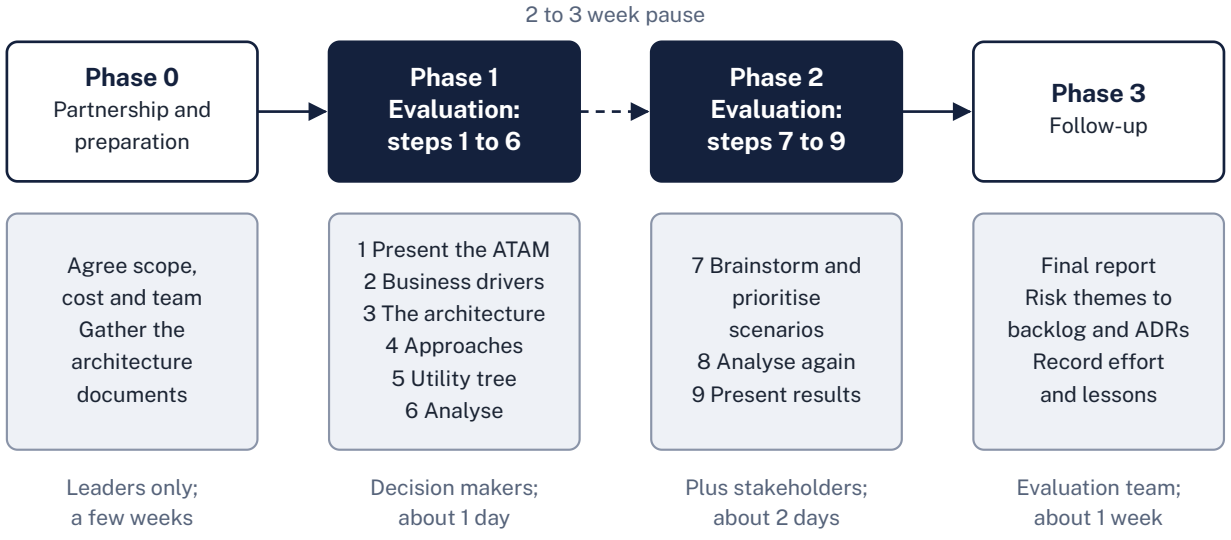
The Architecture Tradeoff Analysis Method (ATAM) is the best-known evaluation method. Rick Kazman, Mark Klein and Paul Clements of the Software Engineering Institute (SEI) at Carnegie Mellon University described it in the technical report *ATAM: Method for Architecture Evaluation*

(CMU/SEI-2000-TR-004, August 2000). It grew out of the earlier Software Architecture Analysis Method (SAAM), which Kazman and colleagues published in 1994 and which focused mainly on modifiability. The ATAM is also described in *Software Architecture in Practice* by Bass, Clements and Kazman, now in its fourth edition (2021).

Its purpose is often misunderstood. The ATAM does not give an architecture a score or a pass mark, and it does not prove that a system will meet its requirements. It identifies where architectural decisions put the business goals at risk and where they force trade-offs between qualities, so that the people responsible can decide what to do. The method's name stresses trade-offs because most interesting risks sit where one quality is bought at the expense of another.

Three groups take part. The **evaluation team**, usually three to five people from outside the project, runs the method: a team leader, an evaluation leader who facilitates, scribes who capture scenarios and proceedings, and questioners with expertise in particular qualities. The **project decision makers** are the architect, the project or product manager and the sponsor or customer who can act on the findings. The **architecture stakeholders** are everyone else with a stake: developers, testers, operations, security, integrators, support and user representatives. Independence matters: evaluators who designed the system, or who report to the architect, find fewer risks.

The phases of the ATAM



The two evaluation meetings are separated by a pause in which the team digests the first findings and the wider stakeholder group is assembled.

The nine steps

1. **1. Present the ATAM**
   The evaluation leader explains the method, the participants' roles and the outputs, so that nobody mistakes the session for an audit or a design review of their personal work.

2. **2. Present the business drivers**
   A senior business representative explains the system's goals, constraints, stakeholders and the qualities that matter most, in business terms. These business drivers are the yardstick for everything that follows.

3. **3. Present the architecture**
   The architect describes the architecture at an appropriate level: context, main structures, runtime behaviour, deployment, and how it addresses the drivers. A typical time limit of about an hour forces focus.

4. **4. Identify architectural approaches**
   The architect names the architectural approaches in use (styles, patterns and tactics such as a broker, replication or caching) and the evaluation team catalogues them, without yet analysing them.

5. **5. Generate the quality attribute utility tree**
   The decision makers build or refine a utility tree and rate each leaf scenario for business importance and technical difficulty.

6. **6. Analyse the architectural approaches**
   For the highest-priority scenarios, the evaluators ask which approaches address each one and probe them with quality-specific questions, recording risks, non-risks, sensitivity points and trade-off points.

7. **7. Brainstorm and prioritise scenarios**
   In phase 2 the wider stakeholder group proposes scenarios of its own and votes on them. The result is compared with the utility tree; important scenarios missing from the tree are a finding in themselves.

8. **8. Analyse the architectural approaches again**
   The highest-ranked new scenarios are analysed in the same way as in step 6, now with the knowledge of the stakeholders in the room.

9

9. Present the results

The evaluation team presents the approaches, scenarios, utility tree, risks, non-risks, sensitivity points, trade-off points and risk themes, and relates each theme to the business drivers it threatens.

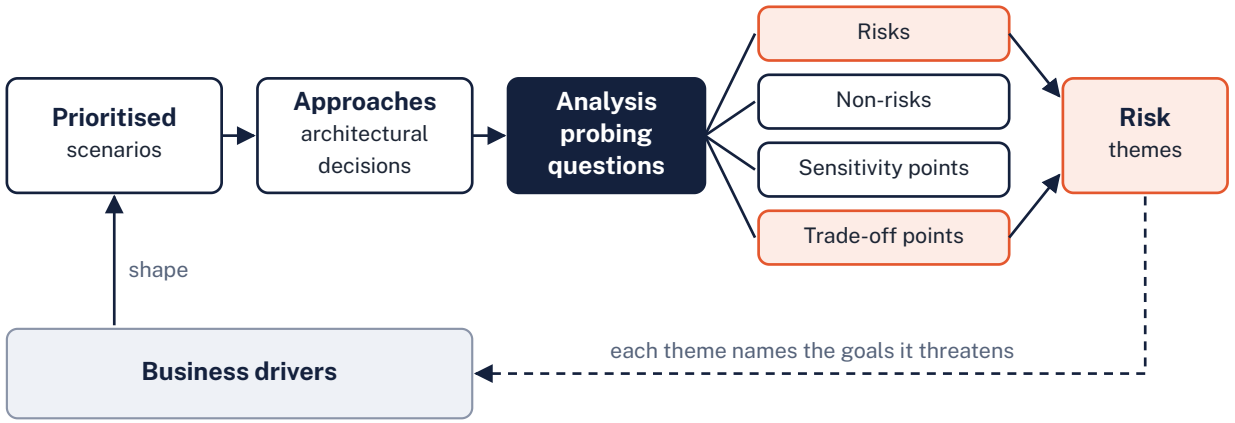
The vocabulary of the analysis

The heart of the method is steps 6 and 8. For each important scenario the evaluators ask the architect to walk through how the architecture responds, and they probe with questions drawn from each quality: how many synchronous calls are on this path, what happens to in-flight requests when that node fails, where is this credential stored, what limits the queue? Each answer is classified using five concepts that have become standard vocabulary well beyond the ATAM itself.

The outputs of the analysis

Output	Meaning	Example from a food-delivery platform
Risk	A decision, or a missing decision, that may prevent an important scenario or business goal from being met	Card authorisation sits inside the synchronous order path, so a slow payment provider slows every order
Non-risk	A decision judged sound for the scenario, recorded with the assumptions it depends on	The transactional outbox and idempotent consumers meet the no-lost-orders scenario, assuming the broker replicates each partition three times
Sensitivity point	A property of components or connections that is critical to one quality response	Order latency is sensitive to the number of synchronous hops and to the provider's p99 response time
Trade-off point	A property that is a sensitivity point for more than one quality, so that improving one worsens another	The courier location update interval: shorter improves map freshness but drains phone batteries and raises mobile data and ingestion cost
Risk theme	A group of risks with a shared underlying cause, linked to the business drivers it threatens	Third-party services sit on synchronous critical paths, threatening the goal of doubling order volume without losing conversions

How the analysis produces its outputs



Analysis relates prioritised scenarios to the decisions that address them; the resulting risks are grouped into themes that point back to the business goals they threaten.

Non-risks deserve more attention than they usually get. Recording that a decision is sound *because* of a specific assumption (three replicas, a contractual response time from a supplier, traffic below a certain level) turns a vague reassurance into something that can be checked later. When the assumption changes, the non-risk becomes a candidate risk. Sensitivity and trade-off points are similarly valuable after the evaluation: they tell future maintainers which parameters to change with care, and they are natural places for the automated checks described in Section 11.3.

The utility tree revisited

Section 2.8 introduced the utility tree as a tool for prioritising quality requirements. It was born in the ATAM, and inside the method it plays two distinct roles. In step 5, the decision makers build the tree top-down: utility, then quality attributes, then refinements, then concrete scenarios rated (business importance, technical difficulty). The (H,H) leaves become the agenda for step 6, because they matter most and are hardest to achieve, so that is where risks are most likely to hide.

In step 7, the wider stakeholder group brainstorms scenarios bottom-up and votes on them, much as in a Quality Attribute Workshop. Comparing the two views is one of the most revealing moments of an evaluation. If the stakeholders' top scenarios already appear in the tree, the decision makers understand their system's purpose. If a scenario that receives many votes is absent, for example an operations scenario about restaurant tablets going offline, the decision makers have a blind spot, and the architecture may have one too. Such scenarios are added to the tree and analysed in step 8.

WORKED EXAMPLE

Worked example: a condensed ATAM of the food-delivery platform

The platform from Chapters 2 and 4 connects customers, 4,000 restaurants and 9,000 couriers. The business drivers are to double order volume within 18 months (from 50 to 100 orders per second at the Friday peak) without losing conversions, and to sign national restaurant chains that demand contractual availability. The architecture: microservices per bounded context (ordering, restaurant, dispatch, notification) each with its own PostgreSQL database; an API gateway; synchronous REST for order placement, including card authorisation with an external payment provider; a Kafka broker with a transactional outbox for order events; courier positions sent every 4 seconds and held in Redis; one cloud region across three availability zones, with nightly backups copied to a second region. The phase 1 utility tree reused the six scenarios of Section 2.8.

Order latency (H,H)	p95 under 1.5 s and p99 under 3 s at 100 orders/s. Approaches: gateway, four synchronous hops, card authorisation in the request. Measured provider latency: p95 900 ms, p99 2.5 s
Analysis	The provider alone consumes most of the p99 budget, leaving 500 ms for four hops and database writes. Little's law uses mean time, not p99 (Section 2.4), but the risk is clear: if a provider slowdown pushes the mean to 2.5 s, 100 orders/s means 250 requests in flight, and the ordering service's pool of 200 workers saturates. **Risk R1**; **sensitivity point S1**: number of synchronous hops and provider p99
Checkout uptime (H,H)	Payment provider times out at peak: orders accepted and queued, none lost, customer told within 10 s. Current design fails the order when authorisation times out. **Risk R2**
Option raised	Authorise asynchronously: accept the order as pending, authorise from a queue. Improves latency and availability, but some accepted orders later fail payment and restaurants must not cook unpaid orders. **Trade-off point T1**: where in the order flow payment is confirmed
No lost orders (part of H,H)	Outbox in the same local transaction as the order, Kafka with three replicas, idempotent consumers. **Non-risk N1**, assuming replication factor 3 and producer acknowledgements from all in-sync replicas
Region failure (H,M)	RTO 15 min, RPO 1 min. Nightly backups give an RPO of up to 24 hours and a restore measured in hours. **Risk R3**
Tracking freshness (M,M)	Map position never more than 5 s old. 9,000 couriers every 4 s is 2,250 writes/s, comfortable for Redis. **Trade-off point T2**: the update interval (freshness against battery, data and cost)

Account takeover (H,M)	Rate limiting by IP address at the gateway does not detect credential stuffing spread across thousands of addresses. **Risk R4**
Phase 2 brainstorm	Restaurant and operations stakeholders voted highly for a scenario missing from the tree: a restaurant tablet offline for 10 minutes at peak must receive every order on reconnection. Push notifications are sent once with no acknowledgement. **Risk R5**, and a blind spot in the tree
Risk themes	**Theme A** (R1, R2, R4): third-party and edge dependencies sit on synchronous critical paths without isolation; threatens doubling volume without losing conversions. **Theme B** (R3, R5): recovery and delivery guarantees lag behind the availability being promised to restaurant chains; threatens the chain contracts
What happened next	Decision records for asynchronous authorisation with a pending state (accepting T1, with restaurants notified only after authorisation), a warm standby region, acknowledged order delivery to tablets, and credential-stuffing detection at the identity provider. N1's assumptions became an automated configuration check

The findings are about decisions, not people or code quality, and each one is traceable to a scenario and a business driver.

Two of the five risks came from scenarios that the architect had not prioritised or not considered, which is why the stakeholder step exists.

Trade-off points are not problems to eliminate: T1 was accepted deliberately and recorded, with its consequences for restaurants and customers.

CBAM: putting money on the table

The ATAM tells you where the risks are; it does not tell you which fixes are worth paying for. The Cost Benefit Analysis Method (CBAM), developed at the SEI by Rick Kazman, Jai Asundi and Mark Klein (first published in 2001), builds on an ATAM's scenarios to make that economic case.

In outline, stakeholders take the prioritised scenarios and assign utility (on a 0 to 100 scale) to several response levels for each: worst case, current, desired and best case. The architects propose architectural strategies and estimate the response each would achieve. The utility gained by each strategy is read off the scenario's utility-response curve and weighted by the scenario's priority; summed across scenarios, that is the strategy's benefit. Dividing benefit by cost gives a value-for-cost ratio (VFC), and strategies are chosen in order of VFC within the available budget, then sanity-checked against intuition and constraints.

A miniature CBAM for the region-failure risk

Strategy	Expected response	Utility gain x weight	Annual cost	Value for cost
Current: nightly backups	RTO about 6 h, RPO up to 24 h	utility 10 (baseline)	already paid	n/a
A: warm standby region, asynchronous database replication	RTO 10 min, RPO under 1 min	(90 - 10) x 0.3 = 24	£180k	24 / 180 = 0.13
B: pilot light (data replicated, compute started on demand)	RTO 40 min, RPO under 1 min	(60 - 10) x 0.3 = 15	£70k	15 / 70 = 0.21

Strategy B gives more benefit per pound, but it misses the 15-minute recovery time that the restaurant contracts require. The numbers inform the decision; they do not make it. CBAM's real value is that it forces explicit, comparable estimates of benefit and cost, and exposes when a constraint, not a ratio, should decide. Its weakness is that utilities and costs are judgements; teams should vary them and see whether the ranking changes, exactly as Section 1.6 recommends for weighted decision matrices. Section 11.8 shows how to turn such estimates into a business case.

Costs and practicalities

A full ATAM is expensive in people's time. As a rough illustration: a four-person evaluation team spending two days each on preparation (8 person-days), a one-day phase 1 with three decision makers (7), a two-day phase 2 with ten stakeholders added (34) and a week of follow-up by two people (10) comes to around 60 person-days, before anyone fixes anything. For a system whose failure would cost millions, or that will run for a decade, that is cheap insurance. For a small product team it is out of proportion.

Several practices keep evaluations worthwhile. Scale the method: Bass, Clements and Kazman describe a lightweight architecture evaluation, a reduced ATAM for internal teams that can be completed in a day or less, and some organisations run only steps 2 to 6 with a handful of people. Prepare properly: most of the cost of a poor evaluation is spent discovering, in the room, that nobody can describe the architecture. Keep evaluators independent but informed. Agree beforehand who owns the findings and how they will be followed up, ideally as decision records and backlog items with owners. And repeat: the findings are a snapshot, and the risks change as the system and the business do.

Practitioners disagree about the method's place today. Its critics see it as a heavyweight artefact of large, up-front projects that fits poorly with teams deploying many times a day. Its defenders answer that the concepts (scenarios, sensitivity and trade-off points, risk themes) are exactly what agile teams lack when they skip evaluation, and that the method scales down. Both have a point: the full ceremony suits high-stakes, long-lived systems and major commitments, while the vocabulary and the habit of scenario-driven questioning belong everywhere. Section 11.2 covers the lighter end.

WATCH OUT

Common mistakes in architecture evaluations

- Treating the evaluation as an exam with a pass or fail, so that the architect defends rather than explores, and risks are hidden.
- Evaluating without business drivers, which produces a list of technical worries with no way to rank them.
- Letting the analysis wander into code style, tool preferences or personal taste instead of scenario-driven questions.
- Recording non-risks without their assumptions, so nobody notices when an assumption stops holding.
- Producing a long risk list and no risk themes, which leaves managers unable to see the few underlying causes.
- Holding the evaluation after the contract is signed or the platform is bought, when the findings can only be regretted.
- Failing to follow up: findings that do not become decision records, backlog items and owners are forgotten within a quarter.

Questions people ask

Can the ATAM be applied to a system that already exists?

Yes. The steps are the same; the architecture is recovered from documentation, code and the people who run the system, and measurements of the live system replace estimates. Evaluating an existing system is common before a major extension, a migration or an acquisition.

Does a clean ATAM mean the system will meet its requirements?
No. The ATAM examines a sample of scenarios at the level of design decisions. It reduces the chance of unpleasant surprises but cannot replace measurement, testing and operational experience. Its findings should be turned into tests and fitness functions where possible.

Who should present the architecture?
The architect or lead engineer who knows it best. The evaluation team's job is to ask questions, not to present, and the architect's job is to explain and explore, not to defend.

CHECK YOUR UNDERSTANDING

▼ **What is the difference between a sensitivity point and a trade-off point?**

A sensitivity point is a property critical to one quality response. A trade-off point is a property that is a sensitivity point for several qualities at once, so that changing it helps one and hurts another.

▼ **Why does the ATAM record non-risks, and what must be recorded with them?**

To show which decisions were examined and judged sound. The assumptions they depend on must be recorded too, so that the non-risk can be re-examined if an assumption changes.

▼ **What does it mean if stakeholders in phase 2 vote highly for a scenario that is missing from the utility tree?**

The decision makers have a blind spot about what matters, and the architecture may not address that scenario. It is added to the tree and analysed in step 8.

▼ **In CBAM, how is a strategy's value for cost calculated?**

Its benefit, the priority-weighted utility it adds across the scenarios it affects, is divided by its cost.

▼ **Try it: Classify these ATAM findings: order throughput depends directly on broker partition count; encryption on one link improves security but adds latency; one payment provider, no fallback, on the order path; a failed zone is handled with a tested failover.**

Partition count is a sensitivity point: one property strongly affects one quality. Encryption is a trade-off point, because the same decision helps security and hurts performance. The single payment provider is a risk, threatening the business driver of order availability. The tested failover is a non-risk, on stated assumptions such as enough capacity in the other zones. Group the risks into themes, then make each finding a decision record or backlog item with an owner.

SOURCES

- SEI: ATAM: Method for Architecture Evaluation (CMU/SEI-2000-TR-004): https://www.sei.cmu.edu/library/atam-method-for-architecture-evaluation/
- SEI: Making Architecture Design Decisions: An Economic Approach (the CBAM): https://www.sei.cmu.edu/library/making-architecture-design-decisions-an-economic-approach

SECTION 11.2 OF 11.8 · ADVANCED · 27 MIN READ · LAST REVIEWED NOVEMBER 2026

Lightweight reviews and risk-storming

BEFORE YOU READ THIS

This section builds on:

- 4.3 The C4 model
- 4.8 Communicating
- 11.1 Evaluation with ATAM

Terms it uses from earlier: Risk (in architecture evaluation), C4 model, System context diagram, Container diagram, Architecture Decision Record (ADR)

THE IDEA IN BRIEF

Before a long hike, a sensible group spreads the map on the table and everyone, quietly and on their own, marks the places that worry them: the river crossing, the steep scree, the stretch with no phone signal. Then they compare. Places marked by many people get a plan; a place marked by only one person gets a question, because that person may know something the others do not. Lightweight architecture reviews work like this. Instead of a distant committee approving a finished plan, the people who know the system look at its diagrams together, early and often, and agree which risks to deal with first.

WHAT YOU NEED TO KNOW

- Review boards that meet at fixed points tend to see designs too late, lack context, invite defensive presentations and add queueing delay; independent review works better when it is frequent, early, collaborative and focused on risks.
- Risk-storming (Simon Brown) has four steps: draw diagrams, identify risks individually and silently, converge them on the diagrams, then review and summarise, scoring probability times impact from 1 to 9.
- A pre-mortem (Gary Klein, Harvard Business Review, 2007) asks people to imagine the effort has already failed and write down why, which surfaces doubts about plans, skills and schedules that diagram reviews miss.
- Architecture katas (Ted Neward) build design and review skills, while TARA (Eoin Woods) reviews existing systems by combining expert judgement with automated analysis of code and runtime data.
- Written design and RFC reviews work best with a stated decision, a comment deadline, required reviewers and comments labelled as questions, suggestions or blocking concerns.

- A good review produces a risk register with owners, dates, accepted risks and non-risks with their assumptions, published within days and tracked in the normal backlog.

Why heavyweight review boards often fail

Many organisations govern architecture through an architecture review board: a panel of senior architects that meets monthly, receives a slide deck or a long document, and approves or rejects designs. The intention is sound: independent, experienced eyes on significant decisions. The results are often poor, for reasons that are structural rather than personal.

Timing is the first problem. A board that meets at fixed points sees a design either too early (nothing is decided yet) or, far more often, too late, when the team has already built a prototype, signed up to a vendor and promised a date. Rejection is then so costly that the board approves with conditions nobody tracks. Context is the second: reviewers who spend an hour on a system they have never seen ask generic questions and miss the specific risks that the team's own developers, operations staff and security engineers could name in minutes. Incentives are the third: when a review is an exam, teams present the design they want approved, not the doubts they have, and the board learns little. Queueing is the fourth: a monthly meeting with a full agenda adds weeks of waiting to every significant decision, which pushes teams to avoid the board by labelling decisions as minor.

None of this means independent review is worthless. Regulated industries (banking, healthcare, aviation) often need a documented sign-off, and some decisions really do need the authority of a senior group. The lesson is that review should be *frequent, early, collaborative and focused on risks*, with formal approval reserved for the few decisions that genuinely require it. Section 11.4 returns to how governance can work without slow gates.

What a good lightweight review gives you

A lightweight review is a short, structured session, typically an hour or two, in which the people who know a design best and a few people who see it fresh look for risks together. It aims at four things: discovering risks while they are cheap to address, building a shared understanding of the design across the team, giving an independent challenge to the architect's assumptions, and leaving a record of the risks accepted, mitigated or rejected. Its output is a prioritised risk list with owners, not a verdict.

Several techniques fit this description. They differ in what they examine (a diagram, a plan, a document or a running system) and in what they are best at, and they combine well.

Lightweight review techniques compared

Technique	Examines	Best for	Typical time
Risk-storming (Simon Brown)	Architecture diagrams	Finding technical risks quickly with the whole team	45 to 90 minutes
Pre-mortem (Gary Klein)	A plan or decision	Surfacing doubts about the whole endeavour, including people and schedule	30 to 60 minutes
Architecture kata (Ted Neward)	A short, invented or real brief	Building design skill and exploring alternatives	1 to 3 hours
TARA (Eoin Woods)	An existing system: people, documents, code and runtime data	Assessing a running system for drift and risk	A few days of expert effort
Design or RFC review	A written proposal	Asynchronous, inclusive comment before building	A comment window of days, plus a short meeting
Checklist review	Any design	Making sure the usual blind spots were considered	Part of any of the above

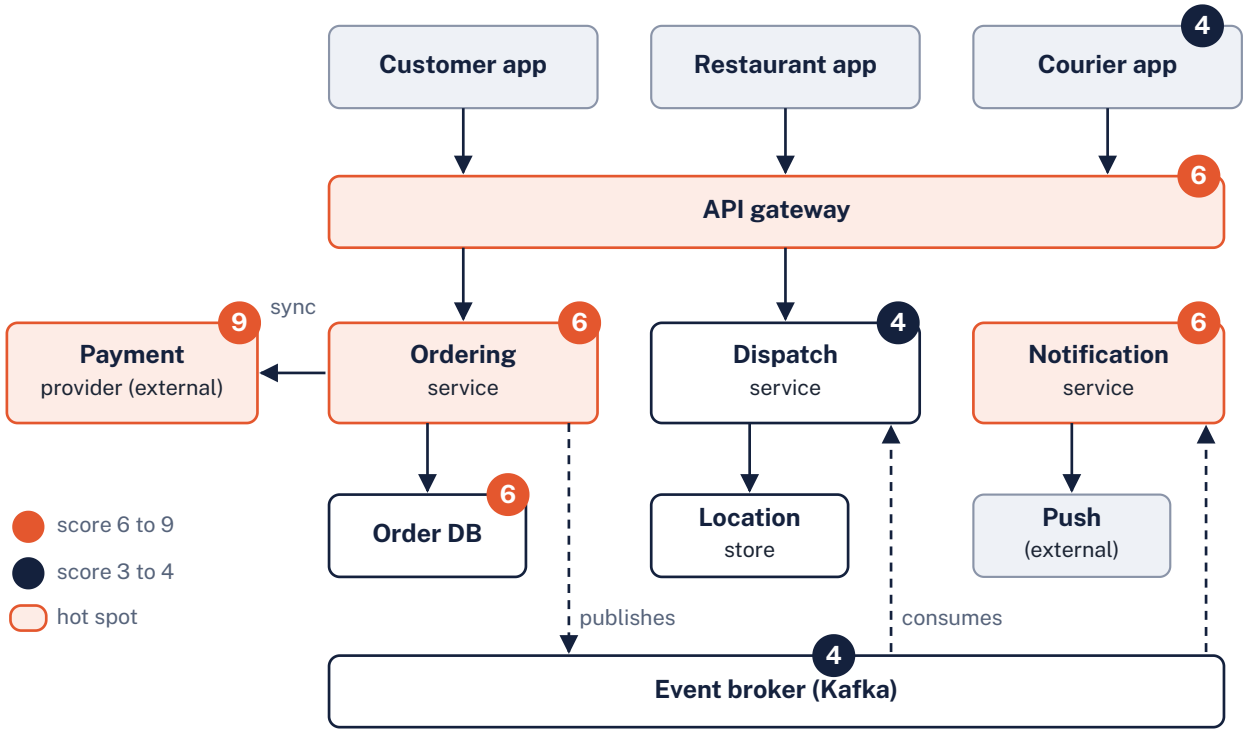
Risk-storming

Risk-storming is a visual, collaborative technique created by Simon Brown, the author of the C4 model (Section 4.3). It rests on a simple observation: different people see different risks, and the best way to collect them is to put everyone in front of the same diagrams and ask them to look independently before discussing. It has four steps.

1. Draw the architecture. Put up diagrams of what you plan to build or change, ideally at more than one level: a system context diagram and a container diagram are usually enough, with a component diagram for the riskiest container. **2. Identify risks individually.** Each participant silently writes one risk per sticky note, with a short description and a priority, for about ten minutes. Silence matters: it prevents the first or most senior speaker from anchoring everyone else, which is the main way group discussions lose information. **3. Converge the risks on the diagrams.** Everyone places their notes on the part of the diagram the risk concerns. Clusters appear immediately, and the diagram becomes a heat map. **4. Review and summarise.** The group discusses the clusters, then pays special attention to two kinds of note: risks raised by only one person (often someone with knowledge others lack, such as the support engineer) and risks given very different priorities by different people (a sign of a misunderstanding). Agreed risks go into a risk register with owners; items agreed not to be risks are recorded too.

Priority is usually scored as probability times impact, each rated low (1), medium (2) or high (3), giving scores from 1 to 9. Brown's guidance bands them as high (6 or 9), medium (3 or 4) and low (1 or 2), conventionally shown with pink or red, yellow or amber, and green notes. A prompt list helps people think beyond their own speciality: scalability, availability, performance, security, data consistency, third-party dependencies, cost, technology maturity, team skills, and delivery and operation.

A risk-storming heat map on a container diagram



After convergence, the scores cluster on the order and payment path, the shared gateway and restaurant notifications; those containers are the hot spots to discuss first.

WORKED EXAMPLE

Worked example: risk-storming the food-delivery container diagram

Six people (the architect, two developers, an operations engineer, a security engineer and a support engineer who handles restaurant complaints) spent 75 minutes on the container diagram above, with the system context diagram alongside. Each wrote risks silently for ten minutes; 31 notes went up, which merged into 8 distinct risks.

Payment provider on the synchronous order path (9)	Raised by four people. Provider slowdowns at peak stall order placement. Action: decision record on asynchronous authorisation with a pending order state; owner: architect; due in two weeks
Restaurant notifications not acknowledged (6)	Raised only by the support engineer, who sees missed-order complaints every Friday. Nobody else knew. Action: acknowledged delivery with re-send and an alert when a tablet stops acknowledging; owner: notification team
Shared gateway configuration (6)	One faulty route change can take all three apps down at once. Action: per-app gateway configuration and canary rollout of gateway changes
Order database failover never tested (6)	Replica exists but promotion has never been rehearsed. Action: quarterly failover drill, measured against the recovery scenario
Kafka skills (scored 9 by one person, 2 by others)	The one engineer who had run Kafka before explained partition rebalancing during deploys, consumer lag and retention mistakes. After discussion the group agreed 4: real but manageable. Action: managed Kafka service and a runbook
Courier location traffic through the gateway (4)	About 2,250 updates per second share the gateway with order traffic. Action: measure in the next load test; separate ingestion endpoint if order latency suffers
Dispatch consumer lag at peak (4)	Action: alert on consumer lag; add a fitness function to the load test
Location store loses data on restart (2)	Agreed not to be a risk: positions refresh every 4 seconds. Recorded as such, with the assumption

The highest-value finding (restaurant notifications) came from the person least likely to speak first in an open discussion, which is the point of silent, individual identification.

The disagreement about Kafka was resolved by knowledge, not by averaging the scores.

Every agreed risk left the room with an owner and an action; the register was published the same day.

Pre-mortems

Risk-storming looks at a structure. A pre-mortem looks at a plan. Gary Klein described it in the *Harvard Business Review* in September 2007. A post-mortem explains why a patient died, which helps everyone except the patient; a pre-mortem is held before the project starts, when the answers can still change its course.

The procedure is short. The leader briefs the group on the plan or decision, then announces that it is now some months in the future and the effort has failed, badly. Each person spends a few minutes writing down every reason they can think of for the failure, independently. The leader then goes round the room collecting one reason at a time until the lists are exhausted, and the owner of the plan reviews them for ways to strengthen it. Klein cites research by Deborah Mitchell, Jay Russo and Nancy Pennington (1989) finding that imagining an event has already happened (prospective hindsight) increases people's ability to identify reasons for future outcomes by about 30%. The framing also changes the social dynamics: the question is no longer whether the plan might fail, which sounds disloyal, but why it did, which invites creativity.

For architecture, pre-mortems are particularly useful before large commitments: adopting an event-driven style, migrating to a new cloud, replacing a legacy system (Section 11.6) or introducing a platform. Typical outputs that a diagram-based review would miss include the team lacking skills in the new technology, the migration stalling halfway with two systems to run, a key supplier changing its pricing, and the business losing patience before benefits appear.

IF YOU ARE NEW TO THIS

Step by step: imagining the failure first

The idea that needs the most care here is why a pre-mortem works: imagining that a plan has already failed brings out doubts that a plain 'what could go wrong?' discussion often leaves unsaid. Follow one plan from the risk-storming session above.

- Suppose the team decides to take card authorisation off the synchronous order path, accepting orders in a pending state. Before the work starts, the team and people from neighbouring teams meet for half an hour.
- The leader explains the plan, then announces that it is now six months later and the change has failed badly. Each person writes down, alone and in silence, every reason they can think of.
- The framing does the work. Asked whether the plan might fail, someone with doubts sounds disloyal for saying so. Asked why it did fail, every doubt becomes a useful answer.
- The leader collects one reason at a time until the lists run out: nobody in the ordering team had built this kind of design before; the change stalled half way, leaving old and new payment paths both running; the payment provider changed its pricing; the business lost patience before the benefits appeared.
- Few of these would appear as sticky notes on the container diagram: they concern schedule, suppliers and patience more than boxes and arrows. The plan's owner then strengthens it, for example by arranging training first and setting a date for switching off the old path.
- **The lesson:** risk-storming examines a structure; a pre-mortem examines a plan. Imagining the failure has already happened makes reasons easier to find and safer to say, so hold one before a large commitment, while the answers can still change the plan.

Architecture katas

Architectural katas were created by Ted Neward, inspired by Dave Thomas's code katas, as practice exercises for architects. A kata is a short brief for a fictional system (a few paragraphs of requirements, users and constraints, deliberately incomplete). Small teams of three to five people work on it for a fixed time, questioning a moderator who plays the customer to uncover missing requirements, then sketch an architecture and present it to the other teams, who challenge it hard but fairly before the room votes. Neal Ford maintains a public collection of katas, and O'Reilly has run architectural kata competitions.

Katas are not reviews of a real design, but they contribute to review culture in two ways. They build the skill of explaining and defending a design under questioning, and of asking good questions about someone else's, which is exactly what lightweight reviews need. And an internal kata on a real upcoming problem, with two or three teams designing independently, is a cheap way to explore the option space before committing: when three teams produce three different designs, the differences show where the real trade-offs lie.

TARA and checklist-based reviews

Reviews of systems that already exist face a different problem: the architecture as built may differ from the architecture as described, and the evidence is spread across people, documents, code and production data. Eoin Woods, co-author of *Software Systems Architecture*, described the Tiny Architectural Review Approach (TARA) in 2011 as a deliberately lightweight, industrial alternative to scenario-heavy methods. A small number of reviewers elicit the system's key requirements and context, establish its actual structure, assess its implementation using automated analysis (module dependencies, size, code metrics, test coverage) alongside runtime information such as logs, use expert judgement to link what they find to the requirements, and present findings and recommendations in a predefined form. Its strength is speed and evidence; its weakness, which Woods acknowledged, is dependence on the reviewers' expertise.

Checklists complement every technique. A checklist does not find risks by itself, but it stops a review from forgetting a whole category (backup restore, secrets, cost at scale) because nobody in the room happened to think of it. Keep checklists short, specific to your organisation's history of incidents, and phrased as questions that demand evidence rather than yes or no answers.

Well-Architected reviews

The large cloud providers publish frameworks that can serve as ready-made review checklists. AWS and Google Cloud each describe six pillars and Azure five, covering operations, security, reliability, performance and cost, with sustainability added by the first two. A review walks one workload through structured questions. The AWS Well-Architected Tool carries no extra charge, records your answers and flags high-risk issues, and custom lenses can add your own rules; Azure offers an online Well-Architected Review. For the food-delivery ordering workload, a half-day session gives broad coverage and a list of risks to fix or accept. The limits are real: the questions express one provider's view of good practice, and they do not weigh your own quality scenarios against each other as Section 11.1 does. Check the current version, as pillars and questions change.

Design documents and RFC reviews

Section 4.8 described design documents and RFCs as a way to propose a change in writing before building it. Reviewing them well is a skill of its own. A written review has real advantages over a meeting: reviewers comment when they have time to think, remote and quieter colleagues contribute on equal terms, and the comments become a record. Its typical failures are documents nobody reads, comment threads that never close and reviews that become an unofficial gate.

Good practice is to state the decision sought and the deadline for comments at the top, name the reviewers whose input is required (security, operations, the owners of affected services) and invite anyone else; ask reviewers to label each comment as a question, a suggestion or a blocking concern, so the author knows which ones must be resolved; review against the quality scenarios and the checklist below, not against personal taste; close with a short meeting only for unresolved concerns; and record the significant decisions as ADRs. In organisations using the advice process (Section 11.4), the RFC is the vehicle for seeking advice, and the decision stays with its author.

Running a 90-minute architecture review, step by step

1. **Before: prepare (a few days ahead)**
   The architect circulates a one-page summary, the context and container diagrams, the top three to five quality scenarios with numbers, and the open questions. Invite five to eight people: the team, operations, security, someone from each neighbouring team and one experienced outsider. Choose a facilitator who is not the architect.

2. **0 to 10 minutes: goals and rules**
   The facilitator states the purpose (find risks, not approve or reject), the time box and the output. A business representative or product owner restates the business drivers in two minutes.

3. **10 to 25 minutes: walk the architecture through scenarios**
   The architect walks two or three of the most important scenarios through the diagrams, for example one order at peak and one payment-provider outage. Questions of clarification only.

4. **25 to 35 minutes: silent risk identification**
   Everyone writes risks on notes, one per note, with a probability and impact. No discussion.

5. **35 to 50 minutes: converge and cluster**
   Notes go on the diagrams. The facilitator merges duplicates aloud and highlights clusters, lone notes and priority disagreements.

6 **50 to 75 minutes: discuss the top risks**
Take the highest-scoring clusters first. For each, agree what the risk really is, its score and whether it is a risk, a non-risk with an assumption, or a question needing a spike.

7 **75 to 85 minutes: actions and owners**
Each agreed high or medium risk gets a mitigation, an owner and a date. Decisions to be recorded as ADRs are listed.

8 **85 to 90 minutes: close**
Read back the risks, the accepted risks and the non-risks with their assumptions. Agree when the design will be looked at again.

9 **After: publish within two days**
Publish the risk register, photographs of the annotated diagrams and the action list where the team keeps its architecture documentation, and track the actions in the normal backlog.

DECISION GUIDE

Decision guide: which kind of review?

Start at the top; techniques can be combined.

Q1 **Is an expensive, hard-to-reverse commitment imminent, such as a multi-year contract, a platform purchase or a funding gate, with many stakeholders?**

- **Yes** Use an ATAM or a reduced ATAM with independent evaluators (Section 11.1), and add a pre-mortem on the plan.
- **No** Continue.

Q2 **Is there a written proposal for a significant change?**

- **Yes** Run a written design or RFC review with a comment deadline, then a short meeting for blocking concerns.
- **No** Continue.

Q3 **Do diagrams of the proposed or existing architecture exist?**

- **Yes** Run risk-storming on them, using the 90-minute format.
- **No** Draw a context and a container diagram first; drawing them often exposes risks on its own.

Q4 **Is the main worry the success of the whole effort (skills, schedule, migration, adoption) rather than the structure?**

- **Yes** Add a pre-mortem.
- **No** Continue.

Q5 **Is the system already running, with suspected drift or growing debt?**

- **Yes** Use a TARA-style review combining interviews with automated code and runtime analysis, and turn findings into fitness functions.
- **No** Rely on fitness functions between reviews, and review again at the next significant decision.

Checklist: questions for any lightweight architecture review

Ask for evidence, not reassurance. Adapt the list to your organisation's own incident history.

- [ ] **Which three to five quality scenarios matter most, and what are their numbers?**
  Without measurable drivers the review cannot rank risks (Section 2.3).
- [ ] **What is on the synchronous path of the most important journey?**
  Count hops, list timeouts and retries, and identify every third party on the path.
- [ ] **What happens when each dependency is slow or down?**
  Slow is usually worse than down; ask about both, and about how users are told.
- [ ] **Who owns each important piece of data, and has a restore been tested?**
  Single writer, consistency expectations, retention, and evidence of a recent successful restore (Chapter 7).
- [ ] **Where are the trust boundaries, identities and secrets?**
  Look for a current threat model and for secrets in code or configuration (Chapter 8).
- [ ] **What breaks first at twice the expected peak?**
  Ask for the capacity model or load-test results, not intuition.
- [ ] **How is the system observed, deployed and rolled back?**
  Dashboards, alerts tied to SLOs, deployment strategy and the time a rollback takes.
- [ ] **What is likely to change in the next year, and what would that change touch?**
  Check that likely changes stay inside one module or service.
- [ ] **What will it cost to run at expected volume?**
  A cost per order, per user or per transaction, with the main drivers.
- [ ] **Are the significant decisions recorded with alternatives and consequences?**
  ADRs or a design document, including the trade-offs accepted.
- [ ] **Does the team structure match the design?**
  Each part has an owning team with the skills to run it (Section 3.7 and Section 11.5).

WATCH OUT

How lightweight reviews go wrong

- Open discussion before silent identification, so the first speaker anchors the group.
- Averaging disagreements away instead of asking why one person scored a risk so differently.
- Reviewing the diagram's notation instead of the architecture's risks.
- No owners or dates, so the risk register becomes a list of good intentions.
- The architect as facilitator, which makes it hard to receive criticism and run the session at the same time.
- Running reviews only when something is already in trouble; regular, routine reviews are cheaper and less defensive.

CHECK YOUR UNDERSTANDING

▼ Why does risk-storming ask people to identify risks silently before any discussion?

To stop the first or most senior speaker anchoring the group, so that each person's independent knowledge reaches the diagram.

▼ In risk-storming, which two kinds of note deserve special attention during the review step?

Risks raised by only one person, who may know something others do not, and risks given very different priorities, which signal a misunderstanding.

▼ What does a pre-mortem examine that a risk-storming session on a container diagram usually does not?

The plan and its context as a whole: skills, schedule, migration, suppliers and organisational patience, rather than the technical structure.

▼ Name two structural reasons why monthly architecture review boards often fail.

Any two of: they see designs too late, reviewers lack context, teams present defensively, and queueing adds weeks of delay that encourages avoidance.

▼ **Try it: A risk-storm rates (probability, impact): payment provider outage (3, 3); broker misconfiguration (2, 3); search index stale (3, 1); data centre fire (1, 3). Score and rank them. Why were notes written silently first?**

Payment provider 9, high. Broker 6, high. Search index 3 and fire 3, both medium, though the fire deserves a line stating its assumptions, because it has high impact. Tackle the top two first, with owners and dates in the normal backlog. Silent writing stops the first or most senior voice steering the group. When four people independently raise the same risk, that is useful evidence, and disagreements show differing assumptions.

SOURCES

- Simon Brown: Risk-storming: https://riskstorming.com/
- Gary Klein: Performing a Project Premortem, Harvard Business Review (2007): https://hbr.org/2007/09/performing-a-project-premortem
- Neal Ford: Architectural Katas: https://nealford.com/katas/about.html
- Experiences with ALMA: Architecture-Level Modifiability Analysis: https://research.vu.nl/en/publications/experiences-with-alma-architecture-level-modifiability-analysis/
- The pillars of the AWS Well-Architected Framework: https://docs.aws.amazon.com/wellarchitected/latest/framework/the-pillars-of-the-framework.html
- What is AWS Well-Architected Tool?: https://docs.aws.amazon.com/wellarchitected/latest/userguide/intro.html
- Workloads (AWS Well-Architected Tool): https://docs.aws.amazon.com/wellarchitected/latest/userguide/workloads.html
- AWS Well-Architected Tool pricing: https://aws.amazon.com/well-architected-tool/pricing/
- Azure Well-Architected Framework pillars (Microsoft): https://learn.microsoft.com/en-us/azure/well-architected/pillars
- Google Cloud Well-Architected Framework: https://docs.cloud.google.com/architecture/framework

SECTION 11.3 OF 11.8 · ADVANCED · 27 MIN READ · LAST REVIEWED NOVEMBER 2026

Fitness functions and evolutionary architecture

BEFORE YOU READ THIS

This section builds on:

- 2.3 Quality attribute scenarios
- 3.8 Simplicity and measurement
- 10.5 CI/CD and deployment

Terms it uses from earlier: Quality attribute scenario, Response measure, Architecture Decision Record (ADR), Utility tree, Last responsible moment

THE IDEA IN BRIEF

A car's dashboard has warning lights for the things that must never go wrong unnoticed: oil pressure, engine temperature, brakes. You do not wait for the annual service to discover the engine overheated in March. A fitness function is a warning light for a software architecture. The team decides which qualities really matter, such as how fast an order is confirmed or which parts of the code may talk to each other, writes an automatic check for each, and runs those checks every time the software changes. When a check fails, the change is stopped or someone is alerted, while the problem is still small.

WHAT YOU NEED TO KNOW

- An architectural fitness function is any mechanism that gives an objective assessment of an architectural characteristic, ideally automated and traceable to a quality attribute scenario whose response measure becomes its threshold.
- Fitness functions are classified as atomic or holistic, triggered, continual or temporal, static or dynamic, automated or manual, and intentional or emergent; a healthy set mixes them.
- Typical examples are dependency rules for structure, load tests and latency budgets for performance, vulnerability scans for security, chaos experiments for resilience and schema compatibility checks for data.
- Evolutionary architecture (Ford, Parsons, Kua and Sadalage) supports guided, incremental change across multiple dimensions; fitness functions provide the guidance, and small deployable units and pipelines provide the increments.
- Each fitness function needs an owner, a recorded rationale and a threshold, and a failure forces a decision: fix the code or change the rule and record why.

- Start with a few reliable checks for the highest-priority characteristics, use baselines to ratchet existing systems, and keep slow holistic checks out of every build.

From one-off evaluation to continuous checking

An evaluation such as the ATAM (Section 11.1) is a snapshot. The day after it ends, developers start changing the system, and every change can quietly undo a decision the evaluation relied on: a convenient import across a module boundary, an extra synchronous call on the order path, a library with a known vulnerability. Section 1.7 described the result: erosion, when the architecture is violated, and drift, when it is ignored. Periodic reviews catch some of this, months late. The idea in this section is to check the most important architectural decisions automatically and continuously, in the same way that unit tests check behaviour.

The term comes from Neal Ford, Rebecca Parsons and Patrick Kua's *Building Evolutionary Architectures* (O'Reilly, 2017), whose second edition, with Pramod Sadalage, appeared in November 2022. They borrowed it from evolutionary computing, where a fitness function scores how close a candidate solution is to the goal so that the algorithm can keep the better variants. In architecture, the goal is the set of characteristics the system must keep while everything else changes.

What a fitness function is

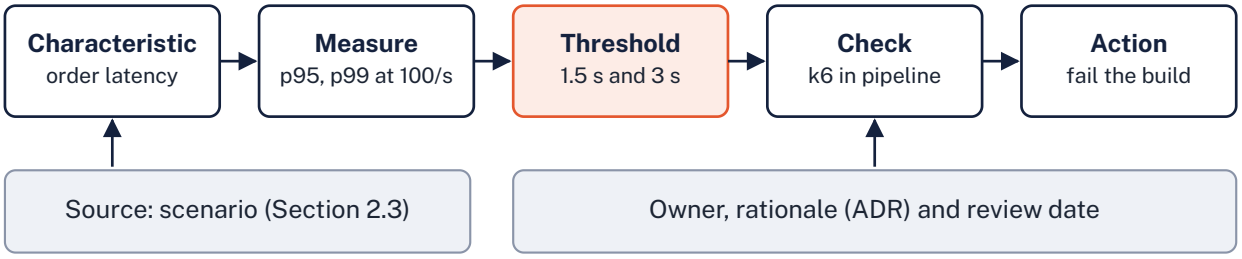
The second edition defines an architectural fitness function as any mechanism that provides an objective integrity assessment of some architectural characteristic or characteristics. Three words in that definition do the work. *Any mechanism*: a unit test, a static-analysis rule, a load test, a monitor, a chaos experiment or even a scheduled manual review all qualify. *Objective*: the result must not depend on who runs it; "the code feels well structured" is not a fitness function, "no package in the domain layer imports the web framework" is. *Architectural characteristic*: the check protects a quality the architecture exists to deliver (performance, availability, modifiability, security, data compatibility), not a business rule, which is the job of ordinary tests.

In practice a useful fitness function has a few more properties. It is traceable to a requirement, ideally a quality attribute scenario whose response measure becomes its threshold (Section 2.3). It has an owner who decides what happens when it fails. It fails loudly, by breaking a build or paging someone, rather than populating a dashboard nobody reads. And its threshold and the reason for it are recorded, usually in an ADR, so that a later team can tell the difference between a rule that protects something important and an arbitrary number.

Section 3.8 drew the important distinction with metrics: a metric describes, a fitness function decides. A coupling metric on a dashboard tells you things are getting worse; a fitness function encodes a decision the team has already made, so breaking it forces a conscious choice: fix the

code, or change the rule and record why.

Anatomy of a fitness function



A fitness function turns a scenario's response measure into a threshold, an automated check and an action, with an owner and a recorded reason.

Categories of fitness function

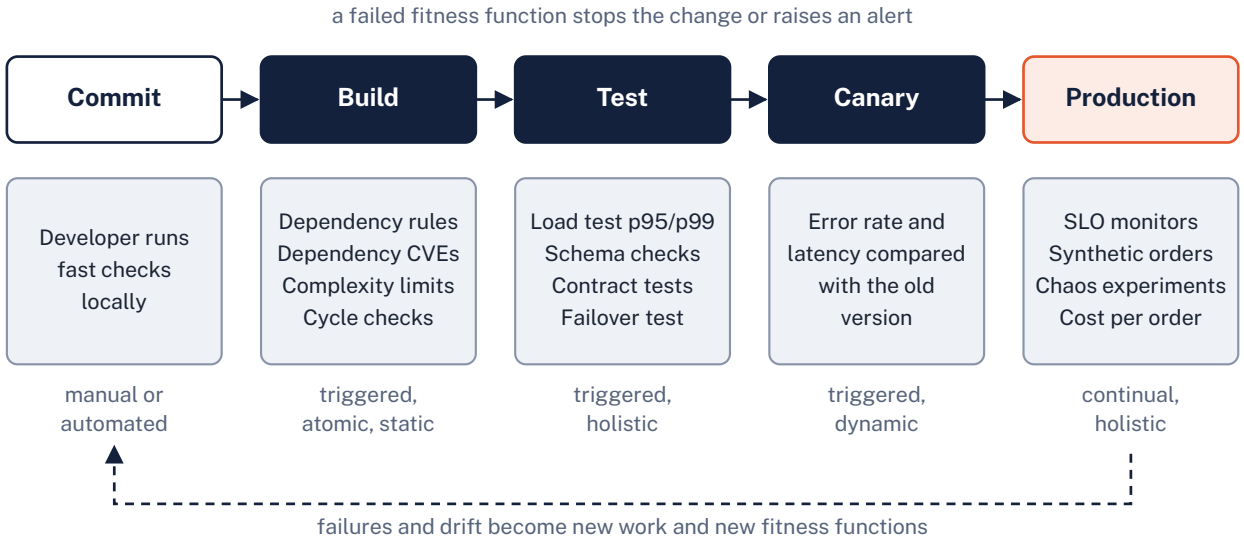
Ford and his co-authors classify fitness functions along several independent dimensions. The classification is useful less as taxonomy than as a design prompt: asking "should this be atomic or holistic, triggered or continual?" often reveals a better check than the first one that came to mind.

Dimensions of fitness functions

Dimension	Options	What it means	Example
Scope	Atomic or holistic	Atomic checks one characteristic in one context; holistic checks several characteristics interacting in a shared context	Atomic: no cycles between packages. Holistic: order latency measured with encryption, audit logging and caching all switched on
When it runs	Triggered, continual or temporal	Triggered runs on an event such as a commit, build or deployment; continual runs all the time against the live system; temporal fires after time passes, as a reminder to revisit something	Triggered: dependency scan on every build. Continual: SLO burn-rate alert in production. Temporal: fail the build when a pinned framework version is more than 12 months old
Kind of result	Static or dynamic	Static compares against a fixed threshold; dynamic adjusts the threshold to context, such as scale	Static: p95 under 1.5 s. Dynamic: p95 may rise to 2 s above 150 orders/s, as agreed for extreme peaks
How it is run	Automated or manual	Automated is preferred; some characteristics, such as legal or usability requirements, still need people, ideally on a schedule	Manual: quarterly accessibility review against WCAG 2.2 AA
Planning	Intentional or emergent	Intentional checks are designed up front from known requirements; emergent ones are added as the team learns what matters	Emergent: a check on consumer lag added after an incident

Holistic checks deserve attention because qualities interact: security controls add latency, caching complicates data freshness, and an atomic test of each in isolation can pass while the combination fails. Continual checks matter because some characteristics only exist in production: real user latency, real failure modes, real cost. A healthy portfolio combines fast, atomic, triggered checks in the build with a smaller number of slower holistic checks before release and continual checks in production.

Where fitness functions run



Cheap structural checks run on every build, holistic checks before release, and continual checks in production, with failures feeding back as new work.

IF YOU ARE NEW TO THIS

Step by step: when every small check passes

The step most people trip over here is that checks which each pass on their own can miss a failure that appears only when everything runs together, which is why a team needs both atomic and holistic fitness functions. Follow the order-latency scenario through both.

- The food-delivery platform's scenario says that at the Friday peak of 100 orders a second, 95% of order confirmations must return within 1.5 seconds.
- Suppose that over a few months the team switches on encryption and audit logging on the order path. Each change has its own atomic check, which confirms that it works and, measured alone, adds only a little time. Every check passes.
- But the time adds up. With both switched on at peak load, their costs combine with everything else on the path, and, say, 6% of confirmations now take longer than 1.5 seconds, more than the 5% the scenario allows. No single check was wrong; none looked at the whole.
- A holistic check catches this: a load test sending 100 orders a second to a production-like environment with encryption, audit logging and caching all switched on, failing if the 95th percentile exceeds 1.5 seconds.
- Holistic checks have costs. A ten-minute load test is too slow for every build, so it runs nightly and before each release. And a failure says that the whole is too slow, not which part caused it; the worked example later in this section shows one way round that.
- **The lesson:** fast atomic checks on every build catch most mistakes cheaply, but cannot see how qualities interact. Add fewer, slower holistic checks before release, and continual checks in production, where real users' experience can be measured.

Fitness functions across the qualities

Most teams start with structure, because structural checks are cheap, fast and deterministic, then add performance, security, resilience and data checks as the pipeline matures. The table gives typical examples; tools change, the patterns do not.

Examples by quality

Quality	Fitness function	Kind	Typical tooling
Structure	Layer and dependency rules, no cycles, modules use each other only through public APIs (Section 3.4)	Atomic, triggered, static	ArchUnit, import-linter, dependency-cruiser, Spring Modulith verification
Performance	End-to-end load test with percentile thresholds; per-service latency budgets in each service's own pipeline	Holistic, triggered	k6, Gatling, JMeter, Locust; thresholds that fail the job
Security	Fail the build on dependencies with known high or critical vulnerabilities; no secrets in code; container images only from the approved registry	Atomic, triggered	pip-audit, npm audit, OWASP Dependency-Check, Trivy, secret scanners; policy as code (Section 11.4)
Resilience	Chaos experiments that kill instances or slow dependencies and check that the steady state holds (Section 9.8)	Holistic, triggered or continual	Toxiproxy, Chaos Toolkit, cloud fault-injection services
Data	Schema compatibility checks on events and APIs; migrations rehearsed on production-sized data (Section 7.7)	Atomic, triggered	Schema registry compatibility checks, buf breaking, consumer-driven contract tests
Operability and cost	Every service exposes health checks and standard metrics; cost per order stays below a target	Atomic or continual	Pipeline checks on service templates; cost dashboards with alerts

Evolutionary architecture and incremental change

Fitness functions are one half of what Ford, Parsons, Kua and Sadalage call evolutionary architecture: an architecture that supports guided, incremental change across multiple dimensions. Each part of the phrase is a design requirement.

Incremental change has a development side and an operational side. Developers must be able to make small changes safely, which depends on modularity and appropriate coupling (Chapter 3) and on keeping each independently deployable part, the architecture quantum of Section 5.1, small enough to change without coordinating with others. Operations must be able to release small changes frequently and safely, which depends on a deployment pipeline, progressive rollout and fast rollback (Section 10.5). *Guided* is where fitness functions come in:

they define what "fit" means and stop increments from degrading it. *Multiple dimensions* is a reminder that architecture is not only code structure: data, security, operations, team structure and cost all evolve and all need protecting.

The approach changes how architects think about the future. Instead of predicting every requirement and building flexibility for it in advance, which Section 3.8 showed is expensive and often wrong, the team keeps the system easy to change and protects only the characteristics that matter, deferring other decisions to the last responsible moment (Section 1.6). Critics point out real limits. Not every quality can be checked automatically or cheaply; badly chosen thresholds create noise that teams learn to ignore; and some decisions, such as a database or a cloud provider, remain one-way doors however good the pipeline is. Evolutionary architecture is a discipline for keeping the reversible parts reversible and the important parts protected, not a promise that everything can be changed later.

WORKED EXAMPLE

Worked example: three Chapter 2 scenarios as fitness functions

Section 2.8 built a utility tree for the food-delivery platform. The team turns three of its leaves into fitness functions, one triggered and holistic, one a chaos experiment, one structural, after the ATAM in Section 11.1 led to asynchronous payment authorisation.

Scenario 1: order latency (H,H)	At the Friday peak of 100 orders/s, 95% of order confirmations return within 1.5 s and 99% within 3 s
Fitness function 1	Holistic, triggered: a k6 load test against a production-like environment, nightly and before every release, failing if p95 or p99 exceeds the threshold or more than 0.1% of requests fail. Backed by continual production monitoring of the same percentiles from real traffic
Latency budgets	The 1.5 s is split into budgets: mobile network and TLS 400 ms, gateway 50 ms, ordering 250 ms, pricing 100 ms, database writes 150 ms, leaving 550 ms of headroom now that card authorisation is off the path. Each service's pipeline runs an atomic component load test against its own budget, so a regression is caught in the service that caused it
Scenario 2: checkout uptime (H,H)	When the payment provider times out during the peak, orders are accepted and queued, no confirmed order is lost, and customers are told within 10 s
Fitness function 2	Holistic chaos experiment, weekly in staging and quarterly as a production game day with a small blast radius: the provider is made to hang; orders must be accepted as pending within 10 s and all must settle after recovery. Owner: ordering team
Scenario 3: new POS vendor (M,L)	A new restaurant point-of-sale vendor is integrated in 5 person-days by adding one adapter
Fitness function 3	Atomic, triggered, static: an import-linter contract forbids the restaurant domain and API modules and the ordering package from importing a vendor SDK, and a second contract stops adapters importing each other, so the change stays local. Supplemented by a manual check: the effort of each real vendor integration is recorded and compared with the 5-day target
Recorded with each	Source scenario, threshold, owner, the ADR that explains it, and a review date

Each response measure became a threshold without reinterpretation, which is the payoff of writing scenarios well in the first place.

The modifiability scenario cannot be fully automated: the structural rule protects the conditions for a 5-day change, and the manual measurement checks the outcome.

Latency budgets turn one holistic number into atomic checks that tell the right team about a regression.

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

export const options = {
  scenarios: {
    friday_peak: { executor: 'constant-arrival-rate', rate: 100, timeUnit: '1s',
                   duration: '10m', preAllocatedVUs: 300 },   // 100 orders per second
  },
  thresholds: {   // the scenario's response measure, as pass/fail rules
    'http_req_duration{name:place_order}': ['p(95)<1500', 'p(99)<3000'],
    'http_req_failed{name:place_order}': ['rate<0.001'],
  },
};

const order = { restaurantId: 'r-42', items: [{ sku: 'pad-thai', qty: 1 }] };

export default function () {
  const res = http.post(`${__ENV.BASE_URL}/orders`, JSON.stringify(order), {
    headers: { 'Content-Type': 'application/json' }, tags: { name: 'place_order' } });
  check(res, { 'order accepted': (r) => r.status === 201 || r.status === 202 });
}
```

Fitness function 1: a k6 load test whose thresholds are the scenario's response measure. k6 exits with an error when a threshold fails, which fails the pipeline job.

```
# Fitness function: checkout keeps accepting orders when payments hang (illustrative)
# Runs weekly in staging; toxiproxy, api and ledger are test fixtures.
def test_orders_survive_payment_provider_outage(toxiproxy, api, ledger):
    toxiproxy.add_latency("payment-provider", latency_ms=30_000)   # provider hangs
    placed = [api.place_order(sample_order()) for _ in range(200)]

    assert all(r.status in (201, 202) for r in placed)           # still accepted
    assert all(r.json()["state"] == "PENDING_PAYMENT" for r in placed)
    assert max(r.elapsed_seconds for r in placed) < 10           # told within 10 s

    toxiproxy.remove_toxics("payment-provider")                   # provider recovers
    ledger.wait_until_settled(timeout_seconds=300)
    assert ledger.lost_orders(placed) == []                       # none lost
```

Fitness function 2: a chaos experiment written as a test (illustrative; the fixtures wrap a fault-injection proxy, the API and the payments ledger).

```
# .importlinter: only the POS adapters may use vendor SDKs
[importlinter]
root_package = fooddelivery
include_external_packages = True

[importlinter:contract:pos-vendors-isolated]
name = Vendor POS SDKs are used only inside the POS adapters
type = forbidden
source_modules =
    fooddelivery.restaurant.domain
    fooddelivery.restaurant.api
    fooddelivery.ordering
forbidden_modules =
    tillpro_sdk
    kitchenlink

[importlinter:contract:adapters-independent]
name = POS adapters do not import each other
type = independence
modules =
    fooddelivery.restaurant.pos_adapters.tillpro
    fooddelivery.restaurant.pos_adapters.kitchenlink
```

Fitness function 3: import-linter contracts that keep vendor-specific code inside the adapters, run with lint-imports on every build.

Data compatibility is protected in the same way. Before a change to the `OrderPlaced` event can merge, the pipeline asks the schema registry whether the new schema is compatible with the latest registered version under the subject's compatibility rule, and fails if it is not:

```
# Data fitness function: block the merge if the new event schema would break consumers
curl -s -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
  --data @build/order-placed-v7.request.json \
  "$SCHEMA_REGISTRY_URL/compatibility/subjects/order-placed-value/versions/latest" \
  | jq -e '.is_compatible == true'
```

A data fitness function using a schema registry's compatibility endpoint; jq -e returns a non-zero exit code, failing the job, when the schema is incompatible.

Designing fitness functions that last

Start small and important. Pick the three to five characteristics that the utility tree rates highest, write one fitness function for each, and make sure they are fast and reliable before adding more. A dozen trusted checks are worth more than a hundred that fail at random.

Use ratchets for existing systems. When a codebase already violates a rule, record the current violations as a baseline and fail only on new ones; tighten the baseline as the team pays the debt down. The same works for performance: fail on regressions of more than an agreed margin against the last release while working towards the scenario's target.

Make thresholds explicit and owned. Keep thresholds in version-controlled configuration next to a reference to the ADR that justifies them, with a named owner. When a fitness function fails, the owner decides between fixing the code and changing the rule, and the decision is recorded. Treat a fitness function that is frequently overridden as a finding: either the rule is wrong or the architecture is under real pressure.

Keep the pipeline fast. Atomic checks belong in every build; heavy holistic checks such as ten-minute load tests run nightly, before release or on the release candidate. Continual checks in production need the same care as any alert: tied to user impact, with a runbook, and silent when nothing is wrong.

WATCH OUT

Fitness function anti-patterns

- Flaky checks: a load test run on shared, noisy infrastructure fails randomly, so teams learn to rerun it until it passes and stop believing it.
- Gaming the measure: when a number becomes a target, people optimise the number (Goodhart's law, Section 3.8); prefer checks tied to real scenarios over abstract metrics.
- Thresholds without reasons: nobody knows why the limit is 15, so nobody dares change it or everybody ignores it.
- Checks that only report: a dashboard nobody watches is not a fitness function.
- Automating the easy and ignoring the important: dozens of style rules but no check on the order path's latency or the payment fallback.
- Fitness functions as a substitute for thinking: they protect decisions already made; they do not tell you which decisions to make.

Questions people ask

Are fitness functions just tests?

Many are implemented as tests, but their subject is different. Ordinary tests check that features behave correctly; fitness functions check architectural characteristics such as structure, latency, resilience or compatibility. Some, such as production monitors and manual reviews, are not tests at all.

Who writes fitness functions: architects or developers?

Both. Architects usually identify the characteristics and thresholds and often write the first checks, which is a good way to stay hands-on (Section 1.4). Teams own and maintain them like any other code.

How do fitness functions relate to SLOs?

An SLO with an alert on its error budget is a continual, holistic fitness function for availability or latency in production (Section 9.5). Pipeline fitness functions try to catch the same regressions before they reach users.

CHECK YOUR UNDERSTANDING

▼ **What is the difference between a metric and a fitness function?**

A metric describes a property; a fitness function encodes a decision with a threshold and an action, so breaking it fails a build or raises an alert and forces a conscious choice.

▼ **Give one example each of an atomic and a holistic fitness function.**

Atomic: a rule forbidding imports from the web layer into the domain. Holistic: a load test measuring order latency with security, logging and caching all active.

▼ **What does a latency budget add to an end-to-end latency fitness function?**

It splits the end-to-end target into per-service limits that each service checks in its own pipeline, so a regression is detected in the service that caused it.

▼ **What are the three parts of the definition of evolutionary architecture?**

It supports guided (by fitness functions), incremental (small, safe changes and releases) change across multiple dimensions (code, data, security, operations, teams and more).

▼ **Try it: Turn each into a fitness function, with trigger and threshold: "95% of order confirmations within 1.5 s at 100 orders/s", and "Orders must never import Payments' internals". Where would each run, and what happens when one fails?**

Latency: a holistic, dynamic load test at 100 orders a second, failing if p95 exceeds 1.5 s, run nightly or before release because it is slow. Dependency rule: an atomic, static check with ArchUnit or import-linter, failing on any forbidden import, run on every build. A failure forces a decision: fix the code, or change the rule and record why. Each needs an owner and a link to its scenario.

SOURCES

- Building Evolutionary Architectures, 2nd edition (O'Reilly): https://oreilly.com/library/view/building-evolutionary-architectures/9781492097532
- Evolutionary Architecture: book site with fitness function katas: https://evolutionaryarchitecture.com/
- Grafana k6 documentation: Thresholds: https://grafana.com/docs/k6/latest/using-k6/thresholds/
- import-linter documentation: Forbidden contracts: https://import-linter.readthedocs.io/en/stable/contract_types/forbidden/

SECTION 11.4 OF 11.8 · ADVANCED · 28 MIN READ · LAST REVIEWED NOVEMBER 2026

Architecture governance in agile organisations

BEFORE YOU READ THIS

This section builds on:

- 1.6 Decisions and trade-offs
- 4.5 Decision records
- 11.3 Fitness functions

Terms it uses from earlier: Architecture Decision Record (ADR), Fitness function, Enterprise architecture

THE IDEA IN BRIEF

A city cannot ask the mayor to approve every new kitchen. Instead it sets a few principles (keep the streets walkable), some building codes that inspectors check automatically, and a small number of decisions that really do need a committee, such as a new bridge. Residents can do most things themselves, but for anything that affects the neighbours they are expected to ask the neighbours first. Architecture governance in agile organisations works the same way: teams decide most things themselves, automatic checks enforce the rules a machine can check, and bigger decisions are taken after asking the people they affect.

WHAT YOU NEED TO KNOW

- Architecture governance makes sure significant decisions are made well, consistently with the organisation's goals, and actually followed; its purposes include alignment, risk and compliance, limiting unnecessary diversity, protecting shared qualities and spreading learning.
- Principles give durable direction, standards make specific choices mandatory, guardrails keep work within limits automatically, and gates stop work for approval; prefer guardrails where rules are objective and keep gates early and few.
- TOGAF-style principles have a name, statement, rationale and implications; a good principle could reasonably be chosen the other way and settles real decisions.
- In the advice process (Andrew Harmel-Law, 2021) anyone may make an architectural decision after seeking advice from those affected and from experts, recorded in an ADR; an advisory forum gives visibility but does not decide.
- A technology radar places technologies in rings such as Adopt, Trial, Assess and Hold with written reasons; an internal radar records the organisation's own experience and steers new choices.

- Policy as code (for example Open Policy Agent with Rego) and fitness functions automate compliance and produce audit evidence as a by-product.

What architecture governance is for

Architecture governance is the set of practices by which an organisation makes sure its significant technical decisions are made well, are consistent with its goals and are actually followed. It answers four questions: who may make which decisions, how they should make them, what direction they should take, and how anyone knows whether the result conforms.

The purposes are practical. Governance aligns technology with strategy, so that fifty teams do not choose fifty ways to store customer data. It manages risk and compliance: data protection, security, accessibility and sector regulation all need evidence that rules are followed. It limits unnecessary diversity, because every additional database, language or messaging product needs skills, patching and on-call support. It protects qualities that no single team owns, such as end-to-end latency, security and cost. And it spreads learning, so that a lesson paid for by one team is not paid for again by the next.

Agile organisations feel a tension here. The Agile Manifesto's principle that the best architectures emerge from self-organising teams (Section 1.8) pulls towards autonomy; scale, regulation and shared infrastructure pull towards alignment. Henrik Kniberg's descriptions of Spotify's engineering culture popularised the phrase *aligned autonomy* for the goal: teams free to decide how, within a shared understanding of what and why. Governance is the machinery that makes that possible. Done badly, it becomes the slow approval process of Section 11.2; done well, teams barely notice it, because the right thing is also the easy thing.

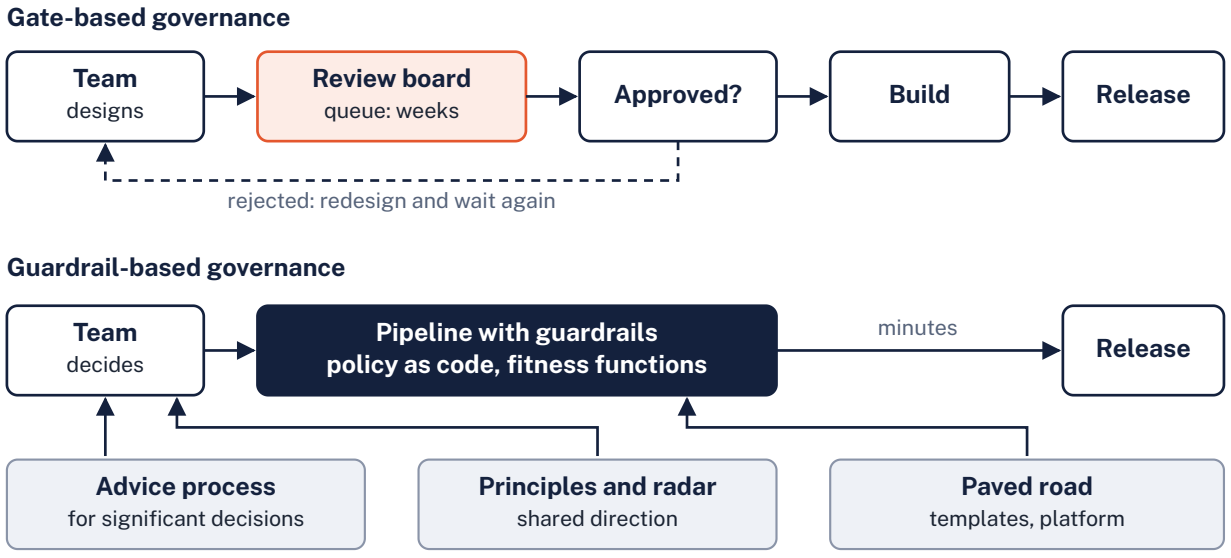
Principles, standards, guardrails and gates

Governance uses four kinds of instrument, which differ in how much they constrain and when they act. Principles express durable direction and leave room for judgement. Standards are specific, mandatory choices (all services expose OpenTelemetry traces; all public APIs are described in OpenAPI). Guardrails keep teams within safe limits while they work, usually automatically and continuously. Gates stop work until someone approves it.

Instruments of governance

Instrument	Example	Strength	Risk
Principle	Each piece of business data has exactly one owning service	Guides many decisions, including unforeseen ones	Too vague to act on, or ignored in practice
Standard	All services emit OpenTelemetry traces and structured JSON logs	Consistency, shared tooling, easy onboarding	Becomes outdated; blocks better options; exceptions pile up
Guardrail	The pipeline rejects container images from unapproved registries	Continuous, fast, objective, no queue	Only works for rules a machine can check; can be bypassed if nobody owns it
Gate	Security sign-off before processing a new category of personal data	Accountable human judgement for high-stakes, irreversible decisions	Delay, late feedback, avoidance and rubber-stamping

Gates versus guardrails



A gate makes every significant design wait for approval; guardrails check objective rules continuously and leave people free for the decisions that need judgement.

The modern consensus, though not a universal one, is to prefer guardrails over gates wherever a rule can be stated objectively, to keep gates for the few decisions where accountable human judgement is genuinely required, and to hold those gates early and time-boxed. A paved road (Netflix's term; Spotify calls it a golden path) complements both: when the supported way of

building a service already includes approved images, logging, security scanning and deployment, most teams conform without being told, and governance effort can concentrate on deliberate departures. Section 10.7 covers the platforms that provide such roads.

IF YOU ARE NEW TO THIS

Step by step: one rule as a gate and as a guardrail

This section's hardest idea is choosing the right instrument for each rule: why a rule a machine can check belongs in a guardrail, and why a few decisions still need a gate. Follow two rules from the table above.

- Take the rule that every container image must come from the organisation's scanned registry. Suppose it is enforced by a gate: each team lists its images for the monthly review board to approve.
- The team waits weeks for a slot. The board checks the list once, so an image changed the following month is never checked. And because the queue is slow, teams learn to call changes minor to avoid it, as Section 11.2 described.
- Now enforce the same rule as a guardrail: the pipeline rejects any deployment whose image comes from anywhere else. It checks every release, in minutes, with no queue, and its log is evidence for an auditor.
- This works because the rule is objective: a machine can answer yes or no. The guardrail still needs an owner and a route for exceptions, or people will find ways round it.
- Take a second rule: security sign-off before the platform processes a new category of personal data. No machine can judge whether the purpose and risks are acceptable, so an accountable person must, and a gate is right. Hold it early, time-box it and state in advance what evidence it needs.
- **The lesson:** if a machine can check a rule, make it a guardrail; keep gates for the few high-stakes decisions that need accountable human judgement, and hold them early. Decisions that need judgement but not approval go through the advice process, later in this section.

Writing architecture principles

TOGAF, the Open Group's enterprise architecture framework (the TOGAF Standard, 10th Edition, 2022), gives a widely used template for principles with four parts. The **name** is short and memorable. The **statement** says the rule in one or two unambiguous sentences. The **rationale** explains the business benefit, in terms that a non-technical sponsor could accept. The **implications** say what the principle requires in practice: what must be done, what it costs and what it rules out. TOGAF also lists criteria for a good set of principles: understandable, robust (precise enough to decide difficult cases), complete, consistent and stable.

Two tests catch most weak principles. First, could a sensible organisation choose the opposite? "Systems should be secure" fails; nobody would choose insecurity. "Prefer buying to building for generic capabilities" passes, because some organisations rightly prefer to build. Second, does the principle change a real decision? If no one can name a choice it would settle, it is a slogan. Andrew Harmel-Law asks for a set that is neither too small to cover the key needs nor too large for teams to remember, sourced from the teams themselves and tied to business strategy, rather than general coding maxims or practices. Fewer, sharper principles are remembered and used; long lists are not.

WORKED EXAMPLE

Worked example: three principles for the food-delivery platform

Written by the architects with the team leads, discussed at the advisory forum and published alongside the radar. Each follows the TOGAF structure.

P1 name	**One owner for every fact**
Statement	Each kind of business data (orders, menus, courier positions, payments) has exactly one owning service, which alone writes it; others use its API or its events.
Rationale	Shared writable data couples teams' releases and makes errors untraceable (Section 7.1). Clear ownership lets teams change their schemas and deploy independently, which is how the business ships weekly.
Implications	No service reads or writes another's tables. Reporting uses published events or a warehouse. New data needs a named owner before it is stored. A fitness function checks database credentials per service.
P2 name	**Asynchronous between domains**
Statement	Domains integrate through published events by default; synchronous calls across domains are used only where the user is waiting for the answer.
Rationale	Synchronous chains multiply unavailability and add latency (Section 6.1); the Friday peak and partner outages are the business's biggest operational risks.
Implications	Teams design for eventual consistency and idempotent consumers. Each synchronous cross-domain call has a timeout, a fallback and a latency budget. More operational investment in the broker and tracing.
P3 name	**Buy generic, build core, stay portable**
Statement	Generic capabilities (identity, payments, email, monitoring) are bought or used as managed services; differentiating capabilities (dispatch optimisation, restaurant tools) are built. Business logic stays behind ports, not inside a provider's proprietary services.
Rationale	Engineering time is the scarcest resource and should go where the business competes (core subdomains, Section 5.7), while avoiding lock-in that would weaken negotiating power (Section 10.8).
Implications	Build-versus-buy is an explicit question in every relevant ADR. Managed services are accessed through adapters. Some managed features with deep lock-in are declined even when convenient.

Each principle settles real, recurring choices, and a sensible organisation could choose otherwise.

Implications include costs, not only benefits.

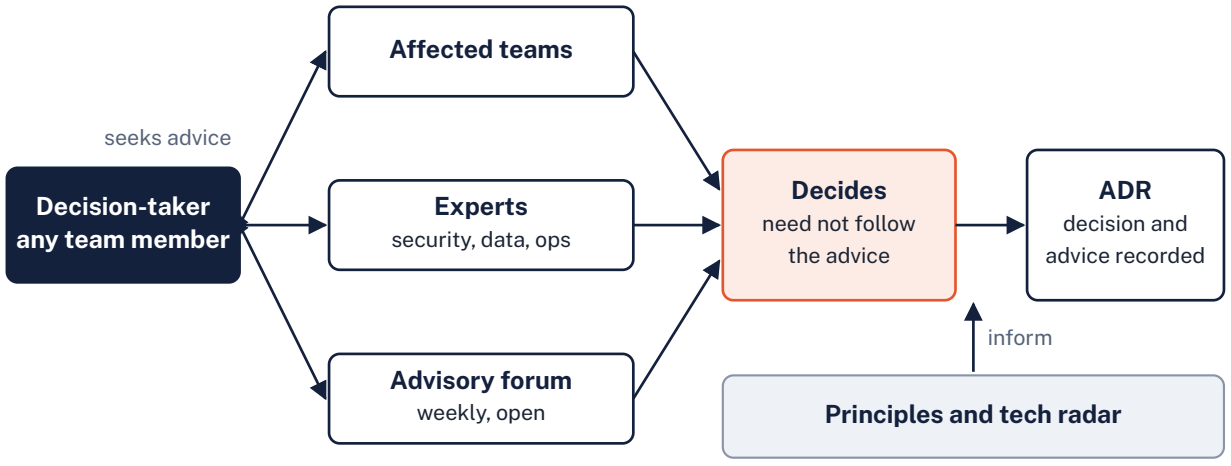
P1 and P2 are partly enforceable as guardrails; P3 needs human judgement, so it is applied through the advice process.

Decentralised decision-making: the advice process

If decisions are no longer funnelled through a board, who makes them? Andrew Harmel-Law's article *Scaling the Practice of Architecture, Conversationally* (martinfowler.com, December 2021) proposed an answer borrowed from self-managing organisations: the advice process. Its single rule is that anyone may make an architectural decision, provided that before deciding they seek advice from everyone who will be meaningfully affected and from people with relevant expertise. The decision-taker must hear and record the advice but need not follow it; the aim is broad input, not consensus. Accountability stays with the person or team who decides, which is what makes the rule work: people who know they own the consequences seek advice seriously.

Harmel-Law describes four supporting elements. Decision records (ADRs, Section 4.5) are the thinking and recording tool, and include the advice received, from whom and when. An architecture advisory forum, typically weekly for about an hour and open to anyone, gives visibility to new spikes, proposed decisions and revisited ones; crucially, it offers advice but does not decide. Team-sourced architectural principles give shared direction. And an internal technology radar shows the organisation's current technology landscape and stance.

The advice process



Advice flows in from those affected and from experts; the decision and the advice are recorded together, and accountability stays with the decision-taker.

The advice process has failure modes worth knowing. Decision-takers may consult only the people likely to agree; a published list of who must be consulted for which kinds of decision (security for anything touching personal data, the platform team for new infrastructure) helps. Advice may be sought too late, after the code is written; making the proposed ADR the start of the work, not the end, helps. And some people find it uncomfortable at first that the forum does not decide. In regulated environments the process can coexist with a small number of formal gates, applied only where the law or the organisation's risk appetite demands them.

Technology radars

Thoughtworks has published its Technology Radar twice a year since 2010. Each edition places a hundred or so blips (techniques, tools, platforms, and languages and frameworks: the four quadrants) in rings that express a recommendation. Adopt means proven and recommended; Trial means worth pursuing on a project that can handle the risk; Assess means worth exploring to understand its effect; and the outermost ring, called Hold for most of the radar's history and Caution in its most recent editions, means proceed with care. Each blip has a short written rationale, which is the most valuable part.

Many organisations build their own radar, for themselves rather than the industry. Thoughtworks provides an open-source Build Your Own Radar tool that renders a radar from a spreadsheet or CSV file with the columns name, ring, quadrant, isNew and description. The value lies less in the picture than in the process: teams nominate blips, a small group with representatives from many teams discusses them, rings are assigned with a written reason, and the radar is refreshed every quarter or half-year. A Hold ring is a gentle but effective governance tool: it says "no new uses" without forbidding what already exists. A radar differs from an approved-technology list in being explicitly provisional and reasoned, and in recording what the organisation has learned, including its mistakes.

WORKED EXAMPLE

Worked example: a radar entry and the decision record behind it

The ordering team wants to replace its polling outbox relay (Section 6.7) with change data capture using Debezium, which reads the database's write-ahead log and publishes changes to Kafka. No team has used Debezium before, so the decision affects the platform team, which runs Kafka Connect, and the data team, which owns database operations.

1. Draft	The team writes ADR-0031 with status Proposed and presents it at the weekly advisory forum
2. Seek advice	Platform, data and security teams comment on the pull request; the forum suggests considering the cloud provider's managed CDC
3. Decide	The ordering team accepts most advice, rejects the managed option because of principle P3 (portability), and records why
4. Guard	A fitness function on event delay (p99 under 2 s) and an alert on replication slot lag are added
5. Radar	Debezium enters the radar in Trial, limited to ordering; polling relays move to Hold
6. Review	After a quarter the radar group reviews the experience; if positive, Debezium moves to Adopt and the ADR's review note is closed

The radar entry is the short public summary; the ADR holds the reasoning and the advice. Each points to the other.

Trial is limited in scope, which keeps the decision reversible while the organisation learns.

Nothing waited in a queue: the decision took about a week, most of it spent collecting genuinely useful advice.

```
name,ring,quadrant,isNew,description
Debezium for outbox relay,Trial,Tools,TRUE,"Change data capture from the order database's write-ahead log to Kafka, replacing polling relays. Trial in ordering only; see ADR-0031. Owner: ordering team. Review: Q3."
Polling outbox relay,Hold,Techniques,FALSE,"Adds load and latency at peak; no new uses. Existing relays move when Debezium leaves Trial."
```

The radar entries in Build Your Own Radar's CSV format (name, ring, quadrant, isNew, description).

```
# ADR-0031: Use Debezium change data capture for the ordering outbox relay

Status: Accepted (2026-05-14)   Decision-taker: Priya N., ordering team
Context: The polling relay queries the outbox table every 500 ms. At 100 orders/s
it adds measurable database load and up to 1 s of event delay to dispatch.
Options: (1) tune polling; (2) Debezium CDC on the write-ahead log; (3) managed CDC
from the cloud provider.
Decision: Option 2, for the ordering service only, as a Trial on the radar.
Advice received:
- Platform team (2026-05-06): supports; will run Kafka Connect, needs runbook.
- Data team (2026-05-07): concern about replication slot growth if Connect stops;
  asks for an alert on slot lag. Accepted.
- Security (2026-05-08): CDC user must have replication rights only. Accepted.
- Advisory forum (2026-05-09): suggested option 3; rejected for portability (P3).
Consequences: lower load and delay; a new component to operate; slot-lag alert
and a fitness function on event delay (p99 under 2 s) added.
Review: after one quarter, or before any second service adopts it.
```

ADR-0031 in a compact Nygard-style format, with the advice recorded as the advice process requires.

The role of enterprise architecture today

Enterprise architecture (Section 1.2) has had an uneasy reputation in agile organisations: central teams producing frameworks, standards and approval processes that delivery teams experienced as obstacles. The role has been changing. In organisations where it works well, enterprise architects curate the principles and the radar rather than dictating them, maintain the map of business capabilities and systems that no single team can see, make or facilitate the genuinely organisation-wide one-way-door decisions (identity, core data, cloud strategy, major vendors), run the advisory forum, and shape the platform and paved road with platform teams.

Gregor Hohpe's image of the architect who rides the elevator between the boardroom and the engine room (Section 11.7) describes the skill this needs: translating strategy into technical direction and technical constraints into business language. Practitioners still disagree about how much central authority is healthy. Some argue that a strong central function is essential in regulated, merger-heavy or highly integrated enterprises; others argue that most of its value can be delivered through platforms, principles and advice. The answer depends on the organisation's size, regulation and how much of its systems landscape is genuinely shared.

Compliance through automation

Whatever the organisational model, the cheapest compliance is automated. Fitness functions (Section 11.3) check architectural characteristics. Policy as code does the same for organisational rules: the rules are written in a machine-readable language, kept in version

control, reviewed and tested like code, and evaluated automatically against infrastructure definitions, deployment manifests or API requests. Open Policy Agent, a graduated Cloud Native Computing Foundation project, is a widely used general-purpose engine with its own policy language, Rego; Conftest applies Rego policies to configuration files in a pipeline, and admission controllers such as OPA Gatekeeper or Kyverno enforce policies inside Kubernetes clusters.

Automated compliance has a further benefit that regulated organisations value highly: evidence. A pipeline log showing that every release passed the same checks is far stronger evidence for an auditor than a signed form, and it is produced at no extra cost.

```
package main

import rego.v1

# Guardrail: images must come from the organisation's scanned registry
deny contains msg if {
    input.kind == "Deployment"
    some c in input.spec.template.spec.containers
    not startswith(c.image, "registry.fooddelivery.example/")
    msg := sprintf("%s: image %s is not from the approved registry",
[input.metadata.name, c.image])
}

# Guardrail: every container declares a memory limit
deny contains msg if {
    input.kind == "Deployment"
    some c in input.spec.template.spec.containers
    not c.resources.limits.memory
    msg := sprintf("%s: container %s has no memory limit", [input.metadata.name, c.name])
}
```

Two guardrails as Rego policies, checked against Kubernetes deployment manifests with Conftest (conftest test deploy/) on every pipeline run.

Balancing autonomy and alignment

The practical question is which decisions are governed how. A useful rule follows from Section 1.6: the more reversible and local a decision, the closer to the team it should be made; the more irreversible and widely felt, the more advice and, occasionally, approval it needs. The table sets out a typical allocation; each organisation tunes its own.

Who decides what

Kind of decision	Who decides	Mechanism	Example
Local and reversible	The team	Team judgement; ADR if significant	A library inside one service; internal module structure
Affects other teams, reversible at moderate cost	The team, after advice	Advice process, advisory forum, radar	Adopting Debezium; a new event published to others
Organisation-wide or hard to reverse	Named senior decision-taker after wide advice	Advice process with enterprise architects; principle or radar change	Choice of identity provider; primary cloud provider
Objective, checkable rules	Nobody case by case	Guardrails: policy as code and fitness functions	Approved image registries; no public storage buckets
Regulated or high-stakes	Accountable approver	Early, time-boxed gate with evidence	Processing health data; safety-related functions

Governance should itself be measured. Useful signals are how long significant decisions take from proposal to acceptance, how many exceptions to standards are requested and granted, what share of services are on the paved road, how many Hold technologies are still spreading, and whether incidents trace back to decisions that skipped advice. If decisions are slow, governance is too heavy; if incidents keep tracing back to unconsulted decisions, it is too light.

DECISION GUIDE

Decision guide: gate, guardrail or advice?

Use this when deciding how a particular rule or kind of decision should be governed.

Q1 **Can the rule be stated objectively and checked by a machine?**

- **Yes** Make it a guardrail: policy as code or a fitness function in the pipeline, with an owner and an exception route.
- **No** Continue.

Q2 **Is the decision local to one team and cheap to reverse?**

- **Yes** Let the team decide; record an ADR if it is significant.
- **No** Continue.

Q3 **Does law, regulation or the organisation's risk appetite require an accountable sign-off?**

- **Yes** Use a gate, but hold it early, time-box it and make the evidence it needs explicit.
- **No** Continue.

Q4 **Is it a one-way door affecting many teams?**

- **Yes** Use the advice process with wide consultation, including enterprise architects, and a named senior decision-taker.
- **No** Use the advice process: consult those affected and the experts, then decide and record.

WATCH OUT

Common governance mistakes

- Principles that nobody could disagree with, which therefore decide nothing.
- Standards without an exception route, which teams then ignore quietly.
- An advisory forum that slowly turns into an approval board.
- A radar maintained by one person, which records opinions rather than the organisation's experience.
- Gates late in delivery, where the only realistic outcomes are approval or expensive rework.
- Automating rules nobody owns, so false positives are bypassed rather than fixed.

CHECK YOUR UNDERSTANDING

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

A guardrail keeps work within limits continuously, usually through automated checks, without stopping it; a gate stops work until someone approves it.

▼ What are the four parts of a TOGAF-style architecture principle?

Name, statement, rationale and implications.

▼ In the advice process, who makes the decision and must they follow the advice?

The person or team taking the decision, who must seek and record advice from those affected and from experts but need not follow it.

▼ What does placing a technology in a radar's Hold ring usually mean for existing uses?

It means no new uses should start; existing uses are not forbidden, though they may be migrated over time.

▼ Try it: Every new database needs approval from a monthly architecture board, and queues reach six weeks. Teams now avoid asking. Security must still be assured. How would you redesign this governance using guardrails and the advice process?

Replace the gate with guardrails where rules are objective: policy as code can check encryption, backups and approved engines automatically and give audit evidence. Publish a radar saying which databases are Adopt, Trial or Hold. For anything unusual, use the advice process: the team decides after consulting affected people and experts, and records an ADR. The forum gives visibility but does not approve. Keep one early gate, if any, only for truly high-risk changes.

SOURCES

- Andrew Harmel-Law: Scaling the Practice of Architecture, Conversationally (martinfowler.com, 2021): https://martinfowler.com/articles/scaling-architecture-conversationally.html
- Thoughtworks Technology Radar: FAQ: https://www.thoughtworks.com/radar/faq
- Thoughtworks: Build Your Own Radar: https://www.thoughtworks.com/radar/byor
- Open Policy Agent documentation: https://www.openpolicyagent.org/docs/latest/

SECTION 11.5 OF 11.8 · ADVANCED · 26 MIN READ · LAST REVIEWED NOVEMBER 2026

Team Topologies and socio-technical design

BEFORE YOU READ THIS

This section builds on:

- 3.7 Conway's law
- 5.7 Domain-driven design
- 10.7 Platform engineering

Terms it uses from earlier: Conway's law, Bounded context, Internal developer platform (IDP)

THE IDEA IN BRIEF

Think of a restaurant group. Each restaurant's kitchen serves its own diners from start to finish. A central supplier delivers ingredients and equipment to every kitchen from a simple catalogue, so cooks need not run a farm. A specialist pastry workshop makes the one product too difficult for every kitchen to master. And a visiting head chef spends a few weeks teaching a kitchen a new technique, then leaves. Team Topologies gives software organisations the same four kinds of team and three ways for them to work together, so each team owns a job it can actually keep in its head.

WHAT YOU NEED TO KNOW

- Socio-technical architecture designs software structure and team structure together, because Conway's law makes each shape the other.
- Team Topologies (Skelton and Pais, 2019; second edition 2025) defines four team types: stream-aligned (the default), platform, enabling and complicated-subsystem.
- Teams interact through three deliberate modes: collaboration (time-boxed discovery), X-as-a-Service (consume a clear interface) and facilitating (help another team gain a capability).
- Team cognitive load, adapted from John Sweller's cognitive load theory (1988), limits how much a team can own; reduce extraneous load with platforms and automation and bound intrinsic load with well-placed boundaries.
- A team API documents what a team offers and expects, both technical interfaces and ways of working; a thinnest viable platform is the smallest platform that genuinely speeds up its users.

- Fracture planes such as bounded contexts, regulatory scope, change cadence, risk and performance isolation show where to split software and teams; technology alone is usually the weakest seam.
- Reorganising teams does not restructure coupled code, so team changes and architecture changes should be planned and evolved together.

Team design is architecture work

Chapter 3 showed that Conway's law is not a curiosity but a force: the system a group of teams builds tends to copy the way those teams communicate. If that is true, then deciding how teams are formed, what each owns and how they interact is an architectural decision, with consequences as lasting as choosing a database or an integration style. The discipline of designing software structure and team structure together is called socio-technical architecture.

The most influential practical framework for doing this is Team Topologies, by Matthew Skelton and Manuel Pais (IT Revolution, 2019; second edition 2025). It starts from a simple aim, *fast flow*: a steady, sustainable stream of small changes from idea to production, with little waiting between teams. It then offers a small vocabulary for designing organisations that achieve it: four team types, three interaction modes, a limit called team cognitive load, and a technique for finding the seams along which software and teams can be split.

Several premises run through the approach. The team, not the individual, is the unit of delivery: work is given to teams, ownership belongs to teams, and teams are long-lived, so that they accumulate knowledge of their domain and code instead of being assembled for a project and scattered afterwards. Teams are small, usually five to nine people, because trust and shared understanding decline quickly beyond that size (the authors draw on Robin Dunbar's research on group sizes). And team boundaries should be designed for the flow of change, not for the convenience of reporting lines or the history of who built what.

For architects this changes the job. A proposed service boundary is only good if some team can own it fully; a proposed team is only viable if the code it owns lets it work without asking permission from three other teams. The architect who draws boxes without asking who will own each one is designing half the system.

> **IF YOU ARE NEW TO THIS**
>
> ### If you are new to this
>
> A *value stream* is the sequence of work that turns a need into something a customer or user benefits from, for example everything needed to let customers schedule an order for later. *Cross-functional* means the team contains all the skills that work needs (design, front end, back end, testing, operations) rather than handing work to other teams. Section 3.7 of this course explains Conway's law and the mirroring between organisation and system that this section builds on.

The four fundamental team types

Team Topologies argues that almost every team in a software organisation should be one of four types. The point of so few types is clarity: each type has a different purpose, a different way of measuring success and a different relationship with the other teams, so knowing a team's type tells everyone what to expect from it.

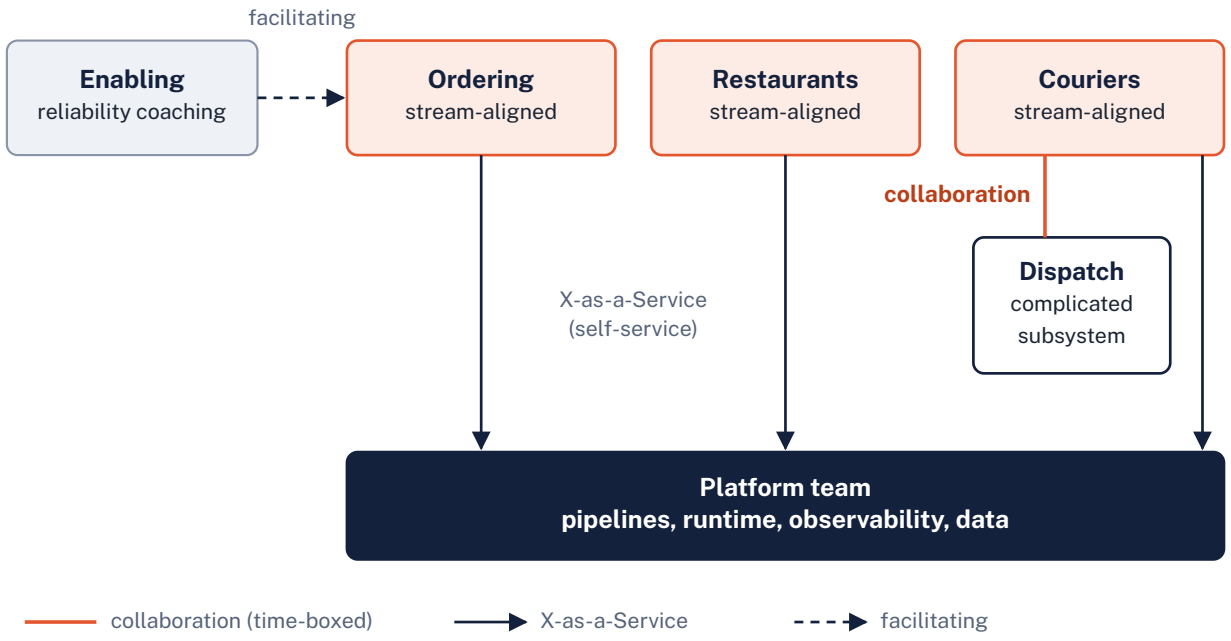
A stream-aligned team is aligned to one flow of change: a product, a customer journey, a user group or a business capability. It owns its part end to end, from discovery through build to running it in production, and should be able to deliver most changes without waiting for anyone. Stream-aligned teams are the default and should be the large majority; the other three types exist to reduce their burden.

A platform team provides internal services (deployment pipelines, a runtime environment, observability, data infrastructure) as a product that stream-aligned teams consume through self-service. A good platform is judged by its users' speed, not by its own feature list. An enabling team consists of specialists who help other teams acquire a capability they lack, such as test automation, observability practices or a new framework, and then move on; its success is measured by the other team no longer needing it. A complicated-subsystem team owns a part that needs rare expertise, such as a pricing engine built on actuarial models, a route-optimisation solver or a video codec, so that stream-aligned teams can use it without having to master it. The book stresses that this type should be used sparingly and only where the specialism is genuine.

The four team types compared

Team type	Purpose	Owns	Healthy signs	Food-delivery example
Stream-aligned	Deliver a continuous flow of change for one part of the business	Features, code, data and operation of its stream, end to end	Releases independently; most changes need no other team; on-call load is bearable	Customer ordering team: menus shown to customers, basket, checkout, order tracking
Platform	Reduce the cognitive load of stream-aligned teams through self-service internal products	Internal platform services, their documentation and their reliability	Teams adopt it voluntarily; onboarding a new service takes hours, not weeks	Delivery platform team: pipelines, container runtime, logging and metrics, event broker
Enabling	Close capability gaps in other teams, then leave	Know-how, guides, examples; not production code of other teams	Engagements are time-boxed and end with the other team self-sufficient	Reliability enabling team coaching teams on service level objectives and incident practice
Complicated subsystem	Encapsulate a specialism too deep for every stream-aligned team	One specialist component behind a clear interface	Consumers use it as a service without needing the specialists in the room	Dispatch optimisation team: the algorithm that assigns couriers to orders

Team types and interaction modes



Most teams are stream-aligned; a platform serves them as a service, an enabling team facilitates, and a complicated-subsystem team works closely with one stream for a limited time.

Three interaction modes

Team types say what teams are; interaction modes say how they work together. Skelton and Pais argue that ambiguous, ad hoc interaction between teams is a major source of delay, because every unplanned conversation is a dependency. They therefore recommend that every pair of teams that must interact chooses one of three modes deliberately, for a stated period.

In **collaboration**, two teams work closely together on a shared goal, with blurred responsibilities, for example while discovering how a new capability should be split. It produces fast learning but costs both teams capacity and cognitive load, so it should be time-boxed. In X-as-a-Service, one team provides something (an API, a platform capability, a library) that another consumes with little or no collaboration; this scales well but only works once the boundary is well understood and the interface is good. In **facilitating**, one team, typically an enabling team, helps another learn or remove an obstacle.

The modes are meant to change. A common sequence is discover then settle: a platform team collaborates with one stream-aligned team for a few weeks to design a new capability, then turns it into a self-service offering and moves to X-as-a-Service. The modes also work as a sensing mechanism. If a team that is supposed to consume a service as a service keeps

needing meetings, the interface or the boundary is probably wrong; if two teams have been collaborating for a year, they may really be one team, or the split between them may be in the wrong place.

The three interaction modes

Mode	What it looks like	Use when	Cost or warning sign
Collaboration	Two teams work as one on a shared problem for a while; joint design sessions, shared backlog items	Discovering a new boundary, technology or capability where nobody yet knows the right interface	High cognitive load on both teams; if it lasts indefinitely, the boundary is probably wrong
X-as-a-Service	One team consumes an API, platform or component that another provides, through documentation and self-service	The boundary and interface are well understood and stable	Slow to adapt if the interface is poor; constant support requests show it is not yet a real service
Facilitating	Coaching, pairing, examples and short workshops that help a team gain a capability	A team lacks a skill or is blocked by an obstacle it can learn to remove	Turns into doing the other team's work, which builds dependency instead of capability

Cognitive load as a design constraint

The idea of cognitive load comes from educational psychology. John Sweller's cognitive load theory, first published in 1988, started from the observation that human working memory can hold and process only a few new items at a time, so instruction should be designed not to overload it. The theory distinguishes intrinsic load (the inherent difficulty of the material), extraneous load (difficulty caused by the way the task is presented) and, in later work, germane load (effort that builds lasting understanding). Researchers still debate whether germane load is a separate category, and Team Topologies uses all three as a practical heuristic rather than a measurement.

Section 3.7 introduced the team version of this idea: the total a team must hold in its heads to build, run and change what it owns. What matters here is how to use it as a design constraint. When an architecture assigns a team more than it can understand, the effects are predictable: slower delivery, shallow fixes, fragile on-call, knowledge concentrated in one or two people and, eventually, departures. These are the same symptoms often blamed on poor engineers or poor code.

The book offers rules of thumb for sizing a team's responsibilities by domain complexity: give each domain to a single team; a team can usually handle two or three simple domains; a team that owns a complex domain should own nothing else; and no team should own two complicated domains. Teams can also be asked directly, for example with a short survey on how well they understand their systems, how confident they feel when on call and how much time goes on tooling rather than the domain. Rising ticket queues, long onboarding times and frequent handovers to the one person who understands a component are useful indirect signals.

The architectural responses mirror the three kinds of load. Reduce extraneous load with platforms, automation, paved paths and good documentation. Bound intrinsic load by drawing service and module boundaries so that each team's share of the domain is coherent and of a manageable size, and by putting deep specialisms behind complicated-subsystem interfaces. Protect germane load by leaving time for learning, which is where enabling teams help.

Team APIs and the thinnest viable platform

If teams interact through deliberate modes, each team needs a clear face to the rest of the organisation. Team Topologies calls this the team API: everything the team offers and expects. It includes the technical API (endpoints, events, libraries and their versioning and deprecation rules) but also the human interface: documentation and onboarding guides, the team's roadmap and current priorities, its working practices, the channels where it can be reached, how to request work from it and what response times to expect.

Writing a team API down is a cheap and revealing exercise. It forces a team to say what it owns, which usually exposes unowned services and contested data, and it gives other teams a way to interact without a meeting. Many organisations keep it as a short page in the team's repository, reviewed each quarter alongside the architecture documentation.

The platform deserves particular care because it is where good intentions most often go wrong. A platform built as a large internal project, with a long roadmap and mandatory adoption, can easily add cognitive load rather than remove it. Skelton and Pais recommend the thinnest viable platform: the smallest set of services, documentation and standards that genuinely speeds up its users. For a small organisation that may be a well-kept page listing the cloud provider services teams should use and how; for a large one it may be an internal developer platform. Platform engineering is covered in depth in Chapter 10; the essential point here is that a platform is a product whose customers are other teams, and it should be designed and measured as one.

Fracture planes: where to split software and teams

When a part of the system has grown too big for one team, it has to be split, and the split must work for both the code and the teams. Team Topologies calls the natural seams along which software can be divided fracture planes. The book lists several, and the first is usually the best: a bounded context from domain-driven design, because a boundary that follows the language and rules of the business tends to keep related changes together.

The others are useful when they line up with a real difference in how parts of the system must be treated. Splitting along a technology boundary alone (for example a mobile team and a back-end team) is generally the weakest choice, because most changes then cross it, which is the layered team structure Conway's law warns about.

Fracture planes with food-delivery examples

Fracture plane	Split when	Example
Business domain (bounded context)	Parts of the domain have their own language, rules and experts	Ordering, restaurant onboarding and courier management as separate contexts
Regulatory compliance	One part is subject to rules that should not burden the rest	Card payment handling kept in a small, audited scope for PCI DSS
Change cadence	Some parts change daily and others yearly	Promotions and campaigns change weekly; invoicing rules change rarely
Team location	Teams work in time zones with little overlap	A courier operations team in another country owns courier onboarding
Risk	Some parts can tolerate experiments and others cannot	Recommendation experiments separated from order acceptance
Performance isolation	One part has very different load or latency needs	Live courier location updates separated from menu browsing
Technology	A part needs a genuinely different technology stack	A legacy restaurant tablet integration that only runs on old hardware
User personas	Different user groups have different needs and journeys	Customers, restaurant staff and couriers each have their own app and team

Conway's law, the inverse manoeuvre and its limits

Team Topologies is, in effect, a toolkit for the inverse Conway manoeuvre that Section 3.7 described: decide the architecture you want, then shape teams and interaction modes so that Conway's law pushes the system towards it. Its additions are practical. Team types say what

kind of team should own each part; interaction modes say which team boundaries are meant to be thin (X-as-a-Service) and which are temporarily thick (collaboration); cognitive load says how much each team can own; fracture planes say where to cut.

The limits from Section 3.7 still apply and are worth restating in this setting. Reorganising teams does not restructure a tightly coupled codebase: for months after a reorganisation the new teams will fight the old architecture, which is why team changes and architecture changes should be planned as one programme. Boundaries that are hardened into teams too early make mistakes expensive, so many organisations keep uncertain boundaries inside one team or one modular monolith until they have evidence. And frequent reorganisations destroy the long-lived team knowledge the approach depends on.

Practitioners also disagree about how literally to apply the framework. Supporters value the shared vocabulary, which turns vague complaints (we keep waiting for the data team) into designable questions (what interaction mode do we want with the data team, and what would make it X-as-a-Service?). Critics warn that the four types are often applied as labels to an unchanged organisation, that real teams sometimes mix types (a platform team that also owns a complicated subsystem) and that the framework says little about funding, management structure or incentives. Both points are fair: the types are a design aid, not an org chart, and the authors themselves present them as a starting point for continuous adjustment.

WHERE THIS CAME FROM

Where socio-technical thinking came from

The phrase socio-technical system comes from the Tavistock Institute in London. In 1951 Eric Trist and Ken Bamforth published a study of British coal mining showing that a new, more mechanised mining method had lowered productivity because it broke up small, self-organising work groups. Their conclusion, that technical and social systems must be designed together, is the same insight Conway reached for software in 1968 and Team Topologies applies today.

WORKED EXAMPLE

Worked example: redesigning the food-delivery teams around streams

The food-delivery platform has grown to 38 engineers in five teams organised by technical layer: mobile apps, web front end, back-end API, database and operations. Lead time from idea to production is about six weeks, releases are coordinated fortnightly across all teams, and the last three incidents each took several hours because nobody owned a whole customer journey. The architect and engineering leaders redesign the organisation in five steps.

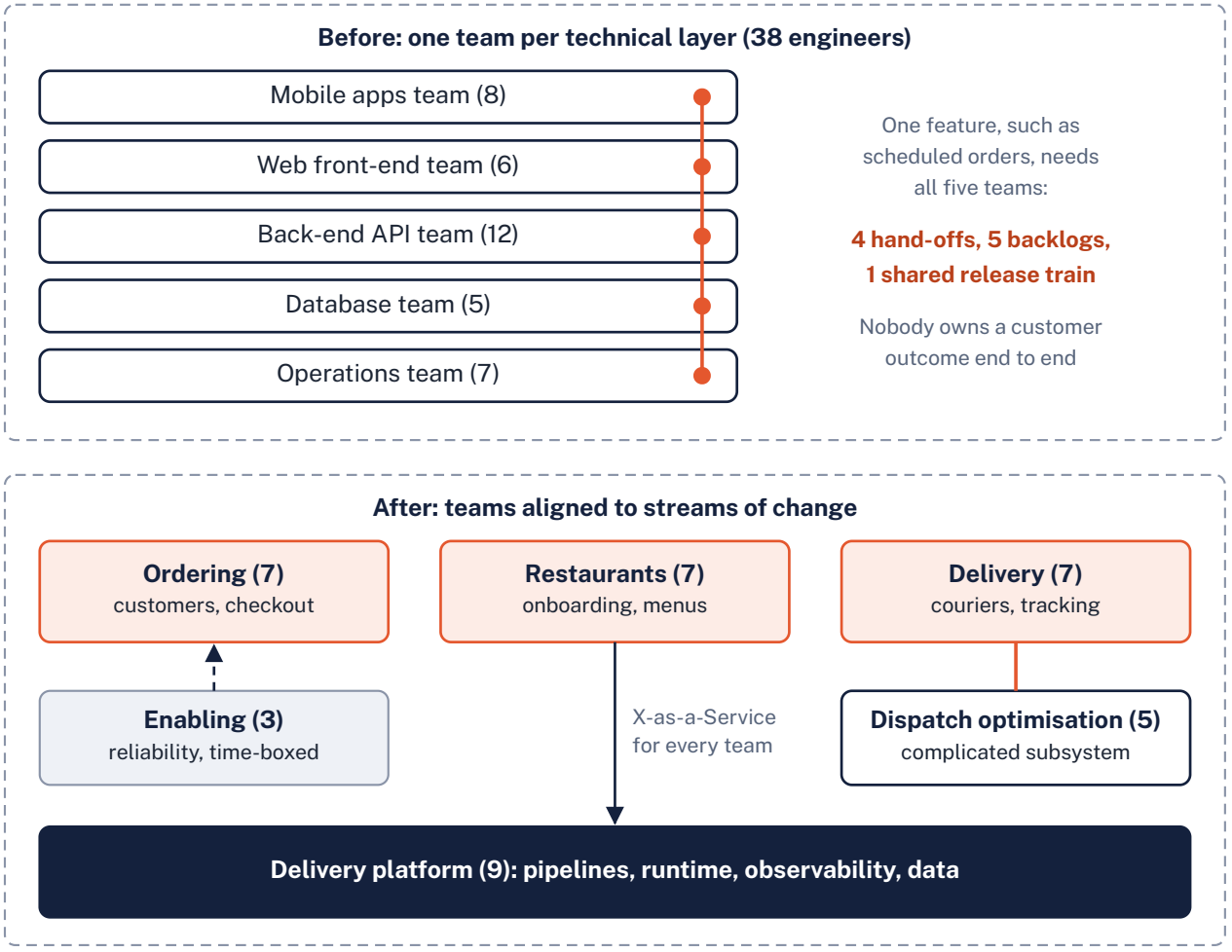
1. Map the streams	Event storming and a review of the last 50 features show three main flows of change: customer ordering (menus, basket, checkout, tracking), restaurant partners (onboarding, menus, opening hours, payouts) and delivery (courier onboarding, assignment, live tracking). 44 of the 50 features fell mainly in one of them.
2. Check the software's fracture planes	Ordering, restaurants and delivery are already separate bounded contexts with their own services and databases (Section 11.1), so the code can follow. Courier assignment uses an optimisation solver that only three engineers understand: a candidate complicated subsystem. Card payments stay with Ordering but in a small, separately audited module (regulatory fracture plane).
3. Size teams by cognitive load	Three stream-aligned teams of 7 (each with mobile, web, back-end, testing and operations skills). A complicated-subsystem team of 5 for dispatch optimisation (the three specialists plus two engineers learning the domain). A platform team of 9 that owns pipelines, the container runtime, observability and the event broker. A small enabling team of 3 for reliability practices. Total: 38.
4. Choose interaction modes	Stream-aligned teams consume the platform as X-as-a-Service from day one, using its existing pipelines. Delivery and Dispatch collaborate for one quarter to redesign the assignment interface, then move to X-as-a-Service. The enabling team facilitates Ordering for eight weeks (service level objectives, on-call runbooks), then moves on to Restaurants.
5. Publish team APIs and measure	Each team writes a one-page team API, every service and database table gets exactly one owning team, and leaders track lead time, deployment frequency, cross-team tickets per feature and an on-call confidence survey every quarter.
Result to look for	After two quarters, most features should be delivered by one team without a coordinated release; persistent cross-team work would show a boundary to revisit.

The redesign starts from flows of change and the software's existing boundaries, not from the current reporting lines.

Specialists are concentrated only where the domain demands it (dispatch), and the platform exists to reduce load on stream-aligned teams, not to control them.

Interaction modes come with end dates, so the organisation is expected to evolve rather than be fixed in a new chart.

Before and after



Layer teams force every feature through every team; stream-aligned teams supported by a platform, an enabling team and one specialist team can deliver most features alone.

WATCH OUT

Common mistakes

- Renaming existing teams with Team Topologies labels while ownership, funding and interaction stay the same.
- A platform team that works through tickets and approvals, so stream-aligned teams wait for it: that is a bottleneck, not a platform.
- An enabling team that ends up permanently doing other teams' work, which builds dependency instead of capability.
- Collaboration with no end date, which quietly merges two teams' cognitive load.
- Splitting teams along technology lines (front end, back end, database) and expecting fast flow.
- Drawing service boundaries first and assigning owners later, leaving orphaned or shared-owned services.

Questions people ask

Are component teams always wrong?
Not always. A team owning a shared component can be the right choice when the component needs deep specialism (a complicated subsystem) or is offered as a genuine self-service product (part of a platform). It becomes a problem when most features need changes in the component and the stream-aligned teams have to queue for it.

Where do architects sit in this model?
Usually not as a separate gatekeeping team. Architects often act like an enabling team, helping stream-aligned teams make good decisions and keeping system-wide qualities visible, and they take part in designing the team structure itself. Section 11.4 discusses how governance can work without gates.

How often should team boundaries change?
Rarely and deliberately. Interaction modes are expected to change every few months; team membership and ownership should be stable for much longer, because long-lived teams are where domain and system knowledge accumulates.

CHECK YOUR UNDERSTANDING

▼ Name three things a team API contains besides its code interfaces.

For example its documentation and onboarding guides, its roadmap and priorities, its working practices, the channels where it can be reached and how to request work from it.

▼ Why are interaction modes expected to change over time?

Because collaboration suits discovering a new boundary, and once the boundary and interface are understood the teams should settle into X-as-a-Service; modes follow how mature the boundary is.

▼ What is the thinnest viable platform, and why aim for it?

The smallest set of services, documentation and standards that genuinely speeds up the teams using it; a larger platform risks adding cognitive load instead of removing it.

▼ Why is a front-end and back-end team split usually a weak fracture plane?

Because most features cross it, so every change needs both teams, hand-offs and coordinated releases, which is the layered structure Conway's law turns into a layered, coupled system.

▼ Try it: At a ticketing company, assign a team type and interaction mode to: a team owning the purchase journey; a small group of specialists on seat-allocation algorithms; a group providing CI and deployment tools; two experts helping teams learn testing.

Purchase journey: stream-aligned, the default. Seat-allocation specialists: complicated-subsystem, offering X-as-a-Service to the journey team, with time-boxed collaboration on new interfaces. Tooling group: platform, offering X-as-a-Service and keeping it the thinnest viable platform. Testing experts: enabling, using the facilitating mode and aiming to leave once the skill has transferred. Check that each team's cognitive load is manageable, and remember renaming teams alone does not change coupled code.

SOURCES

- Team Topologies: key concepts: https://teamtopologies.com/key-concepts
- Martin Fowler: Team Topologies (2023): https://martinfowler.com/bliki/TeamTopologies.html
- Melvin Conway: How Do Committees Invent? (1968): https://melconway.com/Home/Committees_Paper.html

SECTION 11.6 OF 11.8 · ADVANCED · 29 MIN READ · LAST REVIEWED NOVEMBER 2026

Modernising legacy systems: strangler fig and incremental migration

BEFORE YOU READ THIS

This section builds on:

- 1.7 Technical debt
- 5.7 Domain-driven design
- 7.1 Data ownership

Terms it uses from earlier: Technical debt, System of record, API gateway

THE IDEA IN BRIEF

Imagine a town that needs a new bridge. Knocking down the old one and building a replacement would cut the town off for years. Instead, engineers build the new bridge alongside the old, open it one lane at a time, move traffic across gradually and demolish the old bridge only when nothing uses it any more. If a new lane has a problem, traffic simply goes back to the old bridge for a while. Modernising old software works best the same way: replace it piece by piece while it keeps running, so that customers never lose the service they depend on.

WHAT YOU NEED TO KNOW

- A legacy system still delivers value but is hard, risky or expensive to change; modernise for business reasons such as blocked change, lost support, scarce skills or cost, not for fashion.
- Big-bang rewrites fail because value arrives late, the target moves, hidden behaviour is lost and data migration and cutover come last; Spolsky's 2000 essay and TSB's 2018 migration are standard warnings.
- The seven Rs (retire, retain, rehost, relocate, repurchase, replatform, refactor or re-architect) give each application in a portfolio a deliberate strategy; rebuilding is only one option.
- The strangler fig (Fowler, 2004) routes traffic through a facade and moves behaviour slice by slice to new components until the legacy system can be switched off; branch by abstraction does the same inside one codebase.
- Parallel runs compare old and new results in production while the old result stays authoritative; dark launching exercises new paths under real load without users seeing them.

- Data needs a single system of record at every stage; change data capture, event interception and anti-corruption layers keep systems in step, and history is migrated in rehearsed, reconciled cohorts.
- Transitional architecture is a deliberate, temporary cost; a migration is finished only when the old system and the transitional code have been removed.

What makes a system legacy

A legacy system is not simply an old one. It is a system that still does something valuable, often something the organisation could not operate without, but that has become hard, risky or expensive to change. Michael Feathers, in *Working Effectively with Legacy Code* (2004), gave a deliberately provocative definition: legacy code is code without tests, because without tests every change is a gamble. Other common causes are scarce skills (a language or platform few people still know), unsupported technology (an operating system or database past its vendor's end of support), lost knowledge (the people who understood the design have left) and accumulated technical debt in the structure itself, the expensive kind described in Section 1.7.

It is worth saying plainly that most legacy systems are successful systems. They survived because they worked and earned money. That success is also why they are hard to replace: years of real-world use have embedded thousands of small decisions, fixes and business rules that are written down nowhere except in the code and in the habits of the people who use it.

Modernisation therefore needs a reason that can be stated in business terms. Good reasons include: the cost or risk of change is blocking important business goals; the platform is losing vendor support or security patches; skills to maintain it are becoming unavailable; running costs are out of proportion to value; or the system cannot meet new demands for scale, availability or regulation. Weak reasons include fashion, boredom and the belief that new technology is automatically better. Section 11.8 shows how to turn a good reason into a business case.

Why rewrites fail

The obvious response to a painful old system is to start again. In April 2000 Joel Spolsky published an essay, "Things You Should Never Do, Part I", arguing that the decision to rewrite from scratch is among the worst strategic mistakes a software company can make. His example was Netscape, whose browser went nearly three years between major releases while the code was rewritten, during which competitors took the market. His central point was about knowledge: old code looks messy partly because it contains years of bug fixes for situations nobody remembers, and throwing it away throws those fixes away too.

The reasons a big-bang rewrite fails are now well understood. Value arrives only at the end, so the business waits months or years and loses patience. The old system keeps changing while the new one is built, so the target moves and the organisation either freezes the old system (and the business) or pays to make every change twice. Hidden behaviour is missed, because nobody can list every rule the old system enforces. Estimates are optimistic, because the visible features are a fraction of the real work. The data migration and the cutover are left until the end, when they are hardest and riskiest. And the team splits into a prestigious new-system group and a demoralised old-system group. Fred Brooks's second-system effect, from *The Mythical Man-Month* (1975), adds a further risk: designers replacing a system they know tend to load the new one with every idea the old one could not accommodate.

Large migrations that switch over in one step can also fail very publicly. In April 2018 the UK bank TSB moved its customer data and services to a new IT platform; the platform immediately had serious problems, all its branches and a significant proportion of its 5.2 million customers were affected, and the bank did not return to normal operation until December 2018. In December 2022 the Financial Conduct Authority and the Prudential Regulation Authority fined TSB a total of £48.65 million, finding that it had not adequately planned, organised and controlled the migration or managed the risks from its outsourced supplier.

Practitioners do not all agree that rewrites are always wrong. A rewrite can be the right choice when the system is small, its behaviour is well specified and covered by tests, the platform is truly at a dead end, or the business itself has changed so much that the old behaviour is no longer wanted. Even then, the safest rewrites are delivered incrementally, with the old and new systems running side by side, which is the subject of the rest of this section.

Migration strategies: the Rs

When an organisation reviews a whole portfolio of applications, for example before leaving a data centre, it needs a vocabulary for deciding what to do with each one. Gartner described five such options in the early 2010s. In 2016 Stephen Orban of AWS published six strategies, and AWS later added a seventh, giving the seven Rs now used in its migration guidance. Other cloud providers and consultancies use similar lists with slightly different names. The list is most useful as a reminder that rebuilding is only one option among several, and usually the most expensive.

The seven Rs (after AWS Prescriptive Guidance)

Strategy	What it means	Fits when	Architecture change
Retire	Switch the application off and archive its data	Nobody uses it, or its function is duplicated elsewhere	None; remove dependencies on it
Retain	Keep it where it is for now and revisit later	No pressing reason to move, or other work must come first	None for now
Rehost (lift and shift)	Move it unchanged to new infrastructure, such as cloud virtual machines	The main driver is leaving a data centre quickly	None; problems move with it
Relocate	Move whole platforms, such as a virtualised estate, without changing the applications	A compatible platform exists at the destination	None
Repurchase	Replace it with a product, often software as a service	The capability is generic and a good product exists	Integration and data migration; loss of customisation
Replatform	Move it with a few targeted optimisations, such as a managed database	Cheap gains are available without redesign	Small and local
Refactor or re-architect	Redesign it, often incrementally, to meet new quality or business needs	The capability is core, changes often and its structure blocks change	Large; use the incremental patterns below

DECISION GUIDE

Decision guide: choosing a strategy for one application

Ask these questions for each application in the portfolio, in order. The answers come from usage data, the business owner and the architecture, not from technology preference.

Q1 **Does anyone still need what this application does?**

- **Yes** Continue to the next question.
- **No** Retire it, after archiving any data you are legally required to keep.

Q2 **Is there a product that meets the need, for a capability that does not differentiate the business?**

- **Yes** Consider repurchasing, and budget for integration and data migration.
- **No** Continue to the next question.

Q3 **Is there a pressing reason to act now, such as end of support, a data centre exit, cost or change that is blocked?**

- **Yes** Continue to the next question.
- **No** Retain it and set a date to revisit the decision.

Q4 **Is the main problem where it runs rather than how it is built?**

- **Yes** Rehost or relocate it, or replatform where a managed service gives cheap gains.
- **No** Continue to the next question.

Q5 **Is it a core capability whose structure is blocking frequent, valuable change?**

- **Yes** Refactor or re-architect it incrementally, using a strangler fig and branch by abstraction.
- **No** Replatform and contain it: add tests, fix the worst hotspots and avoid a large redesign.

The strangler fig

Martin Fowler named the most widely used incremental pattern after the strangler figs he saw in the rainforests of Queensland in 2001: vines that germinate in the branches of a host tree, grow down to the ground and gradually surround the host until it dies and the fig stands on its own. He first wrote it up as the strangler application in 2004, later renamed it the strangler fig application and substantially revised the article in 2024. The software version, usually just

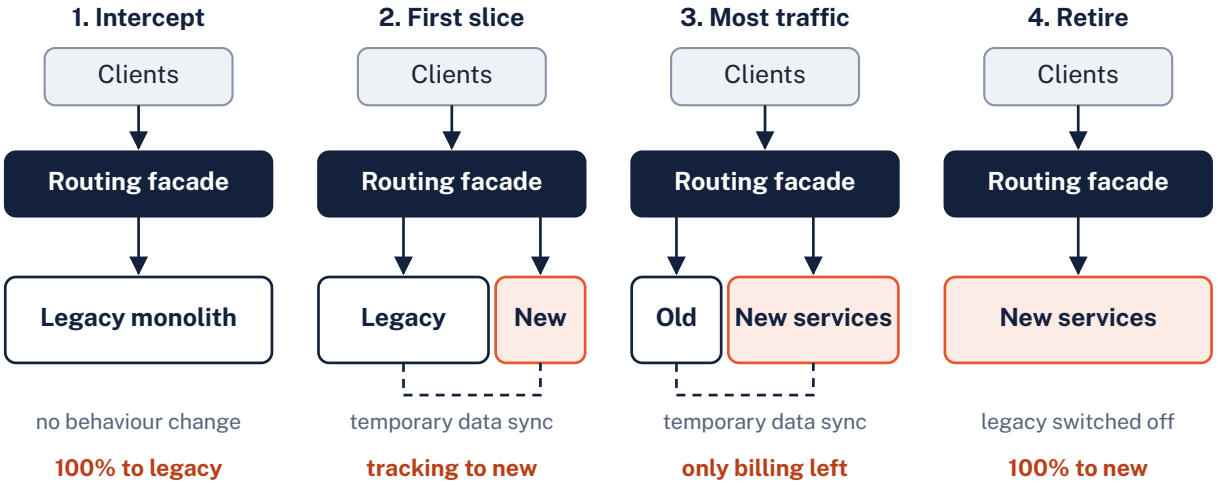
called the strangler fig, keeps the old system running while new components grow around it and take over its behaviour one piece at a time, until the old system does nothing and can be switched off.

The mechanism has three parts. First, an interception layer is placed in front of the legacy system: a reverse proxy, API gateway or routing facade through which all requests pass. Initially it sends everything to the old system and changes nothing. Second, a slice of behaviour is chosen along a seam (a place where behaviour can be redirected without editing much of the old code, in Feathers' terms), rebuilt as a new component and enabled by changing a routing rule, usually for a small share of traffic first. Third, the process repeats until the legacy system handles nothing, at which point it is retired. Throughout, the routing layer makes each step reversible: if the new component misbehaves, the rule is switched back.

The code that lets old and new coexist (the routing facade, adapters, data synchronisation jobs) is transitional architecture: software built to be thrown away. Teams often resist writing it because it feels wasteful. Fowler's argument, and the experience behind it, is that this cost buys reduced risk and much earlier value, which is almost always a good trade. The real danger is the opposite: a migration that stalls half way, leaving both systems and all the transitional code in production indefinitely. Each slice should therefore remove something from the old system, and progress should be measured by what has been switched off, not only by what has been built.

Ian Cartwright, Rob Horn and James Lewis of Thoughtworks have documented a family of related techniques as *Patterns of Legacy Displacement* on martinfowler.com (from 2021). They frame the work as four activities that are not a fixed sequence: understand the outcomes you want, break the problem into parts, deliver those parts incrementally, and change the organisation so that the new system does not decay the way the old one did. They also question the instinct to reproduce every feature: much of a legacy system's behaviour may no longer be wanted, and finding out is cheaper than rebuilding it.

A strangler fig migration in four stages



A routing facade lets traffic move slice by slice from the legacy system to new services, with temporary data synchronisation, until the legacy system can be switched off.

Branch by abstraction

The strangler fig works at the edge of a system, where requests can be routed. Some replacements must happen deep inside one codebase: swapping a persistence framework, a pricing engine or a home-grown messaging layer that hundreds of call sites use. Branch by abstraction, a name Paul Hammant introduced while advocating trunk-based development and that Martin Fowler described in 2014, does this without a long-lived version-control branch. The system stays releasable at every step, so the change can be spread over weeks or months while other work continues on the same main line.

Branch by abstraction, step by step

1. **Introduce an abstraction**
   Create an interface in front of the component to be replaced, with an implementation that simply delegates to the existing code. Nothing changes in behaviour.

2. **Move callers to the abstraction**
   Change call sites, gradually, to use the interface instead of the old component directly. Each change is small and released normally.

3. **Build the new implementation**
   Write the replacement behind the same interface, testing it against the same tests and, ideally, against production behaviour in a parallel run.

4 **Switch over gradually**
Use a feature flag or configuration to send some callers or some customers to the new implementation, increasing the share as confidence grows.

5 **Remove the old implementation**
Once nothing uses the old code, delete it, and remove the abstraction too if it no longer earns its place.

Proving equivalence: parallel run and dark launching

The hardest question in any migration is whether the new implementation really behaves like the old one. A parallel run answers it with production evidence: both implementations receive the same inputs, the old one remains authoritative and its answer is returned to the user, and the two results are compared and any differences recorded. GitHub published its Scientist library in 2016 for exactly this purpose, running a control and a candidate side by side and reporting mismatches; similar tools exist for many languages, and the technique is simple to write by hand, as the sketch below shows.

Dark launching, which Fowler describes as calling new back-end behaviour in production without users seeing its effects, is closely related. It is mainly used to test the new path's performance and stability under real load before any user depends on it, while a parallel run concentrates on correctness. Both differ from a canary release, in which a small share of users really do receive the new behaviour.

Both techniques have sharp edges. The candidate must have no externally visible side effects: it must not send a second email, charge a card twice or write to the shared database, so parallel runs are easiest for reads and calculations, and writes need a sandbox or a separate store. Comparisons must ignore legitimate differences such as timestamps, ordering and floating-point formatting, or the mismatch report drowns in noise. And running both paths costs capacity, so the share of traffic sampled is often limited.

```
def quote_delivery_fee(order):
    old = legacy_pricing.quote(order)          # still authoritative
    try:
        new = new_pricing.quote(order)         # must have no side effects
        if new.amount_pence ≠ old.amount_pence:
            metrics.increment("pricing.mismatch")
            log.info("pricing mismatch", order_id=order.id,
                     old=old.amount_pence, new=new.amount_pence)
        else:
            metrics.increment("pricing.match")
    except Exception:
        metrics.increment("pricing.candidate_error")   # never hurts the user
    return old
```

A minimal parallel run: the legacy result is still returned, while the candidate's answer is compared and mismatches are counted.

Data: the hardest part

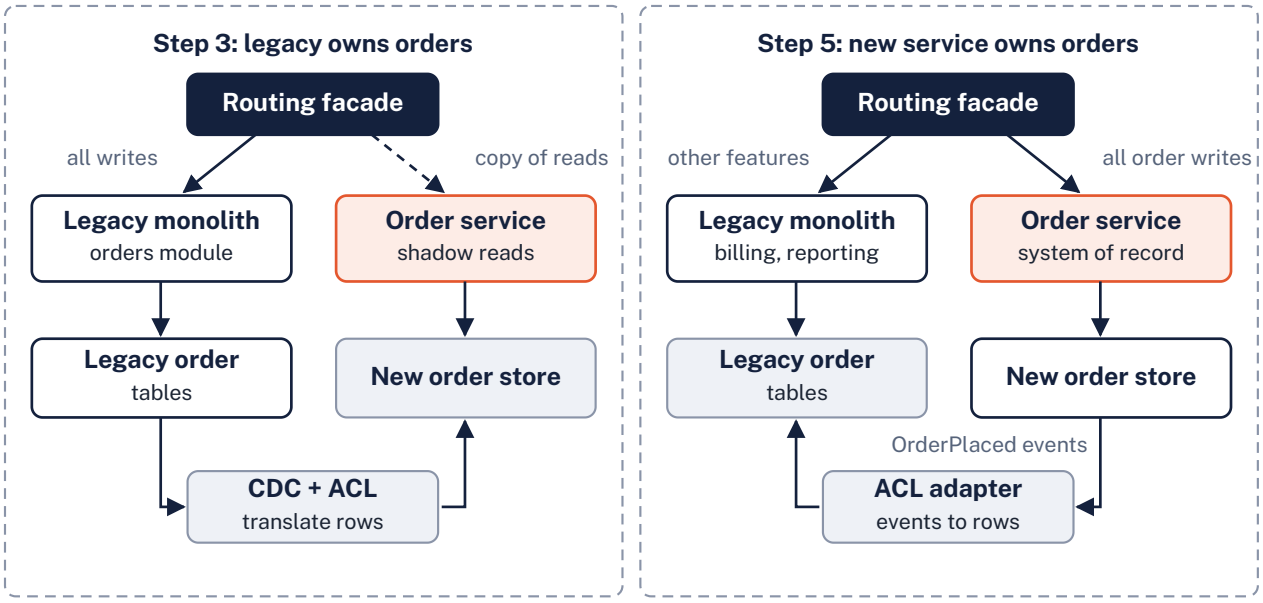
Routing requests is relatively easy; moving data and keeping it consistent while two systems share responsibility for it is where most migrations struggle. The first rule is the one from Chapter 7 on data ownership: for every kind of data, at every stage of the migration, the team must be able to say which system is the system of record, which copies exist and how stale each may be. The diagram below shows the typical flip: the legacy system owns the data at first and the new component reads a copy; later the new component owns it and the legacy system receives a copy for the modules that still need it.

Change data capture is the usual way to copy data out of a legacy database without changing the legacy code: a tool such as Debezium reads the database's transaction log and publishes each change as an event, the same mechanism Section 6.7 used to relay an outbox. Event interception, one of the legacy displacement patterns, works at the other end: it catches state-changing requests or messages on their way into the legacy system and forwards copies (or, later, the originals) to the new component. Either way, never let both systems accept writes to the same data independently with two-way synchronisation unless you have no alternative; conflicts are then almost guaranteed.

Legacy data rarely matches the new model. Status codes are integers with undocumented meanings, one table serves three purposes, and dates are stored in local time. An anti-corruption layer, the domain-driven design pattern for protecting a model from a neighbour's, translates between the two so that the legacy system's quirks do not leak into the new code. It belongs to the transitional architecture and should disappear with the legacy system.

The final data migration, moving history and cutting over, deserves its own plan. Profile the legacy data first to find duplicates, orphans and invalid values. Write the mapping explicitly and test it on a full, production-sized copy. Reconcile after every run with row counts, sums of money columns and checksums, not by spot checks. Rehearse the cutover, time it and write down the rollback. Where possible, migrate in cohorts (one region, one customer segment, one product line) rather than everything at once, so that a problem affects a fraction of customers and teaches you something before the next cohort.

The system of record flips during migration



Early on, change data capture copies legacy rows into the new store; once the new service owns orders, its events are translated back into legacy rows for modules not yet migrated.

IF YOU ARE NEW TO THIS

Step by step: why two systems must not both own the orders

The idea that needs the most care here is keeping one system of record for each kind of data while ownership moves from the old system to the new. Chapter 7 taught the rule; what is new here is that the owner changes while both systems still have users.

- Half way through, suppose both the legacy order module and the new order service accept changes to orders, with a job copying changes both ways every few seconds.
- A customer changes an order's delivery address in the app; the routing facade sends this to the new order service. Seconds later a support agent cancels the same order in the support tools, which still use the legacy module.
- If the job copies whole orders, whichever copy arrives last wins: the cancellation may be undone, or the address change lost. With orders changing all day, such clashes are almost guaranteed.
- The cure: at each stage, only the system of record accepts changes to orders, and the other holds a read-only copy. Early on the owner is the legacy module, and change data capture streams every change it makes across to the new service.
- Later, ownership is handed over: the new service accepts changes, and its events are translated back into legacy rows for modules not yet migrated. The handover is a planned step with a rollback ready, never a period when both are in charge; the worked example below does it city by city, so each order always has one owner.
- **The lesson:** for every kind of data, at every stage, name the one system that owns it and which way the copies flow. Letting both systems accept changes to the same data is a last resort.

Lessons from Kill It with Fire

Marianne Bellotti's *Kill It with Fire: Manage Aging Computer Systems (and Future Proof Modern Ones)* (No Starch Press, 2021) draws on her experience modernising government and commercial systems, including mainframe estates. Several of its themes are worth carrying into any modernisation.

Modernisation should be justified by the value it delivers and the risk it removes, not by the age of the technology; old technology that is well understood and works may be the least risky part of the estate. Technology choices move in cycles, so a team that replaces a system only because its design is out of fashion is likely to build the next generation's legacy. Incremental improvement of the existing system is usually safer than replacement, and a full rewrite needs an unusually strong justification. Finally, modernisation is an organisational problem as much

as a technical one: it needs visible early wins to keep momentum and morale, clear criteria for when it is finished, and attention to the incentives that made the old system decay, or the new one will follow it.

WORKED EXAMPLE

Worked example: strangling the order module out of the monolith in six steps

This example looks back at how the food-delivery platform got the separate order service evaluated in Section 11.1. Earlier in its life, the platform's original monolith contained an order module that handled placing, changing, cancelling and tracking orders. Order changes had become the main bottleneck: they needed the whole monolith to be deployed, and Friday-evening peaks slowed every other feature. Fourteen other modules, including billing, reporting and customer support tools, read the order tables directly. The team planned an incremental migration to a separate order service.

1. Set outcomes and find the seam	Outcomes: order changes deployable daily and independently; three times today's Friday peak without affecting other features; no lost or duplicated orders. Analysis of query logs confirms that only the order module writes the order tables and that all order traffic enters through the /api/orders endpoints. The team records several weeks of production requests and turns them into characterisation tests that capture current behaviour, quirks included.
2. Insert the routing facade	An API gateway is placed in front of the monolith and initially routes everything to it. Released on its own, it changes no behaviour; the team checks the added latency (a few milliseconds at the 95th percentile) and practises switching routes. Rollback: remove the gateway from the path.
3. Build the new service in the shadows	The new order service has its own store. Legacy remains the system of record: change data capture copies order rows across, through an anti-corruption layer that maps integer status codes and local-time dates to the new model. Read endpoints run as a parallel run against legacy responses; the mismatch rate starts at about 2% (rounding of fees, time zones) and is driven below 0.01% before going further.
4. Move reads	Order tracking reads are routed to the new service for 5%, then 25%, then all of traffic, watching error rates and latency at each stage. Rollback: one routing rule.
5. Move writes, one city at a time	The new service becomes the system of record for orders placed in one pilot city, then for every city over six weeks. It publishes OrderPlaced and OrderCancelled events through a transactional outbox; a transitional adapter translates them into legacy-format rows so that billing and reporting keep working unchanged. A daily reconciliation compares counts and money totals between the two stores. Those modules then move to the events or the order API over the following quarters.

6. Migrate history and retire	Three years of historical orders are copied in throttled batches and reconciled by counts and checksums. When database audit logs show no reads of the legacy order tables for 30 days, the team removes the adapter, the change data capture pipeline and the old routing rules, then deletes the order code and tables from the monolith. The migration is complete only at this point.

Every step delivers something usable or removes risk, and every step except the last has a cheap rollback.

The system of record for orders is unambiguous at every stage, and data flows in one direction at a time.

The transitional architecture is planned, measured and deliberately removed; success is measured by what was switched off.

WATCH OUT

Common mistakes

- Starting with the hardest, most central part of the system instead of a slice that delivers value and teaches the team how to migrate.
- Aiming for full feature parity without asking which old features are still used.
- Two-way synchronisation in which both systems accept writes to the same data.
- Migrations that stall half way, leaving two systems and all the transitional code running for years.
- Treating the data migration as a final task rather than designing it, rehearsing it and reconciling it from the start.
- Freezing all change to the legacy system for the duration, which turns the migration into a long wait for the business.

CHECK YOUR UNDERSTANDING

▼ **Why does a big-bang rewrite tend to lose behaviour that users depend on?**

Because years of bug fixes and business rules live only in the old code and in users' habits; nobody can list them all, so a rewrite from a specification misses some of them.

▼ **What does the routing facade contribute to a strangler fig migration?**

It lets traffic be moved slice by slice, and moved back if something goes wrong, without editing the legacy system for each step.

▼ **What is transitional architecture, and why track its removal?**

Temporary code such as routing rules, adapters and synchronisation jobs that lets old and new coexist; tracking its removal guards against a migration that stalls with both systems running indefinitely.

▼ **When might repurchase be a better choice than refactoring?**

When the capability is generic rather than differentiating and a good product exists, so rebuilding it would spend effort where the business does not compete.

▼ **Try it: A 15-year-old box-office system still sells tickets but changes are slow and risky. The board favours a two-year rewrite with a single cutover. What would you recommend, and how would you handle the data during migration?**

Avoid the big bang: value comes late, requirements move, and hidden behaviour is lost. Use a strangler fig, routing traffic through a facade and moving one slice at a time, starting with something valuable but low risk such as seat browsing. Use parallel runs to compare results while the old system remains authoritative. Each data set needs one system of record at every stage, kept in step by change data capture. Finish by removing the old system.

SOURCES

- Martin Fowler: Strangler Fig Application: https://martinfowler.com/bliki/StranglerFigApplication.html
- Joel Spolsky: Things You Should Never Do, Part I (2000): https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/
- Cartwright, Horn and Lewis: Patterns of Legacy Displacement: https://martinfowler.com/articles/patterns-legacy-displacement/
- AWS Prescriptive Guidance: migration strategies (the 7 Rs): https://docs.aws.amazon.com/prescriptive-guidance/latest/large-migration-guide/migration-strategies.html
- Martin Fowler: Branch By Abstraction (2014): https://martinfowler.com/bliki/BranchByAbstraction.html

SECTION 11.7 OF 11.8 · ADVANCED · 28 MIN READ · LAST REVIEWED NOVEMBER 2026

Leadership, influence and decision-making

BEFORE YOU READ THIS

This section builds on:

- 1.4 The architect's role
- 1.6 Decisions and trade-offs
- 4.8 Communicating

Terms it uses from earlier: Architecture Decision Record (ADR), Stakeholder, Power/interest grid, Architecture kata

THE IDEA IN BRIEF

Think of a mountain guide leading a group of experienced climbers. She cannot order anyone up the mountain. People follow her because she knows the route, explains the risks honestly, listens when someone spots danger, admits when she is unsure and has been right before. When the group disagrees about the route, she makes sure everyone is heard, agrees how the choice will be made, and then helps everyone commit to it. Most architects are in the same position: responsible for the route, but without the power to command, so they lead through trust, clear reasoning and good decisions made together.

WHAT YOU NEED TO KNOW

- Architects usually carry responsibility without formal authority, so their influence rests on expertise, information and relationships rather than on positional power (French and Raven, 1959).
- Trust grows with credibility, reliability and intimacy and shrinks with self-orientation (Maister, Green and Galford, 2000); psychological safety lets teams raise problems while they are cheap to fix.
- Settle decision rights and the decision method before debating substance; consent adopts a proposal unless a reasoned objection remains, while consensus requires everyone's support.
- Principled negotiation separates people from problems, focuses on interests rather than positions and agrees objective criteria; choose a conflict mode deliberately and escalate together.
- Hohpe's architect elevator connects the penthouse (strategy) with the engine room (technical reality), carrying risks and options up and strategy and reasons down.

- Growing others through mentoring, coaching, sponsoring and delegation, and supporting staff-plus career paths (Larson, 2021; Reilly, 2022), prevents the architect becoming a bottleneck.
- Groups add their own decision traps, such as deference to seniority, escalation of commitment, bikeshedding and decision by attrition, each with a practical countermeasure.

Leading without authority

Most architects do not manage the people who build what they design. Teams report to engineering managers, priorities belong to product managers, budgets to directors. Yet the architect is expected to make sure significant decisions are made well and kept true across many teams, which is the core responsibility described in Section 1.4. The gap between responsibility and formal authority is the defining condition of the job, and closing it is a matter of influence.

John French and Bertram Raven's classic study of social power (1959) distinguished bases of power that come with a position (legitimate authority, the ability to reward and the ability to punish) from those that belong to a person (expertise and referent power, the respect and identification others feel); Raven later added informational power, the ability to bring facts that change minds. Architects rarely hold much positional power, and when they do, using it is costly: a decision imposed by rank is complied with, not owned, and the people who disagreed stop telling you about problems. Durable architectural influence rests almost entirely on expertise, information and relationships.

This is not a soft skill bolted on to technical work. A sound design that nobody adopts has the same effect as no design at all, and a mediocre design that teams understand and support often outperforms a brilliant one they work around. The rest of this section treats influence as a set of practices that can be learned: building trust, facilitating decisions, handling conflict, communicating upwards, growing other people and avoiding the traps that groups fall into.

Building trust and credibility

David Maister, Charles Green and Robert Galford's *The Trusted Advisor* (2000) offers a useful model. Trust grows with credibility (you know what you are talking about), reliability (you do what you say), and intimacy (people feel safe raising difficult things with you), and it shrinks with self-orientation (the sense that you are serving your own interests, such as being right, looking clever or protecting your design). The model's insight for architects is that self-orientation sits below the line: it divides everything else. An architect who is brilliant and dependable but visibly needs to win every argument is not trusted.

In practice, credibility is built in small, visible ways. Be right about small things before asking to be trusted on big ones. Stay close enough to the code to understand its real cost: write the spike, review the pull request, sit in on an incident review. Say what you do not know and how you would find out. Follow through on every commitment, especially the dull ones. Give credit publicly and take responsibility for your own mistakes, also publicly. Explain the reasoning behind every principle so that people can apply it in situations you did not foresee.

Trust also depends on the climate around you. Amy Edmondson's research on psychological safety, beginning with a 1999 study of work teams, found that teams in which members believe it is safe to admit mistakes and challenge others learn more and perform better. Architects shape that climate more than they realise. Reacting to bad news with curiosity rather than blame, inviting objections explicitly (what would make this fail?) and thanking people who raise them are what make it possible to hear about a flawed design while it is still cheap to fix.

Facilitating decisions

Many architecture arguments are really arguments about process: who gets to decide, who must be asked and how disagreement will be settled. Settling decision rights before discussing the substance removes much of the heat. Section 11.4 described the advice process, in which anyone may make a decision provided they seek advice from those affected and those with expertise; other organisations use an architecture review forum, a named decision owner per domain, or role charts such as RACI. The specific mechanism matters less than its being known in advance.

Within whatever decision rights apply, the architect is often the facilitator: the person who frames the question, makes sure the right people and evidence are present and steers the group to a decision that sticks. Different decision methods suit different situations, and choosing the method deliberately is part of the job.

Two methods are often confused. Consensus seeks a proposal that everyone actively supports; it builds strong commitment but is slow and gives any one person an effective veto. Consent decision-making, developed in Gerard Endenburg's sociocracy in the Netherlands from the 1970s, adopts a proposal unless someone raises a reasoned objection that it would cause harm or move the group backwards. The question changes from do you all agree? to can anyone not live with this, and why? A phrase from sociocratic practice captures the threshold well: good enough for now, safe enough to try. Consent works particularly well for two-way-door decisions, which can be revisited as evidence arrives. The IETF's rough consensus, described in Section 4.8, is a close relative.

When agreement cannot be reached, someone must decide, and it should already be clear who. Disagree and commit, also discussed in Section 4.8, asks people to argue their case fully before the decision and to support it fully afterwards. It works only if the disagreement was genuinely heard, the dissent is recorded (in the architecture decision record, with its reasons) and there is an agreed signal that would reopen the question.

Choosing a decision method

Method	How it works	Use when	Main risk
Decide alone	The accountable person decides, perhaps after a quick check	Two-way doors, emergencies, decisions of narrow scope	Missed knowledge; low commitment from others
Consult, then decide (advice process)	The decider must seek advice from affected people and experts, then decides and records why	Most architecture decisions in autonomous teams	Advice sought for form's sake and ignored
Consent	Adopt unless a reasoned, paramount objection remains; objections improve the proposal	Decisions that affect a group and can be revisited	Objections misused as vetoes on preference
Consensus	Continue until everyone supports the proposal	Small groups, high-stakes values or principles that need deep commitment	Slow; stalemate; decisions made by default
Vote	Majority or weighted vote among participants	Choosing among options that are all acceptable	A losing minority with no commitment; popularity over evidence
Delegate	Hand the decision, with constraints, to a team or person closer to the work	The decision is local and the constraints can be stated	Delegating without the context or the constraints

IF YOU ARE NEW TO THIS

Step by step: consent is not consensus

The step most people trip over here is the difference between consensus, where everyone must actively support a proposal, and consent, where it is adopted unless someone raises a reasoned objection. Follow one small decision through both.

- After the risk-storming session in Section 11.2, the six people who took part must agree when to look at the design again. The architect proposes every three months.
- By consensus: one developer wants monthly reviews, the operations engineer only before major changes. Discussion continues until all six actively support one answer, so any one of them can hold it up. It can drag on until it stalls, or until the most persistent voice wins.
- By consent: the facilitator asks, 'Can anyone not live with every three months, and why?' The developer still prefers monthly but can live with quarterly. A preference is not an objection, so it does not block the proposal.
- The security engineer objects, with a reason: a large change to the payment path is due next month, and a three-month gap would leave it unreviewed while it is newest and riskiest. That is a reasoned claim of harm, so the proposal is improved: every three months, plus a review after the payment change.
- Nobody objects to the improved proposal, so it is adopted. It need not be perfect: good enough for now, safe enough to try, and easy to revisit later.
- **The lesson:** consent asks 'Can anyone not live with this, and why?' rather than 'Do we all agree?' It suits decisions that can be revisited. Keep consensus for the few that need everyone's deep commitment, and agree the method before the debate starts.

Conflict and organisational politics

Organisational politics is simply how groups with different interests, limited resources and their own careers reach decisions. It is not a corruption of architecture work but its environment. Ignoring it does not make an architect principled, only less effective. The tools from earlier chapters apply directly: the stakeholder map and power/interest grid from Section 1.5 show who is affected and who can block, and the trade-off records from Section 1.6 make positions visible.

Roger Fisher and William Ury's *Getting to Yes* (1981) remains the most useful guide to principled negotiation, and its four ideas translate directly. Separate the people from the problem: the payments lead is not obstructive, she is worried about card data. Focus on interests, not positions: we must not use Kafka is a position, we cannot take on more on-call load is an interest that several designs could meet. Invent options for mutual gain before deciding. Insist

on objective criteria, agreed before options are compared. Their idea of the best alternative to a negotiated agreement (BATNA) is useful too: knowing what happens if no agreement is reached tells everyone how much the agreement is worth.

Kenneth Thomas and Ralph Kilmann's conflict-mode model (1974) describes five ways of responding to conflict, according to how much weight you give to your own concerns and to the other party's. None is always right; the skill is choosing deliberately rather than falling back on habit.

Some practical habits help in every case. Never surprise a stakeholder in a public meeting with a proposal that affects them; talk to them first, one to one (Japanese organisations call this preparatory consultation nemawashi). When escalation is needed, escalate together: both parties present the disagreement and the options to the person who can decide, rather than each lobbying separately. And keep disagreements about designs separate from judgements about people, in what you say and in what you write.

Five conflict modes (after Thomas and Kilmann)

Mode	What it means	Useful when	Overused, it leads to
Competing	Pursue your own position firmly	Safety, security or legal issues where you must hold the line	Compliance without commitment; people stop raising problems
Collaborating	Work together to meet both sides' interests fully	Important decisions where both sides' concerns are legitimate	Slow decisions on matters that did not need them
Compromising	Each side gives up part of what it wants	Time is short and both sides have equal power	Designs that satisfy nobody and are incoherent
Avoiding	Postpone or sidestep the issue	The issue is trivial or emotions need time to cool	Problems that grow until they are crises
Accommodating	Yield to the other side's position	You were wrong, or the issue matters far more to them	Lost credibility and abandoned principles

Communicating with executives

Section 4.8 covered tailoring a message to its audience and leading with the answer. Executives deserve a little more attention, because they hold the budgets and the authority that architecture work depends on, and because architects most often lose them by explaining too much and asking too little.

Frame every topic in the terms executives already manage: revenue, cost, risk, time to market, customer experience, regulation and people. Bring options rather than a single plan, each with its cost, its risk and what it gives up, and say which you recommend and why. Quantify where you honestly can, and say how confident the numbers are; a range with a stated basis is more credible than false precision. Be explicit about the decision or support you need from them and by when. Prepare for the question what would you do if this were your money? And never let an executive learn of a serious risk first from someone else.

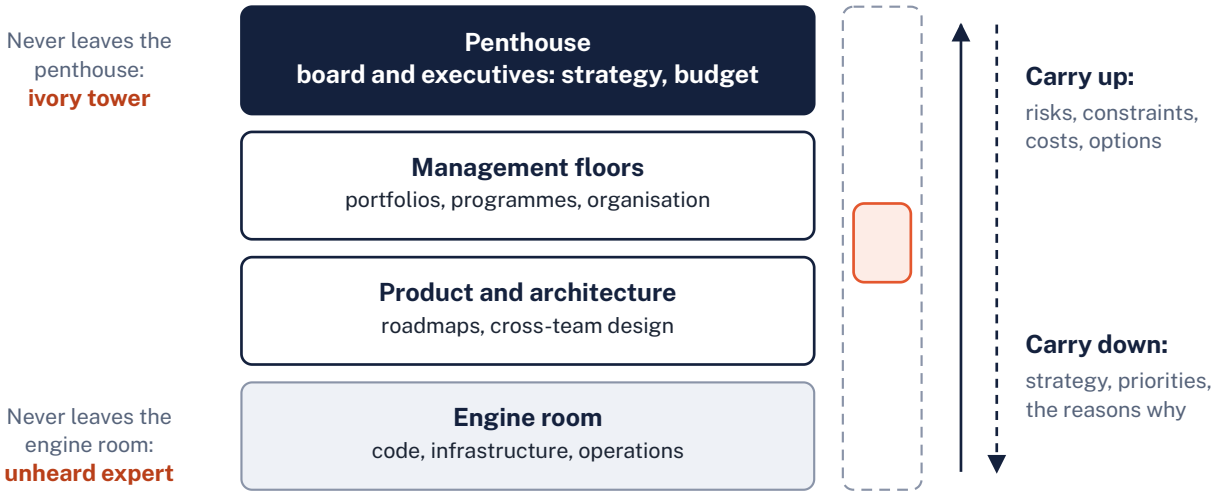
Senior people are often less interested in how a system works than in whether its owners are in control of it. Showing that risks are known, owned, measured and being reduced (for example through an architecture risk register or the fitness functions of Section 11.3) builds more confidence than any diagram. Section 11.8 shows how to put architecture work into the form executives use to compare investments: the business case.

The architect elevator

Gregor Hohpe's *The Software Architect Elevator* (O'Reilly, 2020) gives the most memorable picture of this work. Large organisations are like tall buildings. In the penthouse, executives set strategy and budgets; in the engine room, engineers build and run the systems; many floors of management sit in between, and information passing through them is filtered and distorted on the way. Hohpe's architect elevator rides between the penthouse and the engine room, carrying information in both directions: taking strategy and its reasons down to the people building systems, and taking technical realities (constraints, risks, costs and options the business did not know it had) up to the people making strategic decisions.

Hohpe's point is that the architect's value lies in connecting the floors. An architect who stays in the penthouse becomes an ivory-tower strategist whose plans ignore what is technically possible; one who stays in the engine room becomes a respected expert whose warnings never reach the people who could act on them. Riding the elevator well means changing language and level of detail at each floor without changing the substance, which is exactly the audience skill of Section 4.8 applied to a whole organisation.

Riding the elevator



After Gregor Hohpe: the architect connects strategy in the penthouse with technical reality in the engine room, translating in both directions.

Growing other architects

An architect who makes every significant decision personally becomes a bottleneck, and the organisation's architectural capability leaves with them. Growing others is therefore part of the role, not an optional extra. It helps to distinguish three activities. Mentoring shares your experience and advice. Coaching asks questions that help someone reach their own answer, which builds judgement faster. Sponsoring uses your credibility to give someone opportunities: recommending them to lead a design, putting their name on the proposal, inviting them to the executive meeting.

Practical techniques include pairing on designs and reviews, letting less experienced engineers write the ADR for a decision and reviewing it with them, running architecture katas (Section 11.2) as practice, delegating decisions within stated guardrails, and narrating your reasoning aloud during design discussions so that others can see how trade-offs are weighed rather than only the conclusion.

Many organisations now offer a technical career path parallel to management. Will Larson's *Staff Engineer: Leadership beyond the management track* (2021) popularised the term staff-plus for staff, principal and more senior individual contributors, and described four common archetypes from interviews with people in those roles. Tanya Reilly's *The Staff Engineer's Path* (O'Reilly, 2022) organises the work around big-picture thinking, executing large projects and levelling up the engineers around you. In her 2019 talk "Being Glue", Reilly also drew attention

to the coordinating work (unblocking people, writing the missing document, noticing the gap between teams) that keeps projects alive but is often not rewarded in promotion; architects should notice who does it and make sure it counts.

Staff-plus archetypes (after Will Larson)

Archetype	Focus	Typical architectural contribution
Tech lead	Guides the approach and execution of one team, in partnership with its manager	Owns the team's architecture decisions and keeps them aligned with the wider system
Architect	Responsible for the direction, quality and approach of a critical area	Sets principles, reviews designs and stewards key qualities across several teams
Solver	Digs into hard, ambiguous problems and finds a way forward, often moving between them	Resolves the stuck migration, the performance crisis or the unclear boundary
Right hand	Extends an executive's attention, borrowing their scope and authority	Drives organisation-wide technical initiatives and connects strategy with execution

Decision traps revisited

Section 1.6 listed the individual biases that distort architectural decisions: anchoring, sunk cost, groupthink, résumé-driven development, hype and confirmation bias. Leading decisions among groups adds traps of its own, which come from status, fatigue and organisational momentum rather than from individual reasoning.

Group and organisational traps

Trap	What it looks like	Countermeasure
Deference to the highest-paid opinion	The most senior person speaks first and the discussion converges on their view	Ask the most senior person to speak last; collect written views before the meeting
Escalation of commitment	A failing initiative receives more investment because so much has already been committed to it (Barry Staw, 1976)	Set kill criteria and review dates at the start; review with people who were not involved
Bikeshedding	Long debate about trivial, easily understood details while the hard decision gets minutes (Parkinson's law of triviality, 1957)	Timebox by significance; push two-way-door details to the team that owns them
Decision by attrition	The option backed by whoever has most stamina wins once everyone else is tired	Agree the decision method, the decider and the deadline in advance
False consensus	Silence is taken as agreement; objections surface after the decision	Ask explicitly for objections, by name if necessary, and record them
Not invented here	External solutions are rejected because they were not built internally	Compare build and buy against the same criteria, including the cost of running what you build
Analysis paralysis	More information is always requested; nothing is decided by the last responsible moment	Name the decision date and what evidence would be enough; make reversible decisions quickly

WORKED EXAMPLE

Worked example: navigating a contested decision about an event bus

This example goes back to the decision that introduced the food-delivery platform's event broker, before the architecture evaluated in Section 11.1 existed. The platform team proposes that all services communicate through a company-wide managed Kafka cluster. The ordering team is enthusiastic, the payments team objects strongly, finance is worried about cost, and the CTO wants a decision within a month. Under the company's advice process the platform team's lead architect is the decision maker. A principal architect is asked to facilitate.

1. Reframe the question	Kafka for everything? is a position. The underlying problem, from two Friday outages, is that order placement depends on synchronous calls to notifications and dispatch. The question becomes: how should services publish business events, starting with order events, so that consumers no longer sit in the order path? Scope and decision rights are written at the top of a draft ADR.
2. Listen one to one first	Before any meeting, the facilitator talks to each party. Payments: card data must stay out of new systems, and the team cannot take on more on-call work after a painful incident with a self-hosted broker. Ordering: needs decoupling for peaks. Finance: wants run costs capped. Security: needs encryption and access control per topic. The platform team: wants one supported standard rather than five. Each is recorded as an interest, not a position.
3. Agree criteria before options	At the architecture advisory forum the group agrees weighted criteria: delivery guarantees and per-order ordering (5), operational burden on product teams (5), compliance scope (4), run cost at three times today's volume (3), skills (3) and reversibility (3).
4. Generate options, including the dissenters'	A: managed Kafka, mandatory for all services. B: managed Kafka for order-domain events through a transactional outbox, other teams opt in later. C: the cloud provider's simpler publish-subscribe and queue services. D: keep synchronous calls and add retries and circuit breakers. Payments contributes option D and the cost analysis for C.
5. Gather evidence	A two-week spike runs B and C with production-like load: end-to-end latency, behaviour during broker failover and estimated monthly cost. Security confirms that order events can be designed to contain no card data, which keeps them out of payment-card scope.

6. Consent round	Proposal: B, with the platform team operating the cluster as X-as-a-Service. Payments objects that it may still be pulled into broker operations; the objection is resolved by writing the platform team's ownership and support hours into its team API. Finance's concern becomes a cost alert and a review threshold. One ordering engineer still prefers C; she can live with B, and her reasoning is recorded.
7. Decide, record and communicate	The decision maker accepts B. The ADR records options, scores, the spike results, the dissent and the review triggers (more than three teams adopting, or costs above the agreed threshold, or six months, whichever comes first). The CTO and CFO receive a one-page summary of benefit, cost and risk; teams get a walkthrough; the technology radar entry is updated to Trial.
8. Follow through	A schema-compatibility check runs in every producer's pipeline, the outage scenario is re-tested, and at the six-month review the evidence is used to decide whether to widen, keep or reverse the decision.

The facilitator never argued for an option; the work went into framing, interests, criteria, evidence and a clear decision method.

Dissent changed the outcome (scope narrowed, platform ownership made explicit) and was recorded rather than overruled silently.

The decision was made reversible in practice through limited scope and agreed review triggers.

WATCH OUT

Anti-patterns in architectural leadership

- Winning arguments by seniority, which ends the flow of bad news.
- Deciding in private and announcing in public, so that consultation is seen to be theatre.
- Treating every disagreement as resistance to be managed rather than information to be used.
- Promising executives certainty that the engineering work cannot support.
- Hoarding the interesting decisions, so that nobody else learns to make them.
- Staying neutral on safety, security or ethical issues where the architect should hold a firm line.

CHECK YOUR UNDERSTANDING

▼ Why talk to stakeholders one to one before a decision meeting?

To understand their interests privately, avoid surprising them in public and surface objections early, when they can still shape the proposal.

▼ How do mentoring, coaching and sponsoring differ?

Mentoring shares your experience and advice, coaching asks questions that help someone reach their own answer, and sponsoring uses your credibility to create opportunities for them.

▼ What does an architect lose by never leaving the engine room?

The ability to influence strategy: their knowledge of risks, constraints and options never reaches the people making investment and organisational decisions.

▼ Why should dissent be recorded in the decision record?

So that the reasoning is preserved, the dissenters know they were heard and, if their concerns come true, the organisation knows what to revisit and why.

▼ Try it: The platform team wants one shared Kafka cluster; Payments objects strongly; you have no authority over either team, and the CTO wants a decision within a month. How would you handle this, without forcing an outcome?

First settle who decides and how, say consent with the CTO as tie-breaker. Then separate positions from interests: Payments may fear audit scope or lost control, and the platform team wants less to run. Agree objective criteria, such as cost, latency and compliance, and test them with a small spike. Options might include a separate cluster for Payments. Record dissent in the ADR. If stuck, escalate together with a shared statement of options, not as rivals.

SOURCES

- Gregor Hohpe: The Architect Elevator: https://architectelevator.com/
- Will Larson: Staff archetypes: https://staffeng.com/guides/staff-archetypes/
- Andrew Harmel-Law: Scaling the Practice of Architecture, Conversationally: https://martinfowler.com/articles/scaling-architecture-conversationally.html
- RFC 7282: On Consensus and Humming in the IETF: https://www.rfc-editor.org/rfc/rfc7282

SECTION 11.8 OF 11.8 · ADVANCED · 31 MIN READ · LAST REVIEWED NOVEMBER 2026

Ethics, sustainability and the business case

BEFORE YOU READ THIS

This section builds on:

- 1.7 Technical debt
- 2.7 Security, usability and more
- 8.7 Privacy by design

Terms it uses from earlier: Technical debt, Accessibility, Threat modelling, Legacy system

THE IDEA IN BRIEF

Before paying for a new roof, a sensible homeowner asks what it costs, how much it will save in repairs and heating, how soon it pays for itself and what happens if they wait another winter. A good builder answers those questions honestly. A good builder also refuses to fit wiring that could start a fire, even if the client wants it cheaper, and suggests insulation that cuts energy bills. Architects need both habits: making an honest financial case for the work they propose, and taking responsibility for the harm or waste their designs could cause.

WHAT YOU NEED TO KNOW

- A business case compares options, including doing nothing, by their costs, benefits and risks in money, time and probability, and recommends one; staged funding turns a large bet into smaller options.
- Net present value discounts each year's cash flow: NPV = $\Sigma\ C_t \div (1 + r)^t$; a positive NPV means the investment beats the discount rate, which finance normally sets.
- The payback period is the time until cumulative cash flow turns positive; present it alongside NPV and a sensitivity analysis of the estimates that matter most.
- Technical debt becomes a business case when interest is expressed as annual cost (time, delay, incidents, risk) and principal as the investment; tool remediation figures measure principal, not interest.
- The ACM Code of Ethics (2018) and the BCS Code of Conduct require architects to consider the public interest, avoid harm, evaluate risks thoroughly and be honest about them.
- Foreseeable harms include privacy, security, bias, inaccessibility, safety and manipulative dark patterns, several of which need architectural support and are now regulated.

- Green software practice covers carbon and energy efficiency, carbon awareness, hardware efficiency, measurement and real reductions; the SCI specification (ISO/IEC 21031:2024) measures emissions per functional unit.

Why architects need business cases

Architecture work competes for money and people with everything else an organisation wants to do. A migration, a platform, a resilience programme or a debt repayment rarely produces a feature a customer can see, so it is easy to postpone in favour of work that does. The architects who get such work funded are usually not the ones with the most elegant designs but the ones who can express the case in the terms decision makers use to compare investments: what it costs, what it returns, how soon, how certain, and what happens if we do not do it. That structured argument is a business case.

This is the business-facing side of everything this course has said about trade-offs. A business case is a decision record for an investment: context, options, criteria, consequences and a recommendation, with the consequences expressed as far as possible in money, time and risk. It does not replace architectural judgement, and much of what architecture protects (security, learning, optionality) is hard to price. But putting honest numbers where they can be put, and saying clearly where they cannot, earns the right to be heard on the rest.

Organisations use different formats. The UK government's Green Book, published by HM Treasury, structures public-sector cases around five questions: the strategic case (why change is needed), the economic case (which option gives best value), the commercial case (can it be procured), the financial case (is it affordable) and the management case (can it be delivered). Most companies use something lighter, but the same questions are worth answering for any significant architecture investment.

Options, costs, benefits and risks

Every business case should compare at least three options: doing nothing (the baseline), doing the minimum, and the recommended option, plus any serious alternatives. Doing nothing is rarely free. The legacy system's support contract rises, the incident rate grows with traffic, and changes keep getting slower; that drift is the baseline against which the other options' benefits are measured, and leaving it out makes every investment look worse than it is.

Costs and benefits should be listed by category so that none is forgotten and none is counted twice. Risks belong in the case explicitly: both the risks the investment reduces, which can be valued as expected losses avoided (probability multiplied by impact), and the risks of the investment itself, such as overrun or disruption during migration. Don Reinertsen's cost of

delay, from *The Principles of Product Development Flow* (2009), is a useful extra lens: the value lost each week a piece of work is not delivered. Dividing cost of delay by duration (weighted shortest job first) gives a defensible way to order competing pieces of work.

Finally, structure the investment as a series of stages with decision points rather than one large commitment. The incremental migration in Section 11.6 is a good example: each step delivers something and generates evidence, so funding can be released step by step and the programme stopped if the benefits do not appear. Stage gates of this kind turn a risky bet into a set of smaller options, which is the reversibility thinking of Section 1.6 applied to money.

What goes into an architecture business case

Category	Typical items	How to estimate
One-off costs	Engineering time, transitional architecture, data migration, licences, training, parallel running, decommissioning	Team size multiplied by duration and loaded day rate; vendor quotes
Ongoing costs	Infrastructure, licences and subscriptions, support, on-call, new skills to retain	Current bills and capacity models; provider pricing at forecast volume
Efficiency benefits	Less effort per change, shorter lead time, less manual work and fewer hand-offs	Change volume multiplied by time saved per change, from delivery data
Risk reduction	Fewer or shorter incidents, avoided security breaches, avoided regulatory penalties	Expected loss: frequency multiplied by impact, before and after
Revenue and options	Features that become possible, markets that open, faster experiments	Cost of delay of the opportunities enabled; often shown separately as upside
Cost avoidance	End-of-support upgrades, hardware refresh, rising licence fees that no longer apply	Contracts and vendor road maps

IF YOU ARE NEW TO THIS

Step by step: why later money counts for less

This section's hardest idea is discounting: why money arriving in a few years counts for less than money today, and how net present value uses this to weigh a cost paid now against benefits that come later. Simple numbers prepare for the worked example below.

- The order-module migration from Section 11.6 costs about £420,000 now; its benefits arrive over the next four years. Simply adding up the pounds would treat a pound in year 4 as worth a pound today, and it is not.
- Money held today can be put to work, and a forecast benefit may never arrive. So finance sets a discount rate, here 10%, reflecting what money costs the organisation and the risk of the investment.
- At 10%, £100,000 today grows to £110,000 in a year and £121,000 in two. Turned round, £121,000 arriving in two years is worth £100,000 today.
- Discounting is that sum done backwards: divide a cash flow by 1.1 once for each year of waiting. Four divisions turn a £190,000 benefit in year 4 into only about £129,800 today.
- Net present value adds up every year's discounted cash flow, including the investment, which is paid now and so is not discounted. A positive total means the investment returns more than the discount rate. Since the cost comes first and the benefits later, discounting makes such a case look smaller than a simple total.
- **The lesson:** never add pounds from different years as if they were equal. Convert each year's cash flow to today's value at the rate finance sets, then add them up. The answer is only as good as the estimates, which is why the worked example also tests smaller benefits.

Net present value and payback

Money now is worth more than money later: it can be invested, and later money is less certain. To compare cash flows that arrive in different years, finance teams convert each into its value today, using a discount rate that reflects the organisation's cost of capital and the risk of the investment. The rate is normally set by the finance department, and architects should use it rather than choose their own.

The net present value of an investment is the sum of all its cash flows, each discounted to today: NPV = $\Sigma\ C_t \div (1 + r)^t$, summed over years t = 0 to n, where C_t is the net cash flow in year t (negative for costs) and r is the annual discount rate. The year-0 cash flow is the initial

investment and is not discounted. A positive NPV means the investment returns more than the discount rate; when choosing between options of similar risk, the one with the highest NPV creates the most value.

The payback period is simpler: how long until the cumulative cash flow turns positive. Executives like it because it is easy to understand and because it captures risk crudely (the sooner the money is back, the less can go wrong), but it ignores everything after the payback point. The discounted payback period does the same calculation with discounted cash flows. Present both NPV and payback, and show how sensitive they are to the estimates that matter most.

WORKED EXAMPLE

Worked example: the business case for strangling the order module

The food-delivery platform's architect builds the business case for the order-module migration described in Section 11.6. Finance sets the discount rate at 10%. Benefits are estimated from delivery and incident data; revenue from new features is left out and mentioned only as upside.

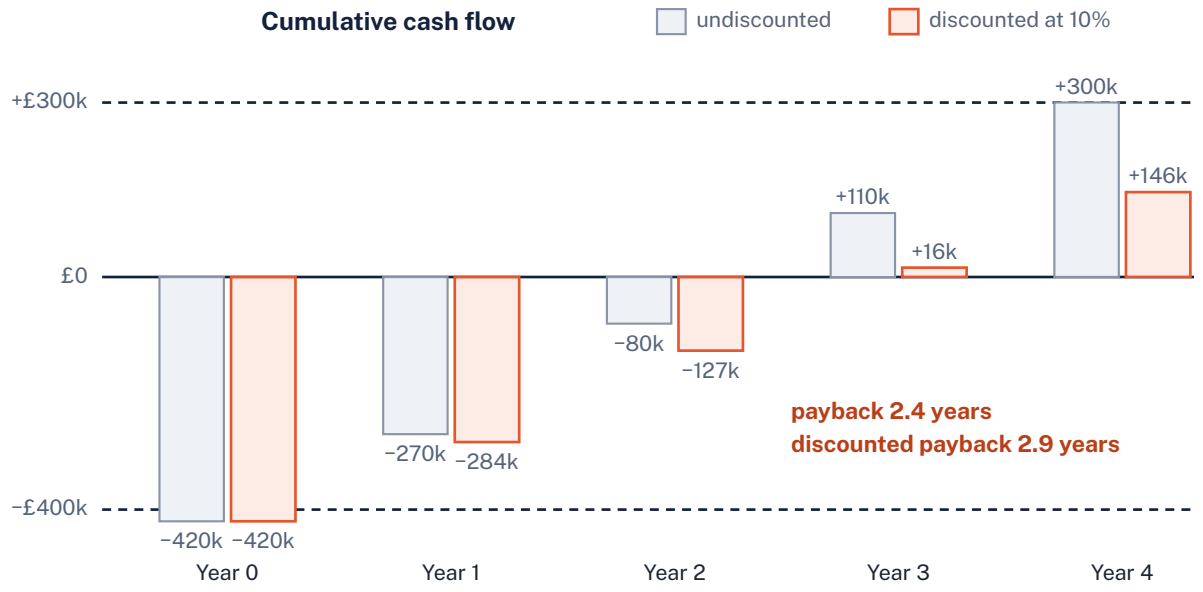
Investment (year 0)	Six engineers for about five months (about 630 engineer-days at the same loaded rate of £600 a day, £378,000), plus transitional infrastructure and parallel running (£42,000): £420,000.
Annual benefit from faster change (years 2 to 4)	About 100 order-related changes a year, each saving 2 developer-days of coordination, release waiting and rework, at a loaded rate of £600 a day: 100 × 2 × £600 = £120,000.
Annual benefit from fewer peak incidents	Six Friday-peak incidents a year, each costing about £20,000 in refunds, vouchers and lost orders; the migration is expected to remove two-thirds of them: 4 × £20,000 = £80,000.
Net extra running cost	New service and broker minus capacity released from the monolith: −£10,000 a year.
Net cash flows	Year 0: −£420,000. Year 1 (benefits ramp up during migration): +£150,000. Years 2, 3 and 4: +£190,000 each.
Discounted cash flows	$-420{,}000 + 150{,}000 \div 1.1 + 190{,}000 \div 1.1^2 + 190{,}000 \div 1.1^3 + 190{,}000 \div 1.1^4 =$ $-420{,}000 + 136{,}364 + 157{,}025 + 142{,}750 + 129{,}773$.
Net present value	NPV ≈ +£145,900 over four years. At a 15% discount rate it would still be about +£87,700.
Payback	Cumulative undiscounted cash flow: −420,000, −270,000, −80,000, +110,000. Payback = 2 + 80,000 ÷ 190,000 ≈ 2.4 years. With discounted flows the cumulative total turns positive in year 3: 2 + 126,612 ÷ 142,750 ≈ 2.9 years.
Sensitivity	If benefits are 30% lower than estimated, NPV falls to about −£23,900. The case is robust to the discount rate but sensitive to the benefit estimates.
Recommendation	Approve steps 1 to 4 of the migration (£250,000), measure change lead time and incidents for orders, and release funding for steps 5 and 6 only if the measured savings support at least 80% of the estimate.

The calculation is simple; the value lies in making the assumptions explicit, so that finance can challenge them and the team can measure them.

The sensitivity analysis shows where to spend effort: validating the developer-time saving early matters far more than refining the discount rate.

Staged funding turns a single large bet into a sequence of options, each released on evidence.

Payback at a glance



Cumulative cash flows turn positive during year 3; discounting delays payback and reduces the final value to an NPV of about £146,000.

```
def npv(rate, cash_flows):
    """cash_flows[0] is year 0 (the investment, usually negative)."""
    return sum(c / (1 + rate) ** t for t, c in enumerate(cash_flows))

flows = [-420_000, 150_000, 190_000, 190_000, 190_000]
print(round(npv(0.10, flows)))        # 145911
print(round(npv(0.15, flows)))        # 87663

pessimistic = [flows[0]] + [c * 0.7 for c in flows[1:]]
print(round(npv(0.10, pessimistic)))  # -23862
```

The same calculation in a few lines, convenient for testing different assumptions.

Technical debt in money and time

Section 1.7 explained technical debt as principal (the cost of fixing a problem) and interest (the extra cost paid on every change while it remains), and estimated the interest on one tangled module. A business case needs the same quantities in a form finance recognises. The interest

becomes an annual cost; the principal becomes the investment; and repaying the debt becomes an investment whose return is the interest no longer paid, plus any risk reduction and enabled opportunities.

Several kinds of evidence can be converted. Developer time: extra days per change, from comparisons with similar changes elsewhere or from developers' estimates, multiplied by change volume and a loaded day rate. Delay: longer lead times multiplied by the cost of delay of the affected work. Incidents: the number and duration of incidents traced to the debt, multiplied by their cost. Risk: the probability and impact of the failures the debt makes more likely, such as a security breach through an unsupported component. Opportunity: business initiatives the debt blocks outright.

Be careful with the debt figures produced by static analysis tools, which typically report an estimated remediation effort (how long it would take to fix every flagged issue). That is an estimate of principal, not of interest, and it says nothing about whether the code is ever changed. Debt in code that is rarely touched costs little; the hotspots of Section 1.7 are where interest is actually paid, and where repayment has the best return. Presenting debt as a portfolio of small investments, each with an estimated return and payback, is usually far more persuasive than a single headline figure for total debt.

Wardley mapping

Business cases compare options; a **Wardley map**, devised by Simon Wardley, helps you see which options exist. The map is anchored on a user and their need. The vertical axis is the value chain, with what the user can see at the top and what it depends on below. The horizontal axis is evolution, from genesis through custom built and product to commodity, and components drift rightwards as competition matures them. For the food-delivery platform, the need is getting a meal delivered. The ordering app sits high up; dispatch optimisation is custom built and differentiating; card payments are a product; compute is a commodity. Hand-building anything far to the right is a warning sign. The GOV.UK Service Manual suggests mapping to learn what to build and what to buy. Positions are judgements, so the value lies in the conversation.

Build, buy or something in between

The Government Digital Service's advice is a sound default: buy common services, and build where the capability is core to your organisation. This matches the core and generic subdomains of Section 5.7. The food-delivery platform would buy payments and messaging but build dispatch. Between those poles lie adopting open source, configuring a product and partnering. GDS warns that heavy customisation is expensive and makes products harder to update, so prefer configuration. It also warns that *you cannot outsource risk*: whichever route

you take, you still need the people and knowledge to oversee it. Compare options over several years, using the methods above, and count integration, operations, staff time, price rises and the cost of leaving (Section 10.8). Keep the supplier behind your own interface. Review the decision as components evolve.

Evaluating vendors and running a proof of concept

Fix the criteria before the first demonstration, or the best salesperson sets them. Separate knock-outs from scored criteria, and weight the latter as in Section 1.6. Knock-outs often include data location, exit terms and security evidence. Where a supplier handles personal data, Article 28(1) of the UK GDPR allows only processors providing sufficient guarantees. Then test the riskiest claim with a **proof of concept**: one question, a fixed time box and a pass mark agreed in advance. Use your own data and peak load, and include failure: what does a new dispatch-mapping vendor do when its API slows during Friday dinner? Try at least two candidates where you can. Treat the code as disposable, and agree beforehand what result would end the evaluation, so that sunk effort does not decide it.

Ethics: what the professional codes ask

Architectural decisions shape what systems do to the people who use them and the people they affect without consent, so architects carry ethical responsibilities as well as technical ones. Two professional codes are particularly relevant. The ACM Code of Ethics and Professional Conduct, updated by the Association for Computing Machinery in 2018, opens with general principles: contribute to society and human well-being, acknowledging that all people are stakeholders in computing; avoid harm; be honest and trustworthy; be fair and take action not to discriminate; respect the work required to produce new ideas; respect privacy; and honour confidentiality. Its professional responsibilities include giving comprehensive and thorough evaluations of computer systems and their impacts, including analysis of possible risks, and designing systems that are robustly and usably secure; its leadership principles ask for special care with systems that become integrated into the infrastructure of society.

The BCS Code of Conduct, from BCS, The Chartered Institute for IT in the UK, is organised around four duties: to the public interest (including public health, privacy, security, wellbeing, the environment and equal access), to professional competence and integrity, to the relevant authority (the employer or client) and to the profession. The joint ACM and IEEE Computer Society Software Engineering Code of Ethics (1999) adds eight principles beginning with the public interest, which takes precedence over the interests of client and employer.

For architects, these codes translate into concrete obligations. Analyse the foreseeable effects of a design, not only its functional correctness. Tell decision makers honestly about risks, including risks they would prefer not to hear. Refuse to design systems intended to deceive or

to harm, and escalate when asked to. Record dissent on ethical grounds in the decision record. And know where to turn if internal escalation fails: in the UK, for example, the Public Interest Disclosure Act 1998 protects workers who report certain kinds of wrongdoing.

Foreseeable harms and architectural responses

Harm	Food-delivery example	Architectural responses
Privacy	Customers' home addresses and order histories kept indefinitely and shared widely with analytics	Data minimisation, retention limits, purpose-based access, pseudonymisation; see Chapter 8 on privacy by design
Security	Courier identity documents in a bucket reachable from the internet	Threat modelling, least privilege, encryption, secrets management, supply-chain controls
Bias and unfair outcomes	A dispatch algorithm that systematically offers fewer jobs to couriers in certain areas or with older vehicles	Fairness criteria in requirements, monitoring outcomes by group, explainability, human review of contested decisions
Accessibility	Checkout that cannot be completed with a screen reader	WCAG 2.2 level AA as a constraint, accessible components in the design system, automated and manual testing
Safety	Incentives that reward couriers for speed, or allergen information that is shown inconsistently	Safety requirements and hazard analysis, rate limits on incentives, a single source of truth for allergens
Manipulation	Hidden service fees shown only at the last step; subscriptions that are easy to join and hard to cancel	Transparent pricing in the API contract, symmetric sign-up and cancel journeys, review of persuasive design

Dark patterns and other foreseeable harms

The table lists the harms most often within an architect's reach. Several are covered in depth elsewhere: privacy and security in Chapter 8, accessibility as a quality in Section 2.7, and bias in machine-learning systems in Chapter 12. Accessibility is increasingly a legal constraint as well as an ethical one; in the EU, the European Accessibility Act has applied to many consumer services, including e-commerce, since 28 June 2025.

Manipulative design deserves particular mention because it is often requested deliberately. Harry Brignull coined the term dark pattern in 2010 for interfaces designed to trick people into choices they would not otherwise make; he now prefers the term deceptive patterns. Familiar examples include drip pricing (fees revealed only at the end), confirmshaming (making the no option feel shameful), fake urgency and scarcity, pre-ticked consent boxes and subscriptions

that are easy to start and hard to cancel. Regulators have acted: the EU Digital Services Act (2022) prohibits online platforms from designing interfaces that deceive or manipulate their users, and the US Federal Trade Commission published a staff report on dark patterns in 2022. Many of these patterns need architectural support (a pricing API that hides fees until checkout, a cancellation flow routed through a retention service), so architects are in a position to notice and challenge them.

The simplest way to make harms visible is to look for them on purpose, at the same moments you look for technical risks. Add a harms question to threat modelling and design reviews: who could be hurt by this system working as designed, by it failing, or by it being misused? Practices such as consequence scanning, developed by the UK think tank Doteveryone, give teams a short, structured format for this conversation during planning.

Sustainability: green software

Software does not burn fuel, but the electricity and hardware it uses do, and architectural decisions fix much of that footprint for years. The Green Software Foundation, launched in 2021 under the Linux Foundation, publishes six core areas of green software practice: carbon efficiency (emit the least carbon possible), energy efficiency (use the least energy possible), carbon awareness (do more when electricity is cleaner and less when it is dirtier), hardware efficiency (use the least embodied carbon possible), measurement, and climate commitments (understand exactly how a reduction is achieved).

Carbon-aware computing exploits the fact that the carbon intensity of electricity varies by hour and by region with the mix of wind, solar, gas and other sources. Demand shifting moves flexible work, such as batch jobs, model training, backups and report generation, to times or regions where electricity is cleaner; demand shaping reduces what the software does when it is dirtier, for example by serving lower-resolution media or deferring non-urgent recalculation. Grid data services, such as the Carbon Intensity API for Great Britain, and open-source tools such as the foundation's Carbon Aware SDK provide the signals. As a simple illustration, a nightly batch job that uses 400 kWh emits about 120 kg CO_2e at an illustrative 300 g per kWh in the early evening, but about 48 kg if moved to a windy night at 120 g per kWh.

Embodied carbon, the emissions from manufacturing and disposing of hardware, is the reason hardware efficiency matters: higher utilisation of fewer machines, scaling to zero when idle and extending device lifetimes (including customers' phones, by not demanding new hardware for each release) all reduce it. The foundation's Software Carbon Intensity specification, published as ISO/IEC 21031:2024 and introduced in Section 2.7, combines operational and embodied emissions per functional unit, such as per order delivered, and excludes offsets so that only real reductions improve the score.

Architectural levers include right-sizing and autoscaling, removing idle environments, data retention limits (stored data has an energy and hardware cost), efficient data formats and caching, choosing regions with cleaner grids where latency and data residency allow, and questioning whether every feature needs a large model call. Beware of the rebound effect, the modern form of William Stanley Jevons's 1865 observation about coal: efficiency that makes something cheaper can increase total use, so measure total emissions as well as intensity.

Checklist: an ethical and sustainable architecture review

Use alongside the technical review methods of Sections 11.1 and 11.2. Each item should produce either a recorded decision or an action with an owner.

- [ ] **Name who is affected**
  List the people affected by the system, including those who never use it directly (couriers, neighbours, people whose data is processed), and their interests.
- [ ] **Ask how it could cause harm**
  For working as designed, failing and being misused, ask who could be hurt and how; record the answers with the threat model.
- [ ] **Minimise personal data**
  Check that every personal data item has a purpose, a retention period and an access rule, and that the system can honour deletion.
- [ ] **Check fairness of automated decisions**
  Identify decisions made or ranked by algorithms, define fairness criteria and plan how outcomes will be monitored by group and challenged.
- [ ] **Confirm accessibility**
  Treat WCAG 2.2 level AA (or the applicable legal standard) as a constraint and confirm how it will be tested.
- [ ] **Look for manipulative design**
  Check pricing, consent, sign-up and cancellation journeys for hidden costs, pre-ticked boxes, false urgency and asymmetry.
- [ ] **Review safety implications**
  Identify any physical or health consequences, such as allergens, incentives or vehicle use, and the controls around them.
- [ ] **Estimate the carbon footprint**
  Estimate energy, grid carbon intensity and embodied emissions per functional unit (SCI), and identify the largest contributors.
- [ ] **Use carbon-aware options**
  Identify work that can shift in time or region, idle capacity that can be removed and data that can be deleted sooner.
- [ ] **Make the business case honest**
  Include run costs, the do-nothing baseline and a sensitivity analysis; separate measured benefits from hoped-for upside.

- [ ] **Record dissent and escalation**
  Record ethical objections in the decision record and know the route for escalating concerns that are not resolved.

- [ ] **Plan to measure**
  Choose indicators (harm reports, accessibility defects, SCI, cost per order) and a date to review them against the expectations above.

WATCH OUT

Common mistakes

- Assuming the do-nothing option costs nothing, which makes every investment look worse than it is.
- Counting the same benefit twice, for example as developer time saved and again as faster delivery of the same features.
- False precision: NPV to the nearest pound on estimates that are good to within 30%.
- Leaving out running costs, transitional costs and decommissioning.
- Treating ethics as a compliance checkbox at the end rather than a question asked during design.
- Claiming carbon reductions through offsets or accounting changes rather than through real reductions in energy and hardware.

CHECK YOUR UNDERSTANDING

▼ Why must a business case include a do-nothing option?

Because doing nothing has its own rising costs and risks, which form the baseline against which every other option's benefits are measured.

▼ Why should an architect use the discount rate set by finance rather than choose one?

Because it reflects the organisation's cost of capital and risk policy, and using the same rate makes architecture investments comparable with every other investment competing for the money.

▼ What is embodied carbon, and name one architectural way to reduce it.

The emissions from making, transporting and disposing of hardware; it falls with higher utilisation of fewer machines, scaling to zero when idle or extending device lifetimes.

▼ **What is the difference between demand shifting and demand shaping?**

Demand shifting moves flexible work to cleaner times or regions; demand shaping changes how much work the software does depending on how clean the electricity is.

▼ **Try it: A migration costs £200,000 now and saves £90,000 a year for three years. Finance's discount rate is 10%. Work out NPV and payback. Then test savings of £75,000 a year. What do you tell the board?**

NPV at £90,000: 81,818 + 74,380 + 67,618 - 200,000 = about +£23,800. Payback: about 2.2 years. At £75,000, NPV is about -£13,500, so the case turns negative. Tell the board it clears the rate, but narrowly, and depends on the savings estimate. Recommend staged funding: fund a first slice, measure real savings, and then decide on the rest. Show the option of doing nothing, too.

SOURCES

- ACM Code of Ethics and Professional Conduct: https://www.acm.org/code-of-ethics
- BCS Code of Conduct: https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct/
- Green Software Foundation: Principles of green software: https://learn.greensoftware.foundation/introduction
- Software Carbon Intensity (SCI) specification: https://sci.greensoftware.foundation/
- Harry Brignull: Deceptive Patterns: https://www.deceptive.design/
- Finding a path (Simon Wardley, chapter 2): https://medium.com/wardleymaps/finding-a-path-cdb1249078c0
- On being lost (Simon Wardley, chapter 1): https://medium.com/wardleymaps/on-being-lost-2ef5f05eb1ec
- The Four Stages of Evolution (wardleymaps.com): https://www.wardleymaps.com/book-passages/wardley-on-evolution
- Choosing technology: an introduction (GOV.UK Service Manual): https://www.gov.uk/service-manual/technology/choosing-technology-an-introduction
- To build or to buy, that's the technology question (GDS, 2021): https://technology.blog.gov.uk/2021/02/03/to-build-or-to-buy-thats-the-technology-question
- UK GDPR Article 28: https://www.legislation.gov.uk/eur/2016/679/article/28

Further reading

- *Software Architecture in Practice, 4th edition*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. The standard textbook; its chapters on architecture evaluation describe the ATAM and its lightweight variant from the method's authors.
- *ATAM: Method for Architecture Evaluation (CMU/SEI-2000-TR-004)*, Rick Kazman, Mark Klein and Paul Clements (2000), Software Engineering Institute, Carnegie Mellon University. The original technical report defining the ATAM's steps, roles and outputs, with a worked case.
  https://www.sei.cmu.edu/library/atam-method-for-architecture-evaluation/
- *Building Evolutionary Architectures: Automated Software Governance, 2nd edition*, Neal Ford, Rebecca Parsons, Patrick Kua and Pramod Sadalage (2022), O'Reilly Media. The source of architectural fitness functions and the clearest account of evolutionary architecture.
  https://oreilly.com/library/view/building-evolutionary-architectures/9781492097532
- *Performing a Project Premortem*, Gary Klein (2007), Harvard Business Review. The short article that introduced the pre-mortem: imagine failure first, then prevent it.
  https://hbr.org/2007/09/performing-a-project-premortem
- *Scaling the Practice of Architecture, Conversationally*, Andrew Harmel-Law (2021), martinfowler.com. Describes the advice process, advisory forums, team-sourced principles and an internal technology radar.
  https://martinfowler.com/articles/scaling-architecture-conversationally.html
- *Design It! From Programmer to Software Architect*, Michael Keeling (2017), Pragmatic Bookshelf. A practical guide to architecture as a team activity, with many workshop, design and evaluation activities for teams.
- *Kill It with Fire: Manage Aging Computer Systems (and Future Proof Modern Ones)*, Marianne Bellotti (2021), No Starch Press. A candid, experience-based guide to modernising legacy systems without a doomed rewrite.

Glossary

ACM Code of Ethics
The Association for Computing Machinery's Code of Ethics and Professional Conduct, updated in 2018, whose principles include contributing to society, avoiding harm, honesty, fairness, privacy and confidentiality.

Anti-corruption layer
A translation layer, from domain-driven design, that converts between another system's model and your own, so that the other model's concepts and quirks do not leak into your code.

Architect elevator
Gregor Hohpe's image of the architect as someone who rides between an organisation's penthouse (strategy and executives) and its engine room (code and infrastructure), connecting the two.

Architectural approach
A significant design decision or combination of decisions, such as a style, pattern or tactic, that the architecture uses to achieve a quality: for example an event broker with an outbox, or a warm standby region.

Architecture advisory forum
A regular, open meeting where proposed decisions, spikes and their advice are discussed. It offers advice and visibility but does not itself decide.

Architecture evaluation
A structured examination of an architecture, before or after it is built, to find out whether its decisions will let the system meet its business goals and quality requirements, and where they put those goals at risk.

Architecture governance
The practices by which an organisation makes sure significant technical decisions are made well, are consistent with its goals and are actually followed, including who decides, how and with what checks.

Architecture kata
A practice exercise created by Ted Neward in which small teams design an architecture for a short, deliberately incomplete brief, then present and defend it to their peers.

Architecture principle
A durable rule that guides architectural decisions, written in TOGAF style with a name, a statement, a rationale and its implications.

Architecture review board
A standing group of senior architects that reviews and approves significant designs, usually at set points in a project. It can add independent judgement but often becomes a slow gate.

ATAM (Architecture Tradeoff Analysis Method)
The SEI's scenario-based evaluation method (Kazman, Klein and Clements, 2000). In nine steps over two meetings it relates business drivers and prioritised scenarios to architectural approaches and produces risks, non-risks, sensitivity points, trade-off points and risk themes.

Atomic fitness function
A fitness function that checks one architectural characteristic in a single context, such as a test that forbids imports between two packages.

Big-bang rewrite
Replacing a system by building a new one from scratch and switching over all at once; widely regarded as high risk because value arrives only at the end and hidden behaviour is easily lost.

Branch by abstraction
A technique for large changes inside one codebase: put an abstraction in front of the part to be replaced, move callers to it, build the new implementation behind it, switch over and remove the old one, keeping the system releasable throughout.

Business case
A structured argument for an investment that sets out the problem, the options (including doing nothing), their costs, benefits and risks, and a recommendation that decision makers can compare with other uses of the money.

Business drivers
The business goals, constraints and context that explain why a system exists and what success means, such as growth targets, regulation or contracts. In the ATAM they are presented first so that every finding can be traced to them.

Carbon-aware computing
Running software so that it does more when electricity is cleaner and less when it is dirtier, by shifting work in time or location or by shaping demand.

CBAM (Cost Benefit Analysis Method)
An SEI method that builds on the ATAM by estimating the benefit of each architectural strategy from the utility it adds to prioritised scenarios, and dividing by its cost to compare value for cost.

Chaos experiment
A controlled test that injects a real failure, such as killing an instance or slowing a dependency, to check a stated hypothesis about how the system keeps its steady state.

Characterisation test
A test that records what existing code actually does, rather than what it should do, so that changes can be checked against current behaviour. The term comes from Michael Feathers.

Complicated-subsystem team
A team that owns a part of the system needing scarce specialist knowledge, such as an optimisation algorithm or video codec, so that stream-aligned teams can use it without mastering it.

Consensus
A decision rule in which the group continues until everyone actively agrees, or at least supports, the proposal; inclusive but slow, and prone to stalemate in large groups.

Consent decision-making
A decision rule in which a proposal is adopted unless someone raises a reasoned objection that it would cause harm or move the group backwards; it asks whether people can live with a proposal, not whether they prefer it.

Continual fitness function
A fitness function that runs all the time against the live system, such as SLO monitoring or synthetic transactions, rather than being triggered by a build or event.

Cost of delay
The value lost for each unit of time a piece of work is not delivered; Don Reinertsen popularised it as a way to prioritise work economically.

Dark launching
Calling new back-end behaviour in production with real traffic while hiding its results from users, to test performance and correctness before a visible release.

Dark pattern
A user interface designed to trick or pressure people into choices they would not otherwise make, such as hidden costs or hard-to-cancel subscriptions; Harry Brignull coined the term in 2010 and now calls them deceptive patterns.

Decision rights
An explicit statement of who may make which decisions, who must be consulted and who must be informed; settling them before a debate prevents most disputes about process.

Design review (RFC review)
A written review of a proposed design document or request for comments, in which affected people comment asynchronously before the design is approved and its decisions recorded.

Disagree and commit
A norm in which people state their disagreement openly before a decision and then commit fully to carrying it out once it is made.

Discount rate
The annual rate used to convert future money into today's value, reflecting the cost of capital and risk; usually set by the finance department.

Embodied carbon
The greenhouse gas emitted in making, transporting and disposing of hardware, as opposed to the emissions from the electricity it uses while running.

Enabling team
A team of specialists that helps other teams acquire a missing capability, such as test automation or observability, for a limited period and then moves on, rather than doing the work for them.

Event interception
A legacy displacement pattern that catches state-changing requests or events on their way into a legacy system and forwards some of them to a new component as well, or instead.

Evolutionary architecture
An architecture that supports guided, incremental change across multiple dimensions, with fitness functions providing the guidance (Ford, Parsons, Kua and Sadalage).

Fracture plane
A natural seam along which a software system can be split so that each part suits one team, for example a bounded context, a regulatory boundary, a different change cadence or a different risk profile.

Gate (governance)
A checkpoint at which work stops until someone approves it, such as a review-board sign-off before build or release.

Holistic fitness function
A fitness function that checks several characteristics together in a shared context, such as a load test that measures latency while security and logging are switched on.

Interaction mode
In Team Topologies, one of three deliberate ways two teams work together: collaboration, X-as-a-Service or facilitating. Modes are expected to change as boundaries are discovered and settle.

Latency budget
The share of an end-to-end response-time target allocated to each step or service on a request's path, so that each team knows the limit it must stay within.

Legacy system
An existing system that still delivers value but has become hard, risky or expensive to change, often because of age, missing tests, scarce skills or unsupported technology.

Net present value (NPV)
The sum of an investment's future cash flows, each discounted back to today's value, minus the initial cost; a positive NPV means the investment earns more than the discount rate.

Non-risk
A decision judged sound for a given scenario, recorded together with the assumptions it depends on, so that it can be re-examined if those assumptions change.

Parallel run
Running the old and new implementations side by side on the same inputs and comparing their results, while the old one stays authoritative, to prove the new one is equivalent before switching.

Paved road (golden path)
A supported, well-documented default way to build and run software, such as a service template with logging, security and deployment already set up, which teams may leave when they have good reason.

Payback period
How long it takes for cumulative benefits to repay an investment's cost; the discounted payback period does the same with discounted cash flows.

Platform team
A team that provides internal services, tools and infrastructure as a self-service product so that stream-aligned teams can deliver without deep expertise in the underlying technology.

Pre-mortem
Gary Klein's technique (2007) in which a team imagines that its project has already failed and each person writes down why, which surfaces risks that people are otherwise reluctant to raise.

Psychological safety
A shared belief within a team that it is safe to take interpersonal risks such as admitting mistakes or challenging a senior colleague; studied by Amy Edmondson since 1999.

Risk (in architecture evaluation)
An architectural decision, or a missing decision, that may prevent the system from meeting an important quality requirement or business goal.

Risk register
A maintained list of identified risks with their probability, impact, owner, mitigation and status, so that risks are tracked to closure rather than forgotten after a meeting.

Risk theme
A group of related risks that share an underlying cause, linked to the business drivers it threatens. Risk themes are usually the most useful output of an evaluation for managers.

Risk-storming
Simon Brown's collaborative technique in which people silently mark risks on architecture diagrams with sticky notes, scored by probability and impact, then converge and agree priorities and mitigations.

Sensitivity point
A property of one or more components or connections that is critical to achieving a particular quality response, such as the number of synchronous hops on the order path.

Seven Rs
The migration strategies AWS lists for moving applications to the cloud: retire, retain, rehost, relocate, repurchase, replatform, and refactor or re-architect. They grew from Gartner's earlier five Rs.

Socio-technical architecture
Designing the software structure and the team structure together, because each shapes the other. The idea goes back to socio-technical systems research at the Tavistock Institute, beginning with Eric Trist and Ken Bamforth's 1951 coal-mining study.

Software Carbon Intensity (SCI)
The Green Software Foundation's specification, published as ISO/IEC 21031:2024, that rates software as ((E × I) + M) per R: energy times grid carbon intensity plus embodied emissions, per functional unit.

Staff-plus engineer
A senior individual contributor (staff, principal or distinguished engineer) who exercises technical leadership across teams without managing people; a term popularised by Will Larson.

Strangler fig
An incremental modernisation approach, named by Martin Fowler, in which new components gradually take over behaviour from a legacy system behind an interception layer until the old system can be switched off.

Stream-aligned team
A long-lived, cross-functional team aligned to one flow of change, such as a product, user journey or business capability, that can build, run and change its part end to end. Most teams in an organisation should be of this type.

TARA
The Tiny Architectural Review Approach, Eoin Woods's lightweight method for reviewing an existing system by combining expert judgement with automated analysis of its code and runtime data.

Team API
Everything a team offers to and expects from other teams: its code interfaces, documentation, versioning rules, ways of working, communication channels and how to request work from it.

Team Topologies
An approach to organising software teams for fast flow, set out by Matthew Skelton and Manuel Pais in the book of the same name (2019; second edition 2025): four team types, three interaction modes and team cognitive load as the main constraint.

Technology radar
A periodically refreshed map of technologies and techniques, grouped into quadrants and placed in rings such as Adopt, Trial, Assess and Hold (renamed Caution in recent Thoughtworks editions), popularised by Thoughtworks.

Transitional architecture
Temporary software, such as routing layers, adapters and synchronisation jobs, built only to let old and new systems coexist during a migration and removed when it is complete.

X-as-a-Service

An interaction mode in which one team consumes something another team provides, through a clear, documented interface and with little day-to-day collaboration.