BE SOFTWARE ARCHITECTURE WISE · CHAPTER 2

Quality Attributes and Architecturally Significant Requirements

Architecture exists to deliver qualities: speed under load, staying up when parts fail, being safe to change, keeping data secret. This chapter shows how to turn vague wishes such as 'fast', 'reliable' and 'secure' into precise, testable requirements that can drive design decisions. It covers the difference between functional requirements, quality attributes and constraints, the ISO/IEC 25010 quality model, quality attribute scenarios, the arithmetic of performance and availability, designing for change, security, usability and cost as qualities, and how to elicit, prioritise and trade them off. It suits developers moving towards architecture and anyone who writes or reviews requirements.

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

AT A GLANCE

You will learn to

- Distinguish functional requirements, quality attributes and constraints, and recognise an architecturally significant requirement.
- Use the ISO/IEC 25010:2023 product quality model as a checklist without letting it turn into paperwork.
- Write six-part quality attribute scenarios with measurable response measures, and turn vague statements into testable ones.
- Calculate with percentiles, Little's law, simple queueing formulas and Amdahl's law to reason about performance and scalability.
- Calculate availability from the nines, from MTBF and MTTR, and for components in series and in parallel.
- Explain how modifiability, testability, deployability, security, usability, cost and sustainability are specified and achieved.
- Elicit and prioritise quality requirements with a utility tree, and explain the trade-offs between competing qualities.

Before you start

Chapter 1, especially the sections on stakeholders and concerns and on decisions and trade-offs. Some comfort with simple arithmetic (percentages, multiplication and division).

Time

About 3 hours to read; 45 minutes for the test.

Key terms

quality attribute · constraint · architecturally significant requirement · ISO/IEC 25010 · quality attribute scenario · response measure · latency and throughput · percentile · Little's law · availability · MTTR · tactic · utility tree · fitness function

Why qualities, not features, decide the architecture

Ask any group of stakeholders what a system must do and you will get a list of features: search for a flight, pay for a basket, book an appointment. Ask how well it must do them and the answers are usually vaguer: it should be fast, it should not go down, it should be secure, it should be easy to change. Those second answers are quality attributes, and they are what architecture is mostly about. The same list of features can be delivered by a single program on one server or by forty services spread across three continents. What separates those designs is not what the system does but how many people it must serve at once, how much downtime the business can tolerate, how often the code will change and how sensitive the data is.

Chapter 1 described architecture as the set of significant decisions that satisfy stakeholders' concerns through trade-offs. This chapter makes those concerns precise. A requirement that has a measurable effect on the structure of a system is an architecturally significant requirement, and most of them turn out to be quality attributes or constraints rather than functions. Finding them early, and stating them so that they can be tested, is one of the most valuable things an architect does.

How the chapter fits together

Section 2.1 separates functional requirements, quality attributes and constraints, explains why many architects avoid the term 'non-functional requirements', and works through a concert-ticketing system on the day a big tour goes on sale. Section 2.2 introduces the ISO/IEC 25010:2023 product quality model, with its nine characteristics, and shows how to use it as a memory aid rather than a form to fill in. Section 2.3 teaches the single most useful technique in the chapter: the six-part quality attribute scenario, which turns 'must be fast' into a statement you can test.

Sections 2.4 to 2.7 then visit the qualities that most often shape architectures. Section 2.4 covers performance and scalability with real arithmetic: latency and throughput, percentiles, tail latency, Little's law, queueing and Amdahl's law. Section 2.5 does the same for availability and reliability: the nines, mean time between failures and mean time to repair, and the availability of components in series and in parallel. Section 2.6 covers the qualities of change (modifiability, testability and deployability), and Section 2.7 gathers security, usability and accessibility, interoperability, cost, energy efficiency, safety and regulatory qualities.

Section 2.8 closes the loop. It shows where quality requirements come from, how a quality attribute workshop and a utility tree turn a crowd of opinions into a ranked list, which qualities conflict with which, and how fitness functions keep checking the important qualities automatically after the architecture is built.

From vague wishes to numbers

This is the chapter where the course starts to calculate. You will work out how many requests are in flight at once from an arrival rate and a response time, why a request to a server that is 90% busy takes, on average, ten times its service time, why spreading a job that is 10% serial across sixteen processors gives only a 6.4-fold speed-up, and how many minutes of downtime a year a 'four nines' promise actually allows. None of the mathematics is harder than school arithmetic, but it changes conversations: 'the database is slow' becomes 'the connection pool is sized for 100 concurrent requests and Little's law says we need 400 during the evening peak'.

Numbers also expose false comfort. An average response time of 120 milliseconds can hide the fact that one request in fifty takes more than a second, which is why professionals specify the 95th or 99th percentile. A requirement with no response measure cannot be tested, and a requirement that cannot be tested will be quietly traded away when a deadline approaches.

Trade-offs are the job

No system can be maximally fast, available, secure, cheap and easy to change at once. Caching improves speed but complicates consistency; encryption and extra checks protect data but add latency; redundancy raises availability and cost together; flexible plug-in designs make change easier and performance harder to predict. The chapter therefore treats qualities as competing goals that must be ranked in context, not as boxes to tick. The architect's skill lies in knowing which two or three qualities matter most for this system, stating them precisely, and choosing designs whose weaknesses fall where the business can afford them.

The design techniques that achieve each quality, called tactics, are introduced here only in outline. Later chapters build them out: design principles in Chapter 3, architectural styles in Chapters 5 and 6, data, security, resilience and cloud practices in Chapters 7 to 10, and evaluation methods such as the ATAM and fitness functions in Chapter 11.

The one idea to take away

If you remember one thing from this chapter, make it this: a quality is not a requirement until it has a stimulus, a context and a measure. 'The checkout must be fast' is a wish. 'When 2,000 customers a minute submit orders during the Friday evening peak, 99% of checkouts complete within 1.5 seconds' is a requirement. It can be designed for, tested, monitored and argued about with evidence, and it tells everyone, from the product owner to the operations team, what success looks like.

SECTION 2.1 OF 2.8 · INTERMEDIATE · 23 MIN READ · LAST REVIEWED NOVEMBER 2026

Functional requirements, quality attributes and constraints

BEFORE YOU READ THIS

This section builds on:

- 0.8 How software gets built
- 1.3 Why it matters
- 1.5 Stakeholders and concerns

Terms it uses from earlier: Stakeholder, User story, Trade-off

THE IDEA IN BRIEF

Think of ordering a wedding cake. 'Three tiers, lemon sponge, white icing' describes what the cake is: its functions. 'Ready by 2pm on Saturday, able to survive a 40-minute drive, safe for a guest with a nut allergy' describes how well it must work: its qualities. 'The venue only allows cakes from approved bakers' is a constraint: a choice already made for you. A baker can make the same lemon sponge in very different ways, and it is the qualities and constraints, not the flavour, that decide how the cake is built and transported. Software is the same.

WHAT YOU NEED TO KNOW

- Functional requirements state what a system must do; quality attributes state how well it must do it; constraints are decisions already made, such as regulations, mandated technologies, deadlines and budgets.
- Many architects avoid the term 'non-functional requirements' because it defines a large category by exclusion, suggests the requirements are separate from functions and sounds optional.
- An architecturally significant requirement (ASR) is one whose change would force a change to the system's structure; most ASRs are quality attributes or constraints, but some functions are significant too.
- The same functions can be delivered by very different architectures; it is the quality attributes and constraints that decide between them.
- Implicit requirements, such as never losing orders or never charging twice, are rarely written down and must be actively drawn out from stakeholders, incidents and regulations.

Three kinds of requirement

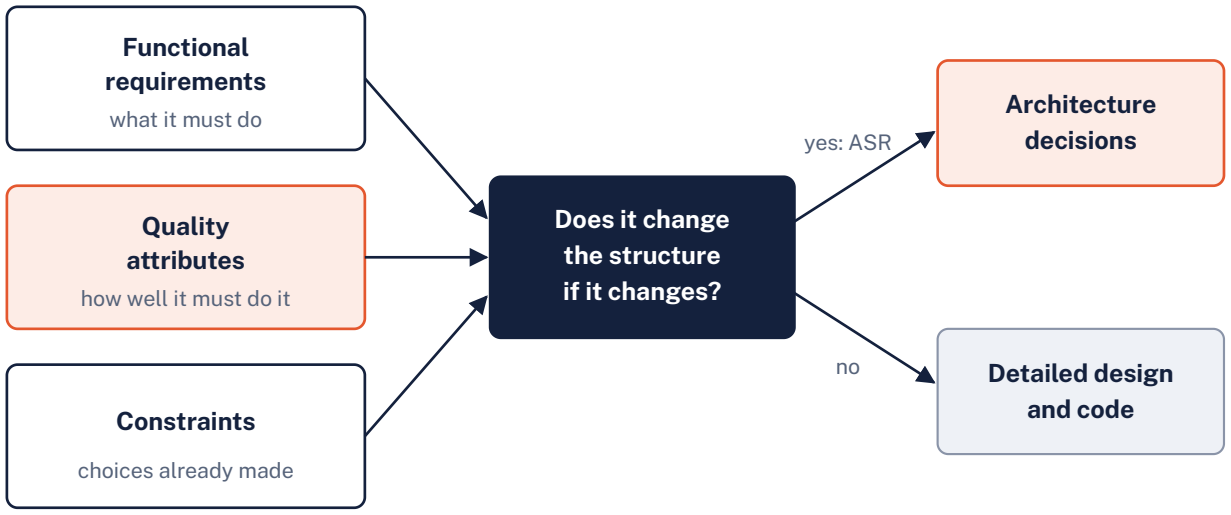
A requirement is anything a system must satisfy to be acceptable to its stakeholders. Architects find it useful to sort requirements into three kinds, because each kind influences the design in a different way. A functional requirement says what the system must do: 'a customer can add an item to the basket', 'the pharmacy system checks every new prescription against the patient's allergies', 'a rider can see the driver's position on a map'. Functional requirements describe behaviour, data and the rules that connect them. They are what most user stories and use cases capture.

A quality attribute says how well the system must do it. 'Search results appear within half a second', 'the booking service is available 99.95% of the time', 'a new payment provider can be added in under two weeks', 'patient records are readable only by clinicians treating that patient'. Quality attributes are properties of the system as a whole or of a function, and they only make sense with a measure attached: fast enough for whom, under what load, measured how?

A constraint is a decision that has already been made for you. 'The system must run in the company's existing Azure tenancy', 'all personal data stays in the European Union', 'the team knows Java and has no budget to retrain', 'the first release must be live before the tax year starts on 6 April'. Constraints remove options rather than describe behaviour. They are not negotiable within the project, although a good architect checks that each one really is a constraint and not just a habit or an assumption.

The boundaries are not always crisp, and that is fine. The point of the classification is not bureaucratic tidiness; it is to make sure that nobody collects fifty user stories and forgets to ask how many users there will be, how long the business can survive an outage, or which regulator is watching.

How requirements reach the architecture



All three kinds of requirement feed the architecture, but only those whose change would force a structural change are architecturally significant.

Why many architects avoid the term 'non-functional requirements'

Requirements documents traditionally split everything into functional and non-functional requirements (NFRs). The term is everywhere, in contracts, tender documents and job adverts, so you need to recognise it. Many architects and researchers nevertheless avoid it, for three reasons.

First, it defines a large and important category by what it is not. Anything that is not a function lands in the same bin: response times, encryption rules, the choice of cloud provider and the colour of the logo. Martin Glinz's paper *On Non-Functional Requirements* (IEEE International Requirements Engineering Conference, 2007) showed that published definitions disagree on what belongs in the category and that the same need can be classed either way depending on how it is phrased: 'only authorised staff may see salaries' reads as a function when written as an access rule and as a quality when written as a security goal.

Second, the name suggests these requirements are separate from functions and can be added later. They cannot. A quality attribute qualifies a function: it is the checkout that must be fast, the payment that must never be lost, the medical record that must be confidential. Bass, Clements and Kazman, in *Software Architecture in Practice*, therefore speak of quality attribute requirements and treat constraints as a separate category.

Third, 'non-functional' sounds optional, as if a system that works but is slow, insecure or impossible to change still 'functions'. In practice those are exactly the systems that get rewritten. Some teams prefer 'quality requirements'; others, notably at Thoughtworks, say

'cross-functional requirements' to stress that they cut across every feature. This course uses quality attributes and constraints. If your organisation says NFRs, there is no need to fight about the word; just make sure every NFR is either a measurable quality or a clearly stated constraint.

Constraints: technical, organisational, regulatory, time and budget

Constraints deserve more attention than they usually get, because they often decide more of the architecture than any feature. A constraint is simply a requirement with no remaining freedom. It helps to group them by where they come from.

Kinds of constraint and their architectural effect

Kind	Example	Typical effect on the architecture
Technical	Must integrate with the existing SAP finance system through its published interfaces; must run on the company's Kubernetes platform.	Fixes integration styles, data formats and deployment model; may rule out some databases or languages.
Organisational	Three teams in two time zones; the operations team supports only PostgreSQL; the security team must approve every external dependency.	Shapes how the system is divided (Conway's law, Chapter 3) and which technologies are realistic.
Regulatory and legal	Personal data of EU residents is processed under the GDPR; card data falls under PCI DSS; a medical device must meet safety regulations.	Drives data location, encryption, audit logging, retention rules and the separation of sensitive components.
Time	The new tax rules apply from a fixed date; a retail platform must be ready before the November peak.	Favours proven technology and simpler designs; may force a phased release with known technical debt.
Budget	Running costs must stay below a fixed sum per month; no money for commercial licences.	Limits redundancy, regions and managed services; makes cost per transaction an explicit quality.

Two habits help. First, write down the source of every constraint (a regulation, a contract, a named decision-maker) so that it can be checked and, if circumstances change, revisited. Second, distinguish real constraints from preferences dressed up as constraints. 'We must use microservices' is rarely a constraint; it is a design decision someone made early, and it should be justified by quality attributes like any other decision. Chapter 1's section on decisions and trade-offs explains why early, hard-to-reverse choices deserve that scrutiny.

Architecturally significant requirements

Most requirements do not affect the architecture. Whether the invoice shows the customer's postcode above or below the street name is a detail; it can be changed in an afternoon. An architecturally significant requirement (ASR) is one that has a measurable effect on the architecture: if it changed, the structure of the system would have to change. Recognising ASRs early matters because they are expensive to accommodate late. Adding a waiting room to a ticketing site the week before a sale is possible; partitioning the seat inventory across servers after launch is a rewrite.

Lianping Chen, Muhammad Ali Babar and Bashar Nuseibeh interviewed practitioners about how they recognise ASRs (*Characterizing Architecturally Significant Requirements*, IEEE Software, 2013). Their findings match most architects' experience. ASRs tend to be hard to satisfy, they often concern the system as a whole rather than one feature, they frequently conflict with each other, and they are often stated vaguely or not at all. Quality attributes and constraints make up the majority, but some functional requirements are significant too.

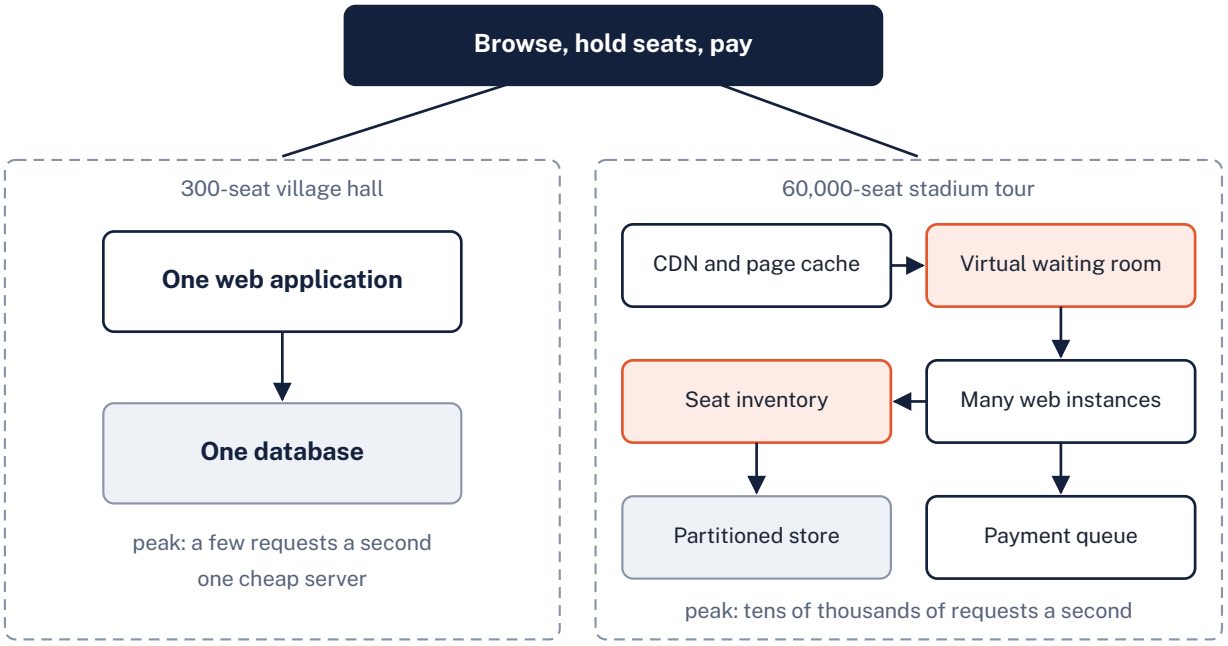
Examples of functional ASRs: 'every change to a patient record must be traceable to a person and a time' (it needs an audit mechanism across all components), 'users can collaborate on the same document in real time' (it demands a synchronisation model and probably a different communication style), and 'customers can undo a bank transfer within 30 minutes' (it affects how money moves between accounts). Each forces a structural decision, not just more code.

Why qualities shape architecture more than functions do

Here is the central idea of this chapter. If functionality were all that mattered, almost any structure would do: you could write the whole system as one large program and it would still compute the right answers. Bass, Clements and Kazman make this point directly: functionality does not determine architecture, because many different architectures can deliver the same functions. What separates them is how well each one achieves the qualities.

Consider two ticketing systems with identical features: browse events, hold seats, pay. One sells tickets for a 300-seat village hall; the other sells a 60,000-seat stadium tour. The village hall's busiest moment might bring a few requests per second, and a single web application with one database is the right answer: cheap, simple and easy to change. The stadium tour must absorb a burst thousands of times larger, must never sell the same seat twice, and must keep bots from buying everything in the first second. That system needs a content delivery network, a queue in front of the shop, a carefully designed seat inventory, partitioned storage and a payment flow that can wait for an overloaded provider. Same functions, completely different architectures, and the difference comes entirely from quality attributes and constraints.

Same functions, different qualities



Identical features can lead to radically different structures when the load, consistency and security requirements differ.

IF YOU ARE NEW TO THIS

Step by step: is this requirement significant?

This section's hardest idea is telling which requirements are architecturally significant (ASRs). The test is not whether a requirement is called a function or a quality, but whether changing it would change the system's structure. Try the test on three requirements.

- The test: imagine the requirement changing (the number doubled or cut, the rule tightened or removed) and ask what would have to move: components, data, or the paths between them.
- An invoice shows the customer's postcode below the street name. Move it above and one template changes in an afternoon. Nothing else moves, so this is detailed design, not architecture.
- The stadium tour must absorb a burst thousands of times larger than the village hall's few requests a second. That burst is a large part of why it needs a content delivery network, a waiting room and partitioned storage. Cut the load to the village hall's and most of that machinery could go; the hall manages with one web application and one database.
- So the load figure is an ASR. What makes it significant is its effect on the structure, not its label as a quality.
- Functions can pass the test too. Take the rule that every edit to a patient's record must show who made it and when. It sounds like a feature, but it needs an audit mechanism across all components. Remove it, and that mechanism goes, so it is significant.
- **The lesson:** ask 'what would this change move?', not 'what kind of requirement is it?'. Most requirements, of every kind, stay inside one part. The few that would move components, data or the paths between them are the ones the architecture must be built around.

Implicit qualities nobody writes down

Some of the most important qualities are never stated because every stakeholder assumes them. These implicit requirements surface only when they are violated, usually in production. Nobody writes 'do not lose orders', 'do not charge the customer twice', 'keep working when the clocks go forward', 'show the same balance on the phone and the website', or 'do not leak one customer's data to another'. Yet each of these is an ASR with heavy consequences: idempotent payment handling, durable writes, time-zone-safe scheduling, consistency between channels and strict tenant isolation.

Implicit requirements hide for social reasons. Business stakeholders think they are too obvious to mention; developers assume someone else has handled them; and requirement templates rarely prompt for them. Useful questions to flush them out are: 'What would make you furious if it happened?', 'What would put us in the newspaper?', 'What has gone wrong with the system we are replacing?', and 'What does the operations team get woken up for?'. Incident reports and support tickets of predecessor systems are a rich source; so are regulators' published guidance and the obligations in customer contracts.

WORKED EXAMPLE

Worked example: a concert-ticketing system on sale day

A ticketing company will sell a four-night stadium run (60,000 seats a night) at 10:00 on a Friday. On a normal day the site sees about 2,000 visitors an hour. Marketing expects 1.5 million people to arrive in the first ten minutes. The features are the usual ones: browse, choose seats, hold them for ten minutes, pay. What are the architecturally significant requirements?

Tickets on sale	4 nights × 60,000 seats = 240,000 tickets
Normal arrival rate	2,000 visitors ÷ 3,600 s ≈ 0.56 visitors per second
Sale-day arrival rate	1,500,000 visitors ÷ 600 s = 2,500 visitors per second, about 4,500 times the normal rate
Request rate	If each visitor's browser makes about 8 requests to the site in the first minute (pages, seat maps, API calls): 2,500 × 8 = 20,000 requests per second
Demand versus supply	1,500,000 visitors for 240,000 tickets: most people will be disappointed whatever the design, so fairness and clear communication matter as much as speed
Payment constraint	The payment provider's contract allows 300 payment authorisations per second. With an average of 2.5 tickets per order, 240,000 ÷ 2.5 = 96,000 orders; 96,000 ÷ 300 = 320 s, so payments alone take at least 5.3 minutes even if everything else is instant
ASR 1: peak load (performance)	Absorb 2,500 arrivals a second without collapsing; serve static pages and seat maps from a cache or CDN so only real purchases reach the core
ASR 2: no seat sold twice (consistency)	Seat holds must be atomic even when thousands of people click the same block at once; this shapes the inventory design and its storage
ASR 3: fairness (business rule with structural impact)	Admit buyers in arrival order through a virtual waiting room, so the core system sees a controlled flow rather than a stampede
ASR 4: bot resistance (security)	Detect and block automated buyers before they reach the inventory; rate-limit per account and per device
ASR 5: graceful degradation (availability)	If payments slow down, keep holds valid and queue payment attempts rather than failing whole orders
Constraint	Fixed date and time announced in advance; payment provider limit of 300 per second

None of these ASRs appears in the feature list. All of them come from load, consistency, fairness, security and a third-party constraint.

The payment limit converts directly into a design decision: since buyers cannot all pay at once, the system must hold seats and queue payments, which in turn sets the hold time.

The real world offers a reminder. In November 2022 Ticketmaster's presale for a major stadium tour suffered long delays and failures under demand; the company reported 3.5 billion total system requests, four times its previous peak.

Checklist: spotting architecturally significant requirements

Run through these questions for each candidate requirement, and for the system as a whole, early in a project.

- [ ] **Would changing it change the structure?**
  If doubling the number, relaxing the rule or removing the requirement would move components, data or communication paths, it is significant.
- [ ] **Does it apply across many features?**
  Requirements that cut across the whole system (audit, security, availability, internationalisation) usually need a structural answer.
- [ ] **Is it hard or new?**
  A demanding number (sub-100 ms, 99.99%), a load far above anything the team has handled, or a first-of-its-kind feature is a strong signal.
- [ ] **Does it carry high business value or risk?**
  If failing it would lose revenue, break the law or harm someone, treat it as significant even if it looks simple.
- [ ] **Does it conflict with another quality?**
  Requirements in tension (fast and strongly consistent, open and secure) need deliberate trade-offs, which are architectural decisions.
- [ ] **Is it a constraint in disguise?**
  Fixed technologies, regulations, deadlines and budgets remove options; record them with their source.
- [ ] **Has it hurt before?**
  Look at incidents, support tickets and post-mortems of the system being replaced, and of similar systems elsewhere.
- [ ] **Is anyone assuming it silently?**
  Ask what would make stakeholders furious or put the company in the news; those answers reveal implicit requirements.

WATCH OUT

Common mistakes

- Collecting hundreds of user stories and one line saying 'the system must be fast, secure and scalable'.
- Treating every requirement as significant. If everything is an ASR, nothing is prioritised; most requirements belong to detailed design.
- Accepting a design decision ('use Kafka', 'use microservices') as if it were a constraint, without a quality attribute that justifies it.
- Writing qualities without measures. 'Highly available' cannot be designed for or tested; Section 2.3 shows how to fix that.
- Assuming quality can be added after the features are built. Performance, security and availability depend on structure and are expensive to retrofit.

How professionals work with ASRs

Experienced architects keep a short, living list of ASRs, typically five to fifteen items, rather than a long requirements document. Each item states the quality as a testable scenario (Section 2.3), names the stakeholder who cares, records its priority and links to the decisions it drove. The arc42 template, for example, asks for the top three to five quality goals on its first page and a fuller list of quality requirements later. The list is revisited whenever a big decision is made or the business changes direction, and the most important items are eventually backed by automated checks called fitness functions, introduced in Section 2.8.

CHECK YOUR UNDERSTANDING

▼ Is 'patients can book appointments online' a functional requirement, a quality attribute or a constraint?

A functional requirement: it says what the system must do. A quality would add how well, for example how quickly bookings are confirmed or how many can be handled at 8am on a Monday.

▼ Why is 'the system must be scalable' not yet a useful requirement?

It has no stimulus, context or measure. It does not say how much load, how quickly the load grows, or what response must be preserved, so nobody can design for it or test it.

▼ **Give one example of a functional requirement that is architecturally significant.**

For instance, a full audit trail of every change to medical records, or real-time collaborative editing. Each needs a structural mechanism across the system, not just more code in one place.

▼ **What question best separates a significant requirement from an ordinary one?**

Would the structure of the system have to change if this requirement changed? If yes, it is architecturally significant.

▼ **Try it: A ticketing brief says: fans can choose seats; confirmation within 2 seconds at sale-day peak; hosting must stay with the current cloud provider; nobody is charged twice. Classify each, and say which are architecturally significant.**

Choosing seats is functional; it matters but barely decides the structure. Confirmation within 2 seconds at peak is a quality attribute and clearly significant, since it drives queues and capacity. The hosting rule is a constraint, a decision already made, and it is significant because it limits the options. Nobody charged twice is an implicit quality, usually unwritten, and highly significant: it shapes idempotency and how payments are recorded.

SOURCES

- Martin Glinz, On Non-Functional Requirements (RE 2007): https://files.ifi.uzh.ch/rerg/amadeus/publications/papers/RE07_Glinz.pdf
- Chen, Ali Babar and Nuseibeh, Characterizing Architecturally Significant Requirements (IEEE Software, 2013): https://pure.itu.dk/en/publications/characterizing-architecturally-significant-requirements/
- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- arc42: Introduction and goals (quality goals): https://docs.arc42.org/section-1/

SECTION 2.2 OF 2.8 · INTERMEDIATE · 21 MIN READ · LAST REVIEWED NOVEMBER 2026

The quality models: ISO/IEC 25010 and how to use it

BEFORE YOU READ THIS

This section builds on:

- 1.5 Stakeholders and concerns
- 2.1 Functional and quality requirements

Terms it uses from earlier: Quality attribute, Stakeholder, Requirement

THE IDEA IN BRIEF

When you buy a used car, a good checklist stops you forgetting things: engine, brakes, tyres, rust, paperwork, fuel use. You do not need every item to be perfect, but you want to have looked at each one and decided whether it matters to you. A quality model is that checklist for software. The international standard ISO/IEC 25010 lists nine areas, such as performance, reliability, security and how easy the software is to change, so that nothing important is missed. The skill is using it to prompt good questions, not to fill in forms.

WHAT YOU NEED TO KNOW

- A quality model gives a shared vocabulary of quality characteristics and sub-characteristics and works as a checklist for finding the qualities that matter.
- ISO/IEC 25010:2023 defines nine product characteristics: functional suitability, performance efficiency, compatibility, interaction capability, reliability, security, maintainability, flexibility and safety.
- Compared with 2011, usability became interaction capability, portability became flexibility, safety was added, and quality in use moved to a separate standard, ISO/IEC 25019:2023.
- Other lists such as FURPS+, the qualities in Software Architecture in Practice and Q42 overlap heavily with ISO/IEC 25010; none is uniquely correct.
- Use a model to prompt a quick, explicit decision about each characteristic, then turn only the relevant ones into a few measurable, prioritised scenarios.

What a quality model is for

Section 2.1 argued that quality attributes decide most of an architecture. The obvious next question is: which qualities? Left to themselves, stakeholders mention the two or three that hurt them last, usually speed and uptime, and forget the rest until it is too late. A quality model

is a structured list of quality characteristics, each broken into narrower sub-characteristics, with agreed names and definitions. It does two jobs. It gives people a shared vocabulary, so that 'reliability' means the same thing to the product owner, the tester and the supplier. And it acts as a checklist, prompting you to consider qualities that nobody has raised yet.

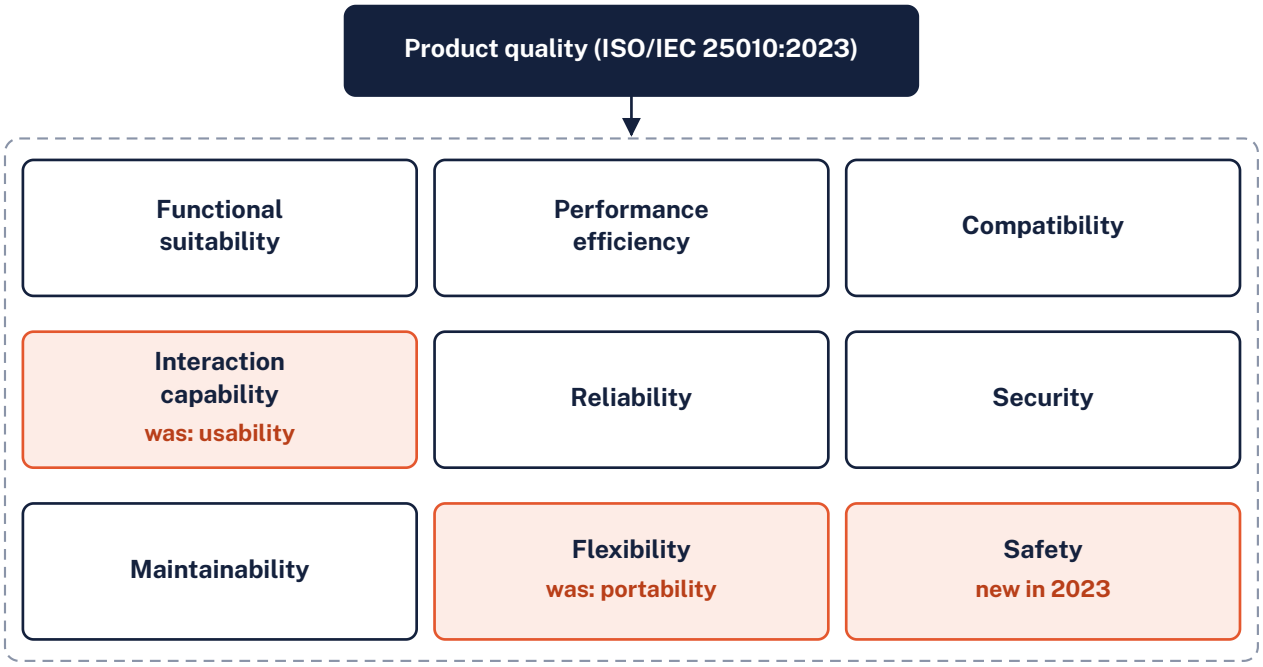
A model does not tell you which qualities matter for your system or how much of each you need. That comes from stakeholders and business goals (Section 2.8). Nor does it tell you how to achieve them; that is the job of tactics, patterns and the rest of this course. It is a map of the territory, not a route through it.

ISO/IEC 25010:2023: the product quality model

The most widely referenced model is ISO/IEC 25010, part of the SQuaRE family of standards (Systems and software Quality Requirements and Evaluation). The second edition, published in November 2023, defines a product quality model of nine characteristics. It applies to software products and to wider ICT products, and the standard lists its uses: specifying requirements, checking that requirements are complete, setting design and test objectives, defining acceptance criteria and establishing quality measures. Related standards in the family define how to measure each sub-characteristic (ISO/IEC 25023) and how to evaluate a product.

The nine characteristics are functional suitability, performance efficiency, compatibility, interaction capability, reliability, security, maintainability, flexibility and safety. The first describes whether the system does the right things correctly; the other eight describe how well it does them and how well it can be operated, protected and changed. The diagram below shows the structure; the table after it lists every sub-characteristic with an example question an architect might ask.

The nine characteristics



ISO/IEC 25010:2023 groups product quality into nine characteristics; two were renamed and one added compared with the 2011 edition.

Characteristics, sub-characteristics and an example question for a mobile banking app

Characteristic	Sub-characteristics	Example question
Functional suitability	Functional completeness, functional correctness, functional appropriateness	Does the app calculate the loan repayment correctly, and does it do everything the customer needs to complete the task?
Performance efficiency	Time behaviour, resource utilisation, capacity	How quickly does the balance appear on a mid-range phone over 4G, and how many concurrent users can the back end serve?
Compatibility	Co-existence, interoperability	Can the app exchange payment data with the national faster-payments scheme, and run alongside other apps without draining shared resources?
Interaction capability (formerly usability)	Appropriateness recognisability, learnability, operability, user error protection, user engagement, inclusivity, user assistance, self-descriptiveness	Can a first-time user set up a standing order without help, and can a screen-reader user do the same?
Reliability	Faultlessness, availability, fault tolerance, recoverability	If the card-processing partner is down, can customers still see balances and move money between their own accounts?
Security	Confidentiality, integrity, non-repudiation, accountability, authenticity, resistance	Can we prove which customer authorised a transfer, and does the app resist credential-stuffing attacks?
Maintainability	Modularity, reusability, analysability, modifiability, testability	How long does it take to add a new account type, and can the transfer rules be tested without a real bank core?
Flexibility (formerly portability)	Adaptability, scalability, installability, replaceability	Can the back end grow from 1 to 5 million customers, and could we replace the push-notification provider within a quarter?
Safety	Operational constraint, risk identification, fail safe, hazard warning, safe integration	Mostly relevant to systems that can harm people, property or the environment: an infusion pump, a train controller, an industrial robot.

A few definitions are worth knowing precisely, because they differ from everyday usage. In ISO/IEC 25010, **availability** is a sub-characteristic of reliability: the degree to which a system is operational and accessible when required. **Faultlessness** is the degree to which a system performs its functions without fault under normal operation. **Capacity**, under performance efficiency, is the maximum limit of a parameter, such as concurrent users or transactions per second, that still meets requirements. **Scalability** sits under flexibility, which reflects a view of scaling as the ability to adapt to changing load rather than a matter of raw speed. **Resistance**, new under security, is the capability to sustain operations while under attack. Sections 2.4 to 2.7 return to these qualities in depth; Section 2.5 shows how to calculate availability.

What changed from the 2011 edition

The first edition, ISO/IEC 25010:2011, replaced the older ISO/IEC 9126 model and became the standard reference for a decade. Many textbooks, tender documents and templates still use its eight characteristics, so it is worth knowing what moved.

ISO/IEC 25010: 2011 compared with 2023

Topic	2011 edition	2023 edition
Number of characteristics	Eight	Nine
Usability	Usability, including user interface aesthetics and accessibility	Renamed interaction capability; aesthetics replaced by user engagement; accessibility split into inclusivity and user assistance; self-descriptiveness added
Portability	Portability: adaptability, installability, replaceability	Renamed flexibility; scalability added as a sub-characteristic
Reliability	Maturity, availability, fault tolerance, recoverability	Maturity replaced by faultlessness
Security	Confidentiality, integrity, non-repudiation, accountability, authenticity	Resistance added (withstanding attack)
Safety	Treated only within quality in use (freedom from risk)	New product characteristic with five sub-characteristics
Quality in use	Part of ISO/IEC 25010 (effectiveness, efficiency, satisfaction, freedom from risk, context coverage)	Moved to a separate standard, ISO/IEC 25019:2023 (beneficialness, freedom from risk, acceptability)

The renamings are more than cosmetic. 'Interaction capability' widens the focus from ease of use to the whole exchange of information between users and the product across different contexts of use, and the new inclusivity sub-characteristic places accessibility for people of different ages and abilities at the centre rather than the edge. 'Flexibility' groups the qualities that let a product cope with change in its environment: new platforms, new installation targets, and more or less load. Adding safety as a product characteristic reflects how much software now controls physical things, from cars to medical devices.

The second structural change is that quality in use, the quality experienced by people using the product in a real context, now lives in its own standard, ISO/IEC 25019:2023. Its three characteristics are beneficialness (usability, accessibility and suitability in use), freedom from risk (economic, health, human life, and environmental and societal risk) and acceptability (experience, trustworthiness and compliance). The split clarifies a useful distinction for architects: product quality is what you build and can test in the product; quality in use is the outcome for real people, which also depends on training, context and how the product is deployed.

WATCH OUT

Watch the edition

When a contract or a client's template says 'ISO 25010', check which edition it means. A requirement listing 'usability and portability' is almost certainly based on the 2011 model. That is not a problem in itself, but mixing the two lists in one document causes gaps (safety and resistance are easy to miss) and duplicates (accessibility appearing twice). Pick one edition and say so.

IF YOU ARE NEW TO THIS

Step by step: what you build and what people get

The idea that needs the most care here is the difference between product quality and quality in use. Product quality is a property of the software, which the team can build and test. Quality in use is what real people actually get from it, which also depends on who they are and the situation they are in. The mobile banking app shows the difference.

- Under user error protection, part of interaction capability, the bank's team writes a scenario: the app warns the customer before a payment to a payee whose name does not match the account.
- That is product quality. Testers send payments with matching and mismatching names and check that the warning appears every time it should.
- Suppose it passes every test. The bank still wants to know something else: are customers actually losing less money to mistaken or fraudulent payments? That is freedom from risk, part of quality in use, which since 2023 has its own standard, ISO/IEC 25019.
- The answer depends on more than the app. A customer in a hurry may tap straight past the warning; a fraudster on the phone may talk them into ignoring it; a customer who has learnt about such scams may stop and check. The software is the same each time; the outcome is not.
- So the two are checked differently: product quality with tests on the product, quality in use by watching real outcomes, for example how many customers report paying the wrong person.
- **The lesson:** the team can design and test product quality directly; quality in use shows only in real use. Good product quality makes good quality in use more likely, but cannot guarantee it alone.

Other lists you will meet

ISO/IEC 25010 is not the only taxonomy, and practitioners often use simpler or differently cut lists. FURPS+ grew out of FURPS, a classification developed at Hewlett-Packard and set out by Robert Grady and Deborah Caswell in *Software Metrics: Establishing a Company-Wide Program* (1987). FURPS groups requirements into functionality, usability, reliability, performance and supportability; the later '+' extension adds constraints: design, implementation, interface and physical constraints and, in some versions, business rules. FURPS+ became popular through the Rational Unified Process and its successors. Its strength is brevity; its weakness is that 'supportability' lumps together maintainability, testability, configurability, installability and localisation.

Bass, Clements and Kazman, in the fourth edition of *Software Architecture in Practice* (2021), do not try to be exhaustive. They devote a chapter each to the qualities that most often drive architectures: availability, deployability, energy efficiency, integrability, modifiability, performance, safety, security, testability and usability, followed by a chapter on working with other qualities. For each, they give a general scenario and a catalogue of tactics. Their list is organised around design rather than evaluation, which makes it especially useful to architects.

Mark Richards and Neal Ford, in *Fundamentals of Software Architecture*, use the term architecture characteristics and sort them into groups such as operational (availability, performance, scalability), structural (modifiability, extensibility, portability) and cross-cutting (security, accessibility, legal) groups. Gernot Starke's open Q42 model, maintained alongside the arc42 template, takes yet another approach: it tags well over a hundred named qualities with a short list of dimensions (such as efficient, flexible, operable, reliable, safe, secure, suitable and usable) and provides example requirements for each.

These lists disagree in detail but overlap heavily. The disagreement is itself instructive: there is no single correct decomposition of 'quality', only more or less useful ones for a purpose. Using a standard model helps when you need to communicate with procurement, auditors or other organisations; a lighter list may serve a product team better day to day.

Using a model as a checklist without creating bureaucracy

The great risk of a quality model is that it becomes a form. A requirements template with a heading for each of the 40 sub-characteristics invites people to fill every box, and the result is a document full of statements such as 'the system shall be maintainable' that nobody reads and nobody tests. Effort spreads evenly across qualities that matter very unevenly, and the two or three qualities that will really decide the architecture are lost among the rest.

A better approach treats the model as a prompt for a conversation. Walk through the characteristics with stakeholders once, quickly, asking for each: does this matter for our system, and if so, what would failure look like? Discard the irrelevant ones explicitly (writing 'safety: not applicable, the app cannot cause physical harm' is useful, because it records a decision). Turn the relevant ones into a small number of concrete, measurable scenarios (Section 2.3). Rank them (Section 2.8). Then put the model away until the next review.

A lightweight way to apply the model

1. **Walk the nine characteristics**
   In a one-hour session, read each characteristic aloud and ask stakeholders whether it matters. Note who cares and why.

2 **Mark relevance**
Label each characteristic high, medium, low or not applicable, and record a sentence of rationale for each decision.

3 **Drill into the high ones**
For high-relevance characteristics only, look at the sub-characteristics to find the precise concern, such as recoverability rather than reliability in general.

4 **Write scenarios**
Express each precise concern as one or two measurable scenarios with a stimulus, context and response measure.

5 **Prioritise and link**
Rank the scenarios and link each to the design decisions it drives, so the list stays short and alive.

WORKED EXAMPLE

Worked example: applying ISO/IEC 25010 to a mobile banking app

A mid-sized bank is replacing its mobile app and its back end. It has 1.2 million active app users; on the evening of payday about 25% of them log in between 18:00 and 20:00. The team walks the nine characteristics in an hour and records the outcome.

Peak load estimate	1,200,000 × 25% = 300,000 log-ins in 2 hours = 300,000 ÷ 7,200 s ≈ 42 log-ins per second on average; allow 3× for bursts ≈ 125 per second
Functional suitability: high	Interest, fees and balances must be correct to the penny. Scenario: every displayed balance reconciles with the core ledger within 5 seconds of a posted transaction.
Performance efficiency: high	Scenario: at 125 log-ins per second, the home screen with balances renders within 2 seconds for 95% of users on a 4G connection. Battery and data use on the phone also matter (resource utilisation).
Compatibility: medium	Must interoperate with the national faster-payments scheme and the open-banking APIs that regulation requires; must run on the two latest major versions of both mobile operating systems.
Interaction capability: high	Inclusivity and user error protection matter most. Scenario: a screen-reader user can complete a payment to a new payee without sighted help; the app warns before a payment to a payee name that does not match the account.
Reliability: high	Recoverability and fault tolerance. Scenario: if the card platform fails, balances and transfers between own accounts keep working; full service is restored within 30 minutes.
Security: high	Authenticity, non-repudiation and resistance. Scenario: every payment is authorised with strong customer authentication and recorded so that the bank can prove who approved it; 10,000 credential-stuffing attempts per minute are throttled without locking out genuine users.
Maintainability: medium	Modifiability and testability. Scenario: a new savings product can be launched in under 6 weeks without changing the payments component.
Flexibility: medium	Scalability. Scenario: the back end can grow to 3 million users by adding instances, with no redesign.
Safety: not applicable	The app cannot cause physical harm; risks to money are covered under security and reliability. Recorded explicitly so the decision is visible.
Outcome	Five characteristics rated high, three medium, one not applicable. Eleven scenarios in total, each with a number. The top three (security, reliability, performance at payday peak) become the quality goals on the first page of the architecture document.

The arithmetic converts 'lots of users on payday' into a design figure (about 125 log-ins per second at burst) that can be tested.

Sub-characteristics did their job: 'security' became authenticity, non-repudiation and resistance, each of which suggests different mechanisms.

Marking safety as not applicable is a decision, not an omission. If the bank later adds a feature that could cause harm, the record shows it needs revisiting.

Questions people ask

Do I have to cite ISO/IEC 25010 to do architecture properly?
No. It is valuable as shared vocabulary and as a checklist, and essential when contracts or regulators reference it. Many excellent teams use a lighter list. What matters is that the important qualities are identified, made measurable and prioritised.

Where does scalability belong: performance or flexibility?
In ISO/IEC 25010:2023 it is a sub-characteristic of flexibility, while capacity sits under performance efficiency. In practice the two are closely linked, and Section 2.4 treats them together. Do not let classification arguments delay writing the scenario.

Is 'observability' in the model?
Not by that name. Parts of it map to analysability (maintainability) and accountability (security). Many practitioners now treat it as a quality in its own right; Section 2.7 discusses it.

CHECK YOUR UNDERSTANDING

▼ **What were usability and portability renamed to in ISO/IEC 25010:2023?**

Usability became interaction capability and portability became flexibility.

▼ **Under which characteristic does ISO/IEC 25010:2023 place availability?**

Under reliability, alongside faultlessness, fault tolerance and recoverability.

▼ **Why is it useful to write 'not applicable' against a characteristic rather than leaving it blank?**

It records a deliberate decision with its rationale, so reviewers can see it was considered and the decision can be revisited if the system changes.

▼ **What is the main danger of turning a quality model into a requirements template?**

People fill every box with vague statements, effort spreads evenly over qualities of very unequal importance, and the few qualities that really decide the architecture get lost.

▼ **Try it: Walking ISO/IEC 25010 for a courier app, a colleague marks all nine characteristics high "to be safe". What is wrong with that? Name two you would mark high, one you would mark low, and give reasons.**

If everything is high, nothing guides design, and the model has become bureaucracy. Reliability is high: couriers on poor signal must not lose jobs, so look at recoverability. Interaction capability is high: riders need short, simple steps, and safety deserves a look for the same reason. Compatibility might be lower if the app talks only to the company's own services. Record a sentence of reasoning for each, then write scenarios only for the high ones.

SOURCES

- ISO/IEC 25010:2023 Product quality model (ISO catalogue page): https://www.iso.org/standard/78176.html
- ISO/IEC 25019:2023 Quality-in-use model (ISO catalogue page): https://www.iso.org/standard/78177.html
- Q42: ISO 25010 update 2023 (summary of changes): https://quality.arc42.org/articles/iso-25010-update-2023
- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/

Writing quality attribute scenarios that can be tested

BEFORE YOU READ THIS

This section builds on:

- 2.1 Functional and quality requirements
- 2.2 Quality models

Terms it uses from earlier: Quality attribute, Stakeholder, Production

THE IDEA IN BRIEF

Telling a builder 'make the roof strong' gives them nothing to check. Saying 'the roof must hold 50 centimetres of wet snow during a winter storm without leaking or sagging more than 2 centimetres' does. It names what happens (snow), the conditions (a storm), the part involved (the roof), what should happen (no leaks), and a number you can measure (2 centimetres). A quality attribute scenario does the same for software. It turns a wish like 'the website must be fast' into a statement precise enough to design for and test.

WHAT YOU NEED TO KNOW

- A quality attribute scenario has six parts: source of stimulus, stimulus, artifact, environment, response and response measure.
- General scenarios are system-independent templates for one quality; concrete scenarios fill in specific values for one system and serve as requirements.
- A scenario without a response measure, or with an average instead of a percentile for timing, cannot be tested reliably.
- Common mistakes include mixing several stimuli in one scenario, omitting the environment and writing a solution such as a technology choice in place of the need.
- Scenarios translate directly into acceptance tests and later into fitness functions that check the quality continuously.

Why scenarios

Section 2.2 gave you the vocabulary of qualities; this section gives you the grammar. A quality name on its own, such as performance or security, cannot be designed for or tested. Even a number on its own is not enough: '2 seconds' means nothing until you know for which operation, under what load, measured where and for what share of requests. The quality

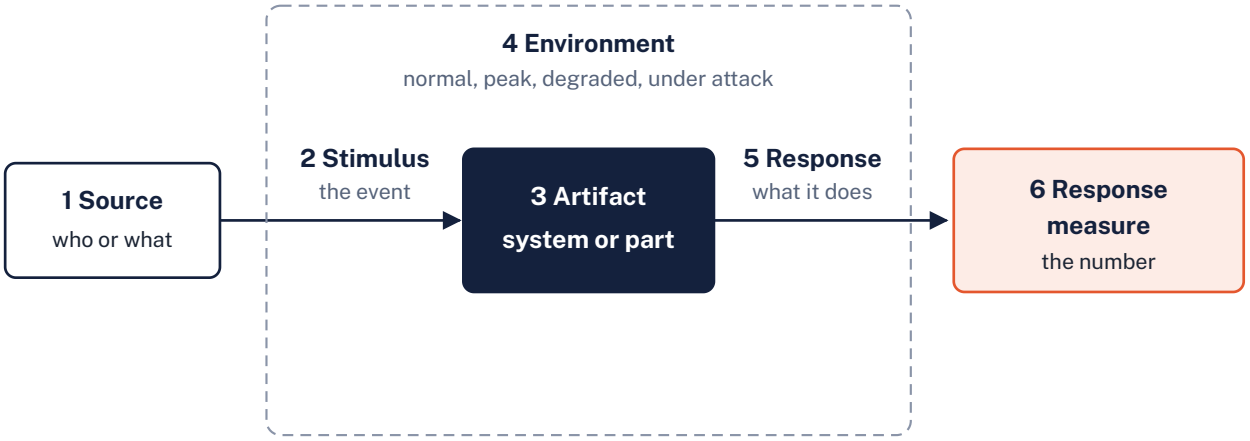
attribute scenario, developed at the Software Engineering Institute (SEI) and set out by Len Bass, Paul Clements and Rick Kazman in *Software Architecture in Practice*, solves this by forcing every quality requirement into the same six-part shape.

The idea is simple. Every quality shows itself when something happens to a system: a surge of users, a crash, a change request, an attack. A scenario names that event, the circumstances, the part of the system affected, what should happen and how you will know it happened well enough. It is short (usually one to three sentences), concrete, and testable. And because every quality uses the same shape, scenarios for performance, security and modifiability can sit side by side in one prioritised list.

The six parts

The six parts are the source of stimulus, the stimulus, the artifact, the environment, the response and the response measure. Learn them in that order, since together they tell a short story: *someone or something* does *something* to *part of the system* under *certain conditions*; the system *does something*, and we *measure it* like this. The stimulus and the response measure are the two parts people most often leave out, and they are the two that matter most.

Anatomy of a scenario



Example: 500 shoppers per second (source, stimulus) hit checkout (artifact) on Black Friday (environment); orders are confirmed (response) with p99 under 1.5 s and no lost orders (response measure).

A source produces a stimulus that reaches an artifact within an environment; the artifact's response is judged by a response measure.

The six parts in detail

Part	Meaning	Examples
Source of stimulus	The person, system or event that generates the stimulus.	Customers on mobile phones; an internal batch job; a developer; an attacker; a failing disk.
Stimulus	The event the system must respond to.	A burst of requests; a server crash; a change request; a login attempt with stolen credentials.
Artifact	The part of the system that is stimulated: the whole system or a specific component.	The checkout service; the order database; the pricing module; the deployment pipeline.
Environment	The conditions under which the stimulus arrives.	Normal operation; Black Friday peak; one data centre down; during a deployment; in development.
Response	What the system (or the people around it) does as a result.	Processes the order; fails over to the replica; the change is made, tested and deployed; the attempt is blocked and logged.
Response measure	How the response is measured, so the scenario can be tested.	p99 below 800 ms; service restored within 60 s with no lost orders; under 2 person-days; 100% of attempts blocked.

The environment deserves special care because the same stimulus can demand very different responses in different conditions. A crashed server during normal operation might require automatic failover within a minute; the same crash during a planned maintenance window might be acceptable with a manual recovery. A change request at design time (before release) costs far less than the same change once data has accumulated in production. Writing the environment explicitly stops people from comparing scenarios that are really about different situations.

The artifact matters for a similar reason. 'The system responds within 200 ms' hides which part is responsible. Naming the artifact (the search service, the API gateway, the mobile client) tells the team where the design work and the measurement must happen. For some qualities, such as modifiability and deployability, the artifact may be code, configuration or the delivery pipeline rather than a running component.

General and concrete scenarios

Bass, Clements and Kazman distinguish two kinds. A general scenario is system-independent: for one quality attribute it lists the typical values each part can take. For availability, for example, the sources might be internal or external, people, hardware, software or infrastructure; stimuli might be a crash, an omission, incorrect timing or an incorrect response; responses might be detecting, recording, recovering, masking or degrading; and measures might be time to detect, time to repair, availability percentage or the proportion of requests handled.

A concrete scenario picks specific values for one particular system: 'the primary order database crashes during normal operation; the replica takes over and writes resume within 60 seconds with no committed order lost'. General scenarios are templates and checklists; concrete scenarios are requirements. You use the general scenario to make sure you have considered the right sources, stimuli and measures, then write the handful of concrete scenarios that matter for your system.

From vague statements to testable scenarios

Most quality requirements start life as adjectives. The skill is to ask the questions that turn each adjective into a scenario: which operation? caused by whom? how often or how much? under what conditions? what should happen? what number tells us it worked? The table shows typical before-and-after pairs. Notice that the 'after' versions sometimes reveal that the stakeholder meant several different things, which then become several scenarios.

Before and after

Vague statement	Testable scenario
The site must be fast.	During the weekday evening peak of 300 searches per second, product search returns results within 400 ms for 95% of requests and 1 s for 99%, measured at the load balancer.
The system must be highly available.	If one of the three application servers fails during normal operation, the booking service continues to accept bookings; no confirmed booking is lost and users see errors for no more than 30 seconds.
The system must be secure.	When an attacker submits 10,000 login attempts per minute using leaked credentials, the system blocks the source after 20 failures, alerts the security team within 5 minutes, and genuine users at other addresses can still log in.
The code must be easy to change.	When the business asks for a new delivery option, one developer familiar with the code can implement, test and deploy it in no more than 3 days, changing only the delivery module and its configuration.
The app must be user-friendly.	A first-time user with no training can register and place an order in under 4 minutes; 90% of test participants succeed at the first attempt.
Deployments must be easy.	A developer can release a change to the ordering service to production in under 30 minutes after merge, with no downtime, and roll it back within 5 minutes if error rates rise.

IF YOU ARE NEW TO THIS

If you are new to this: where do the numbers come from?

People often hesitate to write a number because they do not know the 'right' one. Start from evidence: current traffic logs, the old system's monitoring, competitors' behaviour, business targets ('we plan to double orders by next year') and research on user patience. Then propose a figure and let stakeholders react. 'Is 400 milliseconds for 95% of searches acceptable?' produces a far better conversation than 'how fast should it be?'. A number agreed today can be revised tomorrow; a requirement with no number cannot even be discussed.

Scenario examples across six qualities

The examples below use one online shop so you can see how different qualities look in the same format. Each is written compactly with all six parts labelled; in practice teams often write them as one or two sentences.

Concrete scenarios for an online shop

Quality	Scenario (source, stimulus, artifact, environment, response, measure)
Performance	Source: logged-in customers. Stimulus: 2,000 order submissions per minute. Artifact: checkout service. Environment: Friday evening peak. Response: orders validated, stored and confirmed. Measure: p95 ≤ 1.5 s, p99 ≤ 3 s, zero orders lost.
Availability	Source: internal hardware. Stimulus: the primary database node crashes. Artifact: order database. Environment: normal operation. Response: replica promoted automatically, in-flight requests retried. Measure: writes resume within 60 s; no committed order lost (RPO = 0).
Modifiability	Source: product owner. Stimulus: request to add a new payment method. Artifact: payment component. Environment: design time, after initial release. Response: change made, tested and deployed. Measure: ≤ 5 person-days; no change outside the payment component; no regression in existing methods.
Security	Source: a logged-in customer acting maliciously. Stimulus: attempts to read another customer's orders by changing an ID in the URL. Artifact: orders API. Environment: normal operation. Response: request denied and logged with the caller's identity. Measure: 100% of such requests rejected; alert raised within 1 minute after 5 attempts.
Usability	Source: first-time customer. Stimulus: wants to change a delivery address after ordering. Artifact: web and mobile front ends. Environment: runtime, no help available. Response: completes the change unaided. Measure: 90% of usability-test participants succeed within 2 minutes.
Deployability	Source: developer. Stimulus: a merged change to the catalogue service. Artifact: deployment pipeline and catalogue service. Environment: business hours, normal load. Response: built, tested and rolled out progressively. Measure: in production within 45 minutes; zero downtime; automatic rollback within 5 minutes if error rate exceeds 1%.

Two features of these examples are worth noticing. First, scenarios are not only about running software. Modifiability and deployability scenarios have developers and product owners as sources, and their measures are effort, elapsed time and the scope of change. Second, good measures often combine a speed number with a correctness number: 'within 60 seconds' and 'no committed order lost'. A fast failover that loses data is not what the business wanted.

Common mistakes

Reviewing hundreds of scenarios written by teams reveals the same mistakes again and again. Each makes a scenario untestable or misleading.

Mistakes and how to fix them

Mistake	Example	Fix
No response measure	'Under peak load the checkout responds quickly.'	Add a number and a statistic: 'p95 ≤ 1.5 s, p99 ≤ 3 s'.
Average instead of percentile	'Average response time under 300 ms.'	Averages hide slow outliers; specify percentiles (Section 2.4).
Several stimuli in one scenario	'When load doubles or a server fails or a deployment runs, the system stays fast.'	Split into three scenarios, each with its own environment and measure.
Solution disguised as requirement	'The system uses Redis to cache product pages.'	State the need ('product pages load in under 300 ms for 95% of views at 1,000 views per second') and leave the cache as a design option.
Missing environment	'The service recovers within 5 minutes.'	Say from what and when: 'after a single zone outage during normal operation'.
Unmeasurable words	'Gracefully', 'seamlessly', 'robustly', 'user-friendly'.	Replace each with an observable behaviour and a number.
Impossible absolutes	'The system never goes down.' '100% of attacks blocked.'	Use realistic targets and say what happens when they are missed.

WATCH OUT

Solutions disguised as requirements

This is the most damaging mistake because it looks like progress. 'Use Kubernetes', 'use microservices', 'use a message queue' are design decisions. Writing them as requirements skips the step where the team asks whether they are the best way to achieve the quality, and it removes the evidence needed to revisit the choice later. If a stakeholder insists on a technology, ask what problem it solves; the answer is the real scenario. If the technology is genuinely mandated by policy, record it as a constraint with its source.

DECISION GUIDE

Decision guide: is this scenario ready?

Run each draft scenario through these questions before it goes into the architecture document or the backlog.

Q1 **Does it describe a single stimulus with a named source?**

- **Yes** Continue.
- **No** Split it, or name who or what causes the event.

Q2 **Does it state the environment (normal, peak, degraded, during deployment, at design time)?**

- **Yes** Continue.
- **No** Add the conditions; otherwise the measure cannot be interpreted.

Q3 **Does it name the artifact, the part of the system responsible?**

- **Yes** Continue.
- **No** Name the component, interface or pipeline so the team knows where to design and measure.

Q4 **Is the response an observable behaviour rather than a technology choice?**

- **Yes** Continue.
- **No** Rewrite the need and move the technology into a design decision or a constraint.

Q5 **Does the response measure contain a number, a unit and, for timing, a percentile?**

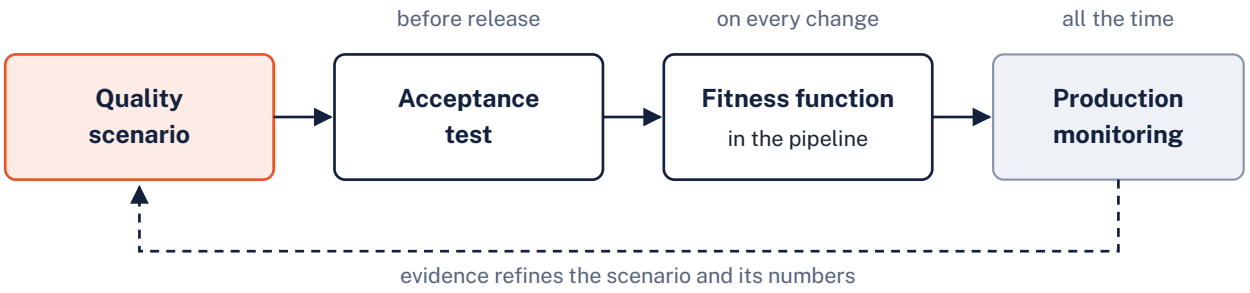
- **Yes** The scenario is testable: add a priority and an owner.
- **No** Add the measure; without it the scenario is a wish.

From scenarios to acceptance tests and fitness functions

A well-written scenario is already most of a test. The source and stimulus tell you what load or event to generate, the environment tells you how to set up the test, the artifact tells you what to observe and the response measure gives the pass condition. A performance scenario becomes a load test; an availability scenario becomes a failover drill or a chaos experiment; a modifiability scenario becomes a review of how much code a sample change touched and how long it took; a security scenario becomes a penetration test case.

The next step is to keep checking. A fitness function, a term popularised by Neal Ford, Rebecca Parsons and Patrick Kua in *Building Evolutionary Architectures*, is any automated or repeatable check of how well the system meets an architectural goal. Run in the delivery pipeline or against production telemetry, it turns a scenario from a one-off acceptance test into a permanent guard: the build fails, or an alert fires, when the quality drifts. Chapter 11 covers fitness functions in depth in Fitness functions and evolutionary architecture.

The life of a scenario



A scenario drives an acceptance test, then a fitness function that runs on every change, and production evidence feeds back to refine the scenario.

```
# A performance scenario expressed as an automated check (illustrative)
# Scenario: at 300 searches/s, p95 ⩽ 400 ms and p99 ⩽ 1000 ms.
import statistics

def check_search_latency(latencies_ms, p95_target=400, p99_target=1000):
    cuts = statistics.quantiles(latencies_ms, n=100)  # 99 cut points
    p95, p99 = cuts[94], cuts[98]
    assert p95 ⩽ p95_target, f"p95 {p95:.0f} ms exceeds {p95_target} ms"
    assert p99 ⩽ p99_target, f"p99 {p99:.0f} ms exceeds {p99_target} ms"
    return p95, p99

# latencies_ms comes from a load test that replays 300 searches per second
# for ten minutes against a production-like environment.
```

A performance scenario turned into a simple automated check of p95 and p99 latency.

How professionals use scenarios

In practice, teams rarely write long tables of six-part scenarios for everything. They write a small number, often ten to thirty for a substantial system, for the qualities that are architecturally significant, and they keep them where everyone can see them: in the architecture document (arc42 section 10 asks for exactly this), in the backlog as acceptance criteria, or in a shared quality board. Some teams use a shorter form; Gernot Starke's Q42

project, for instance, recommends a compact structure of context, trigger and acceptance criteria. The shape matters less than the discipline: every important quality has a stimulus, a context and a number.

Scenarios also power evaluation methods. The Quality Attribute Workshop and the Architecture Tradeoff Analysis Method (ATAM), both from the SEI, collect and prioritise scenarios with stakeholders and then test the architecture against the most important ones. Section 2.8 shows how scenarios are gathered and ranked in a utility tree.

CHECK YOUR UNDERSTANDING

▼ Name the six parts of a quality attribute scenario in order.

Source of stimulus, stimulus, artifact, environment, response and response measure.

▼ What is wrong with 'the system uses a CDN so that pages load fast'?

It states a solution, not a need, and has no measure. The requirement should describe the load, conditions and target response time; the CDN is one possible design to meet it.

▼ Who is typically the source of stimulus in a modifiability scenario?

A person who wants a change, such as a product owner, a developer or a regulator; the stimulus is the change request.

▼ How does a scenario differ from a fitness function?

A scenario is a statement of a requirement; a fitness function is an automated or repeatable check that measures whether the system meets it, run continuously.

▼ Try it: Turn "the ticket site must stay fast on sale day" into a six-part quality attribute scenario, inventing sensible figures. Name each part. Then state one reason why the original sentence could never be tested.

Source: fans' browsers. Stimulus: 2,500 people arrive each second. Artifact: the seat-selection and purchase flow. Environment: on-sale peak, all zones healthy. Response: each page request is served, or the fan is placed in a waiting room. Response measure: p99 page response under 2 seconds. The original could not be tested because "fast" has no number, and it names no load or conditions, so nobody could say whether it had passed.

SOURCES

- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/

- arc42 section 10: Quality requirements and quality scenarios: https://docs.arc42.org/section-10/
- Q42: How to specify quality requirements: https://quality.arc42.org/articles/specify-quality-requirements

SECTION 2.4 OF 2.8 · INTERMEDIATE · 27 MIN READ · LAST REVIEWED NOVEMBER 2026

Performance and scalability: latency, throughput, percentiles and load

BEFORE YOU READ THIS

This section builds on:

- 0.2 Data and algorithms
- 0.4 How computers talk
- 0.7 Servers and the cloud

Terms it uses from earlier: Server, Cache, Database

THE IDEA IN BRIEF

Picture a supermarket. How long you spend at the till is the latency; how many shoppers the tills serve in an hour is the throughput. Your time at the till is partly the scanning (the work itself) and partly the queue in front of you. When the tills are only half busy you barely queue; when they are almost always busy, queues grow very long very quickly, even though each scan takes the same time. Opening more tills helps, until everyone has to share one manager for refunds. Performance and scalability are about exactly these effects in software.

WHAT YOU NEED TO KNOW

- Latency is how long one piece of work takes; throughput is how much work is completed per unit of time. A performance requirement should state both, with the load at which they apply.
- Response time is service time plus waiting time (plus network time); under load, waiting time usually dominates.
- Specify and monitor percentiles such as p95 and p99, not averages; in fan-out systems a rare slow response at one server becomes common for users (Dean and Barroso, 2013).
- Little's law, $L = \lambda \times W$, links request rate, response time and concurrency, and is the basis of sizing pools and limits.
- In the simplest queue model, M/M/1, mean response time is $S \div (1 - \rho)$, so it doubles between 80% and 90% utilisation; other queues differ in the detail but show the same cliff, so keep critical resources well below saturation.
- Amdahl's law limits speed-up to $1 \div s$ for serial fraction s; the Universal Scalability Law adds coherency costs that make capacity peak and then fall.

Two different questions: how fast, and how much

Performance is the quality people mention first and specify worst. It covers two different questions. Latency asks how long one piece of work takes: how many milliseconds between tapping 'Pay' and seeing 'Payment accepted'. Throughput asks how much work the system completes per unit of time: 1,200 payments per second, 40,000 orders per hour. The two are related but not interchangeable. A coach carrying 50 people from London to Oxford in 100 minutes has worse latency and much better throughput than a sports car carrying two people in 60 minutes.

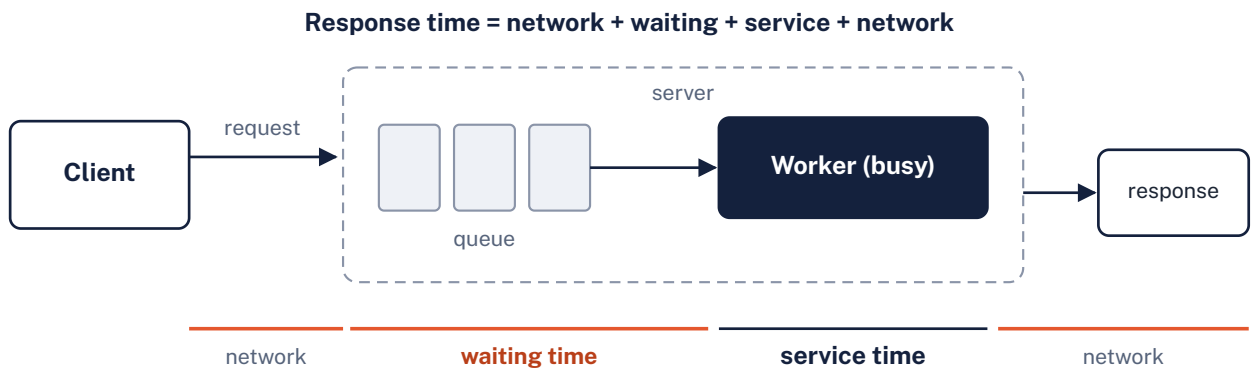
In software the same tension appears everywhere. Batching database writes improves throughput but makes each individual write wait for its batch. Adding a cache improves both for repeated reads but does nothing for the first. A good performance requirement therefore says which question it is about, and usually states both: 'at 1,200 payments per second (throughput), 99% complete within 800 ms (latency)'. ISO/IEC 25010 files these under performance efficiency as time behaviour, resource utilisation and capacity (Section 2.2).

Response time = service time + waiting time

When you measure latency from the client's side you get the response time: everything that happened between sending the request and receiving the answer. It is useful to split it into parts. The service time is the time a resource actually spends working on your request: parsing it, running the query, rendering the result. The waiting time is the time your request spends in a queue because the resource is busy with someone else's. Network transfer time is added on top, in both directions.

The distinction matters because the two parts respond to different remedies. Service time falls when you make the work itself cheaper: a better index, a smaller payload, a faster algorithm. Waiting time falls when you add capacity or reduce load, and it is often much larger than the service time when a system is busy. Many 'slow database' investigations end with the discovery that each query takes 5 ms but waits 300 ms for a free connection in the pool.

Where the time goes



The response time a client sees includes network transfer, waiting in a queue for a busy worker, and the service time itself.

Percentiles: why averages mislead

Response times are never a single number; they form a distribution, and it is almost always skewed. Most requests are quick, and a minority are slow because they hit a cold cache, a garbage-collection pause, a lock, a retry or an overloaded neighbour. The arithmetic mean blends the two groups into a figure that describes nobody. Professionals use percentiles instead. The 50th percentile (p50, the median) is the value half of requests beat; p95 is the value 95% of requests beat; p99 and p99.9 describe the slowest 1% and 0.1%.

The slow end of the distribution is called the tail, and it matters more than it seems. The users in the tail are often your best customers: the ones with the biggest baskets, the longest histories and the most data. A user who loads 40 resources to view one page experiences the slowest of those 40, not the average. And the chance that a 40-resource page hits at least one p99 request is $1 - 0.99^{40} \approx 33\%$: a 'rare' 1% event affects a third of page views.

WORKED EXAMPLE

Worked example: the mean that hides the problem

A product-search API logs 1,000 response times in one minute: 900 at 80 ms, 80 at 150 ms, 15 at 1,200 ms (a cold cache) and 5 at 4,000 ms (a timeout and retry).

Sum of all times	900 × 80 + 80 × 150 + 15 × 1,200 + 5 × 4,000 = 72,000 + 12,000 + 18,000 + 20,000 = 122,000 ms
Mean	122,000 ÷ 1,000 = 122 ms. Looks healthy against a 200 ms target.
p50 (median)	The 500th fastest request: 80 ms
p95	The 950th fastest request falls inside the 150 ms group (requests 901 to 980): 150 ms
p99	The 990th falls inside the 1,200 ms group (requests 981 to 995): 1,200 ms
p99.9	The 999th is in the 4,000 ms group (996 to 1,000): 4,000 ms
Who is affected	20 requests in 1,000 (2%) took over a second. At 300 searches per second that is 6 users every second, or about 21,600 an hour.

The mean of 122 ms passes a 200 ms target while 2% of users wait more than a second. A scenario written as 'average below 200 ms' would pass this system.

Writing the requirement as 'p95 ≤ 200 ms and p99 ≤ 500 ms' fails it, correctly, and points the team at the cold-cache and retry paths.

Percentiles cannot be averaged. The p99 across ten servers is not the mean of their ten p99 values; it must be computed from the combined data, or approximated from merged histograms.

```
# Why the mean misleads: 1,000 response times from one minute of traffic
import statistics

times_ms = [80] * 900 + [150] * 80 + [1200] * 15 + [4000] * 5

mean = statistics.mean(times_ms)
cuts = statistics.quantiles(times_ms, n=1000, method="inclusive")
p50, p95, p99, p999 = cuts[499], cuts[949], cuts[989], cuts[998]
print(f"mean {mean:.0f} ms, p50 {p50:.0f}, p95 {p95:.0f}, p99 {p99:.0f}, p99.9
{p999:.0f}")
# mean 122 ms, p50 80, p95 150, p99 1200, p99.9 4000
```

The same calculation in a few lines of Python; the mean looks fine while p99 and p99.9 reveal the problem.

Tail latency at scale

The tail becomes the main event in large systems that use fan-out, where one user request is split into many parallel requests to other servers and the answer must wait for all of them. A search engine, a social feed or a recommendations page may consult tens or hundreds of back-end shards for every query. Jeffrey Dean and Luiz André Barroso of Google analysed this in *The Tail at Scale* (Communications of the ACM, 2013).

Their illustration is simple. Suppose each server answers in 10 ms typically but takes 1 second for one request in 100. A user request that waits for one server is slow 1% of the time. A user request that waits for 100 such servers is slow whenever any one of them is slow: $1 - 0.99^{100} \approx 63\%$ of the time. A rare event at one server becomes the common case for the user. Dean and Barroso report measurements from a Google service in which the 99th-percentile latency for a single leaf request was 10 ms, but the 99th percentile for waiting on all leaf requests was 140 ms.

Because you cannot eliminate every source of variability (shared hardware, background jobs, garbage collection, network queues), they argue for tail-tolerant designs, much as fault-tolerant designs accept that hardware fails. Their techniques include hedged requests (send a second copy to another replica if the first has not answered within, say, the 95th-percentile time, and use whichever returns first), tied requests that cancel each other, finer-grained partitions that can be rebalanced, temporarily excluding slow machines, and returning a good-enough result when most but not all shards have answered. In one benchmark they describe, hedging after 10 ms cut the 99.9th-percentile latency from 1,800 ms to 74 ms while sending only about 2% more requests.

Little's law: how many requests are in flight

Little's law, proved by John Little in 1961, is the most useful single formula in capacity planning. For any stable system, the average number of items inside it (L) equals the average arrival rate (λ, lambda) multiplied by the average time each item spends inside (W): $\mathbf{L = \lambda \times W}$. It holds regardless of the arrival pattern or the order of service, provided the system is stable (on average, as many items leave as arrive) and you use consistent units.

For software, L is the number of requests in progress at once (the concurrency), λ is the request rate and W is the response time. If a service receives 200 requests per second and each spends 0.25 seconds inside, then on average $200 \times 0.25 = 50$ requests are in progress at any moment. That number drives thread pools, connection pools, memory and licence counts. The law also works backwards: if you can hold at most 50 requests at once and each takes 0.25 s, you cannot sustain more than $50 \div 0.25 = 200$ requests per second.

WORKED EXAMPLE

Worked example: sizing a database connection pool with Little's law

A ride-hailing app's trip service handles 1,800 requests per second at the evening peak. Each request makes database calls that keep a connection busy for 40 ms in total. Each of the 6 service instances has a pool of 10 connections. Is that enough?

Arrival rate λ	1,800 requests per second
Time holding a connection W	40 ms = 0.040 s
Connections in use L = λ × W	1,800 × 0.040 = 72 connections busy on average
Connections available	6 instances × 10 = 60
Verdict	72 > 60. Requests will queue for connections, waiting time will grow without limit at peak, and response times will climb until timeouts start.
Target with headroom	Aim to keep the pool at most about 70% utilised: 72 ÷ 0.7 ≈ 103 connections, so about 18 per instance
Check the database	103 connections must also be acceptable to the database server; if not, reduce W (faster queries, fewer round trips) or the number of database calls per request
What if queries get faster?	Cutting W to 25 ms gives L = 1,800 × 0.025 = 45 connections, which fits in the existing 60 at 75% utilisation

Little's law converts a throughput target and a latency into a concurrency figure that can be compared directly with a configured limit.

It gives averages. Bursts push the instantaneous number higher, which is why the example adds headroom rather than sizing to exactly 72.

Utilisation and queueing: why systems fall off a cliff

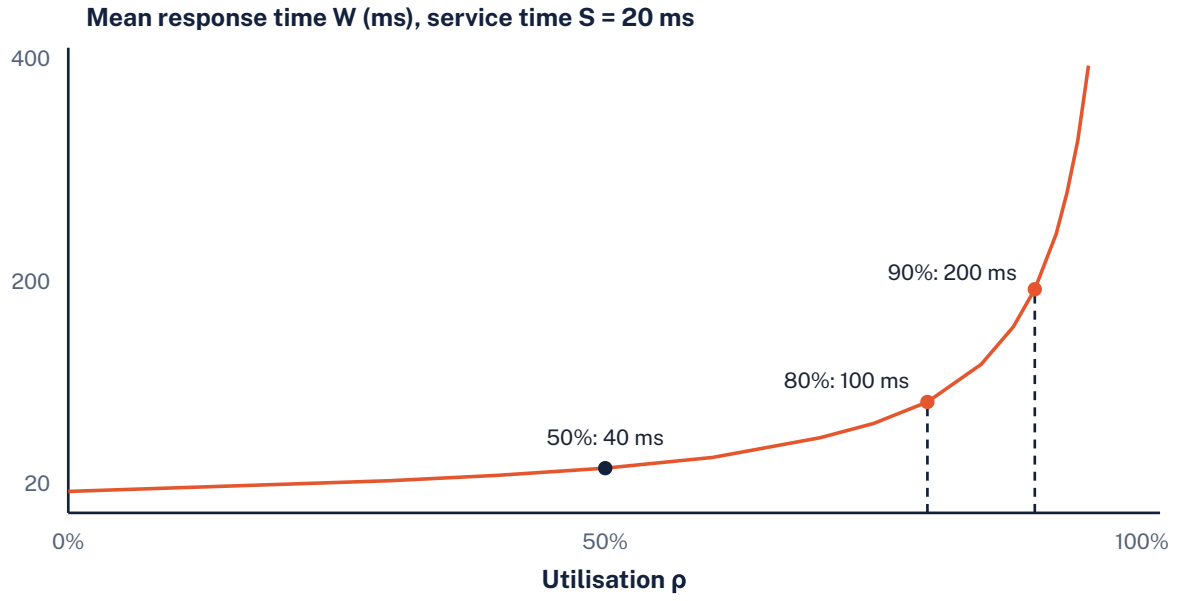
Utilisation (ρ, rho) is the fraction of time a resource is busy. A server that receives 40 requests per second, each needing 20 ms of service, is busy 40 × 0.020 = 0.8 of the time: 80% utilised. Intuition suggests that a resource 80% busy should be only a little slower than one 50% busy. Queueing theory says otherwise, because requests do not arrive evenly; they bunch together, and when they bunch, the later ones wait.

The simplest model is the M/M/1 queue: random arrivals, random service times and a single server. Its mean response time is **W = S ÷ (1 − ρ)**, where S is the mean service time. At 50% utilisation, W = 2S. At 80%, W = 5S. At 90%, W = 10S. At 99%, W = 100S. The denominator shrinks towards zero as utilisation approaches 100%, so response time grows without limit. Real systems are not exactly M/M/1, and multi-server pools behave better at the same utilisation, but the shape is universal: flat, then a knee, then a cliff.

M/M/1 queue: mean response time for a 20 ms service time

Utilisation ρ	W = S ÷ (1 − ρ)	Mean response time	Of which waiting
50%	20 ÷ 0.5	40 ms	20 ms
70%	20 ÷ 0.3	67 ms	47 ms
80%	20 ÷ 0.2	100 ms	80 ms
90%	20 ÷ 0.1	200 ms	180 ms
95%	20 ÷ 0.05	400 ms	380 ms
99%	20 ÷ 0.01	2,000 ms	1,980 ms

The utilisation cliff



In an M/M/1 queue, mean response time doubles between 80% and 90% utilisation and keeps rising steeply towards 100%.

The practical lessons are direct. Do not plan to run critical resources near 100% busy; leave headroom so that bursts do not push you over the knee. Watch utilisation and queue length together (Brendan Gregg's USE method checks utilisation, saturation and errors for every resource). And remember that the cliff makes small changes dangerous: at 90% utilisation, a rise in traffic of only about 6% takes you to 95% and doubles the mean response time. Chapter 9 covers queues, back pressure and capacity planning in Asynchrony, queues and back pressure.

IF YOU ARE NEW TO THIS

Step by step: why a busy server makes requests wait

The step most people trip over here is why waiting grows so steeply as a resource gets busier, even though each request needs no more work than before. Follow one server as its traffic rises.

- Each request needs 20 ms of work on average: its service time. At 40 requests a second the server is busy 40 × 0.020 = 0.8 of the time, so its utilisation is 80%.
- If requests arrived evenly, one every 25 ms, each would find the server free and take just 20 ms.
- Real requests arrive at random, so they bunch. If three arrive together, the first is served at once, the second waits about 20 ms and the third about 40 ms. A gap may follow in which the server sits idle, and idle time cannot be saved up for the next bunch.
- At 50% busy there are plenty of gaps, so queues soon clear. At 80% there are fewer, and one bunch's queue is often still there when the next arrives. Near 100% there is almost no spare time, and queues grow very long.
- The section's table for a single server with random arrivals (the M/M/1 queue) shows the effect: a mean response time of 40 ms at 50% busy, 100 ms at 80% and 200 ms at 90%. Going from 40 to 45 requests a second takes the server from 80% to 90% busy and doubles the mean response time.
- **The lesson:** the work itself has not slowed; the extra time is all queueing. Keep critical resources well below full, because near full a small rise in traffic causes a large rise in response time.

Scalability: load, elasticity, scaling up and out

Scalability is the ability to handle more load by adding resources, ideally with capacity growing in proportion and without redesign. It only means something once you say what 'load' is. Load parameters differ between systems: requests per second for an API, concurrent connections for a chat service, the ratio of reads to writes for a social network, data volume for an analytics platform, or the number of followers of the most-followed account for a feed. A scalability

scenario names the load parameter, the growth, and the response that must be preserved: 'when daily orders grow from 50,000 to 500,000, p95 checkout latency stays below 1.5 s by adding instances, with cost per order no more than 20% higher'.

Elasticity adds time: the ability to add and remove capacity automatically as load rises and falls, so that a ticketing site can grow for a sale and shrink afterwards. It is a large part of the economic case for cloud platforms. There are two basic ways to add capacity. Vertical scaling (scaling up) gives one machine more CPU, memory or faster storage; it is simple, needs no change to the software, and runs into hard limits and rising prices. Horizontal scaling (scaling out) adds more machines and spreads the work; it has no fixed ceiling but requires the software to be designed for it, with stateless instances, load balancing and partitioned data. Chapter 9 covers the strategies in detail in Scaling strategies: vertical, horizontal and the scale cube.

Amdahl's law: the limit of parallelism

Adding processors or nodes only helps with work that can be done in parallel. Amdahl's law, from Gene Amdahl's 1967 paper on the limits of parallel computing, captures this. If a fraction s of the work is serial (it must be done one step at a time) and the rest, 1 − s, can be split perfectly across N processors, the best possible speed-up is **speed-up(N) = 1 ÷ (s + (1 − s) ÷ N)**. As N grows without limit, the speed-up approaches 1 ÷ s. With 10% serial work, no number of processors can make the job more than ten times faster.

WORKED EXAMPLE

Worked example: a nightly billing job

A telecoms company's nightly billing run takes 8 hours on one server. Profiling shows 10% of the time is serial (loading tariffs, then writing a single summary file); the other 90% rates calls independently per customer. The run must finish in under 1.5 hours. How many servers are needed?

Serial fraction s	0.10; parallel fraction 0.90
Speed-up needed	8 h ÷ 1.5 h ≈ 5.33
N = 4	1 ÷ (0.10 + 0.90 ÷ 4) = 1 ÷ 0.325 ≈ 3.08; run time 8 ÷ 3.08 ≈ 2.6 h
N = 8	1 ÷ (0.10 + 0.1125) = 1 ÷ 0.2125 ≈ 4.71; run time ≈ 1.70 h
N = 16	1 ÷ (0.10 + 0.05625) = 1 ÷ 0.15625 = 6.4; run time = 1.25 h
Smallest N that meets the target	Need 0.10 + 0.90 ÷ N ≤ 1 ÷ 5.33 = 0.1875, so N ≥ 0.90 ÷ 0.0875 ≈ 10.3: 11 servers give 1 ÷ (0.10 + 0.0818) ≈ 5.50, run time ≈ 1.45 h
N = 64	1 ÷ (0.10 + 0.0141) ≈ 8.77; run time ≈ 0.91 h
Limit as N grows	1 ÷ 0.10 = 10; run time never below 0.8 h
Alternative	Halve the serial part (s = 0.05), for example by loading tariffs in parallel. Then 8 servers give 1 ÷ (0.05 + 0.11875) ≈ 5.93, run time ≈ 1.35 h: the target met with 8 servers instead of 11.

Going from 16 to 64 servers (four times the cost) cuts the run only from 1.25 h to 0.91 h. Returns diminish quickly.

Reducing the serial fraction is often far cheaper than adding hardware. Architects look for the hidden serial parts: a single lock, one database row everyone updates, a central coordinator.

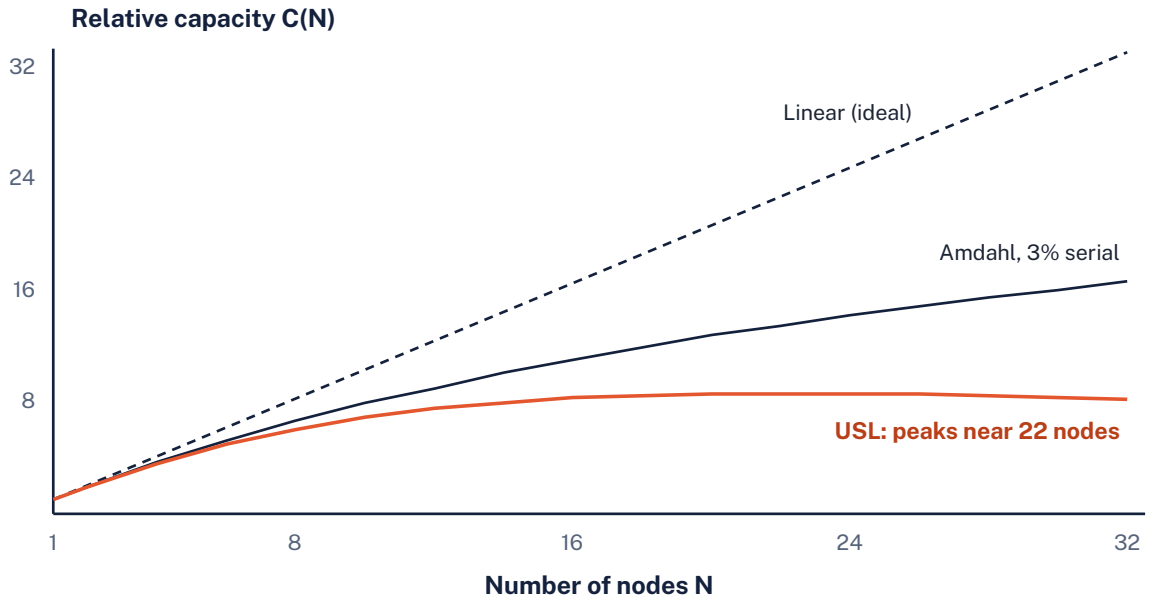
The Universal Scalability Law

Amdahl's law assumes that extra processors never make things worse; the speed-up just levels off. Real distributed systems often do get worse: past some point, adding nodes reduces throughput. Neil Gunther's Universal Scalability Law (USL) models this with two penalties. Contention (α, alpha) is the cost of queueing for shared resources, the same effect as Amdahl's

serial fraction. Coherency (β, beta) is the cost of keeping data consistent between nodes: every node must exchange information with every other, so this cost grows roughly with the square of the number of nodes.

The relative capacity is **C(N) = N ÷ (1 + α(N − 1) + βN(N − 1))**. With β = 0 it reduces to Amdahl's law. With any β above zero, capacity reaches a peak at about N = √((1 − α) ÷ β) and then falls. For α = 0.03 and β = 0.002, the peak is near √(0.97 ÷ 0.002) ≈ 22 nodes, at a capacity of about 8.6 times one node; at 32 nodes capacity has dropped back to about 8.2. Gunther describes the model in *Guerrilla Capacity Planning* (2007). Its practical value is that α and β can be fitted to a handful of load-test measurements, which lets you predict where scaling will stop paying off before you buy the hardware.

Linear, Amdahl and USL scaling



Ideal scaling is linear; Amdahl's law flattens because of serial work; the USL also includes coherency costs, so capacity peaks and then declines.

Tactics for performance: an overview

Bass, Clements and Kazman group performance tactics into two families. The first is to control resource demand: reduce how much work arrives or how much each piece costs. The second is to manage resources: make the available resources work harder or add more of them. Most real designs combine several tactics; each also has costs elsewhere, which is why they are architectural decisions.

Performance tactics and where the course covers them

Tactic	What it does	Typical cost or risk	More in
Reduce demand: rate limiting, admission control, waiting rooms	Caps or smooths incoming load so the core stays below its knee	Some users are delayed or refused	Chapter 9: queues and back pressure
Reduce work per request: efficient algorithms, smaller payloads, fewer round trips	Cuts service time	Developer effort; sometimes less flexible code	Chapter 0: data and algorithms
Prioritise and schedule	Serves important work first (paid orders before reports)	Low-priority work may starve	Chapter 9: resilience patterns
Caching	Keeps results close to the user to avoid repeated work	Stale data, invalidation complexity, memory cost	Chapter 7: caching
Concurrency and asynchrony	Processes independent work in parallel; takes slow work off the request path	Harder reasoning, ordering and consistency issues	Chapter 6: communication
Replication	Adds copies of services or data to spread reads and load	Consistency between copies; cost	Chapter 7: replication and partitioning
Partitioning (sharding)	Splits data and work so each node handles a share	Cross-partition queries and rebalancing are hard	Chapter 7: replication and partitioning
Add resources: scale up or out	Raises capacity	Cost; limited by serial and coherency work	Chapter 9: scaling strategies

WATCH OUT

Common mistakes

- Specifying or monitoring averages instead of percentiles, and averaging percentiles across servers.
- Load-testing at a steady rate when real traffic arrives in bursts; queues form in the bursts.
- Planning to run a critical resource at 90% utilisation or more, leaving no room for spikes.
- Assuming more servers always means more capacity, without measuring the serial and coherency costs.
- Optimising service time while ignoring waiting time, which is usually the larger part under load.
- Measuring only on the server side, when users experience network time and client rendering as well.

CHECK YOUR UNDERSTANDING

▼ A service receives 500 requests per second and each takes 80 ms. How many are in progress on average?

By Little's law, L = 500 × 0.08 = 40 requests in progress.

▼ Using the M/M/1 formula, what is the mean response time of a server with a 10 ms service time at 75% utilisation?

W = 10 ÷ (1 − 0.75) = 40 ms, of which 30 ms is waiting.

▼ Why is p99 usually a better requirement measure than the mean?

Response-time distributions are skewed; the mean can look healthy while a meaningful share of users suffer very slow responses, and p99 exposes that tail.

▼ If 20% of a job is serial, what is the maximum speed-up from adding processors?

1 ÷ 0.2 = 5 times, however many processors are added.

▼ **Try it: An order service handles 2,400 requests per second, each keeping a database connection busy for 30 ms. It has 8 instances with 8 connections each. Is that enough? What pool size would you set per instance, aiming for 70% use?**

By Little's law, 2,400 x 0.030 = 72 connections busy on average, and only 8 x 8 = 64 exist. So requests queue for connections and waiting time climbs at peak. For 70% use you need 72 / 0.7 = about 103 connections, or 103 / 8 = about 13 per instance, 104 in all. Check that the database can handle that many, and cut the 30 ms if it cannot.

SOURCES

- Dean and Barroso, The Tail at Scale (Google Research): https://research.google/pubs/pub40801/
- Neil Gunther: How to quantify scalability (the USL): https://www.perfdynamics.com/Manifesto/USLscalability.html
- Google SRE book: Monitoring distributed systems (latency and percentiles): https://sre.google/sre-book/monitoring-distributed-systems/
- Brendan Gregg: The USE method: https://brendangregg.com/usemethod.html

SECTION 2.5 OF 2.8 · INTERMEDIATE · 22 MIN READ · LAST REVIEWED NOVEMBER 2026

Availability and reliability: nines, MTBF, MTTR and fault tolerance

BEFORE YOU READ THIS

This section builds on:

- 0.7 Servers and the cloud
- 2.2 Quality models
- 2.3 Quality attribute scenarios

Terms it uses from earlier: Server, Load balancer, Component, Database, Data centre

THE IDEA IN BRIEF

Think of a corner shop. Availability is how often the door is open when you arrive. Reliability is whether, once you are inside, the till works all the way through your purchase. Shops stay open by having a spare till, a second member of staff and a quick way to notice and fix problems. Software is the same: architects decide how much spare capacity to keep, how to spot failures quickly and how fast to recover, and they express the goal as a number such as 99.9% of the time.

WHAT YOU NEED TO KNOW

- Availability is the fraction of time or requests for which the service is ready; reliability is the probability of running without failure for a stated period; they are different and a system can score well on one and badly on the other.
- A fault is a cause, an error is wrong internal state and a failure is a deviation users can observe; tactics prevent faults, detect errors or stop errors becoming failures.
- Allowed downtime = (1 - availability) x period: 99.9% is about 8.76 hours a year or 43.2 minutes a month; 99.99% is about 52.6 minutes a year.
- A = MTBF / (MTBF + MTTR), so cutting repair time through automation and redundancy is often cheaper and more effective than making failures rarer.
- Availabilities multiply in series; in parallel, unavailabilities multiply, but only if the copies fail independently and failover works.
- RTO (how long to restore) and RPO (how much data you may lose) come from the business and drive disaster recovery design and cost.

- SLIs measure, SLOs set internal targets and SLAs are contracts; an SLO turns availability into an error budget that teams can spend on change.

Four words that are often confused

Users experience a system as either working or not. Architects need finer words, because each word points at a different design response. Availability is the proportion of time (or of requests) for which the system is ready to deliver its service when asked. Reliability is the probability that the system performs correctly, without failure, for a stated period under stated conditions. Fault tolerance is the ability to keep delivering the service even though some part has failed. Resilience is the broader capacity to absorb disruption, keep the essential service going (perhaps degraded) and recover to normal afterwards.

ISO/IEC 25010:2023 places availability, fault tolerance and recoverability, together with faultlessness (which replaced the 2011 term maturity), as sub-characteristics of reliability. Bass, Clements and Kazman, in *Software Architecture in Practice*, treat availability as its own quality attribute because it drives so many architectural decisions. Neither view is wrong; use whichever helps the conversation, but be explicit about which word you mean.

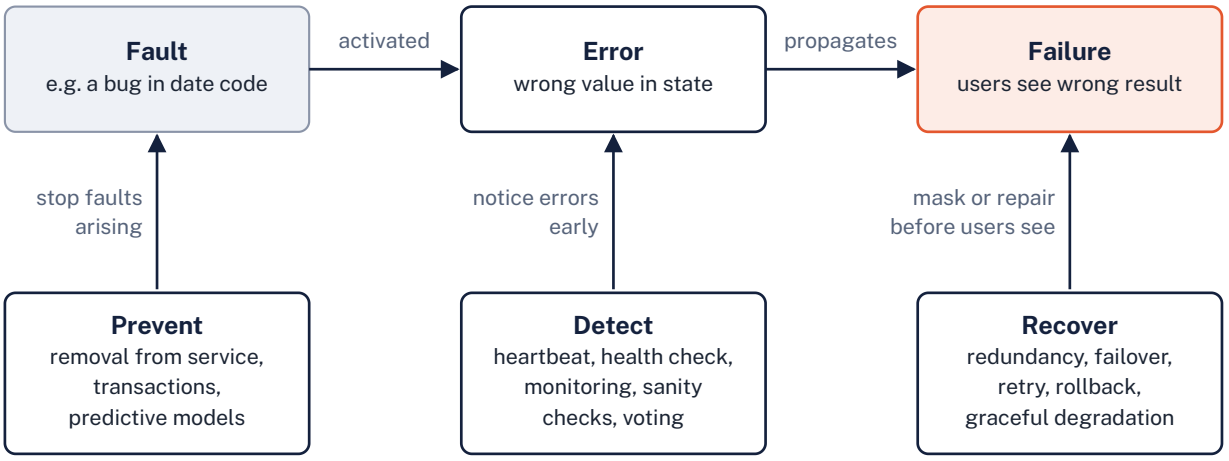
The difference between availability and reliability matters in practice. Imagine a service that crashes once an hour and restarts in one second. Its availability is 3,599 / 3,600 = 99.97%, which looks excellent. Its reliability over a two-hour batch job is poor: if failures arrive at random at a rate of one per hour, the chance of running two hours without one is e^{-2}, about 13.5%. A payment terminal cares about availability; a long-running scientific calculation, a surgical robot or an aircraft cares about reliability over the mission.

Fault, error and failure

The dependability community (Avižienis, Laprie, Randell and Landwehr, 2004) separates three ideas that everyday speech blurs. A fault is the cause: a bug, a failed disk, a misconfiguration, an expired certificate. A fault can lie dormant for years. When it is activated it produces an error: a part of the system's internal state that is wrong, such as a corrupted record or an incorrect value in memory. If the error reaches the service boundary, the result is a failure: the service delivered to users deviates from what it should be.

Bass, Clements and Kazman use the same idea more simply: a failure is a deviation from the specification that the system's users can observe, and the cause of a failure is a fault. The point for architects is that every link in the chain is a place to intervene. You can stop faults arising (code review, removing a component from service before it wears out), detect errors before they spread (validation, health checks), and contain or mask errors so that they never become failures (redundancy, retries, transactions).

The chain and where tactics act



Tactic families from Bass, Clements and Kazman, Software Architecture in Practice (4th edition, 2021)

Prevent tactics stop faults arising, detect tactics notice errors, and recover tactics stop errors becoming visible failures.

Counting the nines

Availability targets are usually written as a percentage with a number of nines: 99.9% is three nines. It helps to convert every target into allowed downtime, because a percentage hides how little time you really have. The calculation is simple: downtime = (1 - availability) x period. For a 365-day year that is 525,600 minutes, so 99.9% allows 0.001 x 525,600 = 525.6 minutes, about 8.76 hours. For a 30-day month (43,200 minutes) it allows 43.2 minutes.

Each extra nine divides the allowance by ten, while the engineering cost usually rises much faster. At 99.99% you have about 4.3 minutes a month: less than the time it takes for a person to be paged, wake up, log in and understand what is wrong. Above three or four nines, recovery must therefore be automatic; humans are too slow to be in the loop.

Allowed downtime for common availability targets (365-day year, 30-day month)

Availability	Per year	Per month	Per week	Per day
99% (two nines)	3.65 days (87.6 h)	7.2 hours	1.68 hours (100.8 min)	14.4 minutes
99.9% (three nines)	8.76 hours	43.2 minutes	10.1 minutes	1.44 minutes
99.95%	4.38 hours	21.6 minutes	5.04 minutes	43.2 seconds
99.99% (four nines)	52.6 minutes	4.32 minutes	1.01 minutes	8.64 seconds
99.999% (five nines)	5.26 minutes	25.9 seconds	6.05 seconds	0.86 seconds

WATCH OUT

Time-based or request-based?

The table assumes availability is measured as uptime over clock time. Many online services instead measure it as the proportion of successful requests: 999,000 good responses out of 1,000,000 is 99.9%. The request-based view handles partial outages naturally (one region down, or only checkout failing) and weights busy hours more than quiet ones. Whichever you choose, write down exactly what counts as success (status codes, a latency limit?), where it is measured (at the load balancer or in the user's browser) and over what window. Two teams can both claim 99.9% while meaning very different things.

MTBF, MTTF and MTTR

Three averages connect failures and repairs. Mean time to failure (MTTF) is the average time a component runs before it fails. Mean time to repair (MTTR) is the average time from failure to restored service (some organisations read the R as recover or restore; the idea is the same, but say which clock you use: from failure, from detection, or from the first alert). Mean time between failures (MTBF) is the average time from one failure to the next.

The classic steady-state formula is **A = MTBF / (MTBF + MTTR)**. Strictly, for a repairable system MTBF = MTTF + MTTR, and the exact expression is MTTF / (MTTF + MTTR): uptime divided by uptime plus downtime. Textbooks often write MTBF where they mean mean uptime; when repairs are short compared with the time between failures, as they usually are, the difference is negligible. The formula assumes a long-run average and says nothing about how outages are distributed, which is why it pairs well with a reliability measure and with percentiles of repair time.

WORKED EXAMPLE

Worked example: is it cheaper to fail less often or to recover faster?

An order service for a food-delivery app fails on average once every 30 days (MTBF = 720 hours). Each failure takes 4 hours to repair, because an engineer must be paged, diagnose the problem and restore a database by hand. The business wants better than 99.9%.

Current availability	720 / (720 + 4) = 720 / 724 = 0.99448, so 99.45%
Current downtime per year	(1 - 0.99448) x 8,760 h = about 48.4 hours
Option 1: double MTBF	Better hardware and testing; failures every 60 days. A = 1,440 / 1,444 = 0.99723 (99.72%), about 24.3 hours a year
Option 2: cut MTTR to 15 minutes	Automated detection, a hot standby and scripted failover. A = 720 / 720.25 = 0.99965 (99.97%), about 3.0 hours a year
Target check	Only option 2 beats 99.9% (8.76 hours a year)

Halving how often failures happen halved the downtime. Cutting repair time by a factor of 16 cut the downtime by about the same factor, and passed the target.

Raising MTBF fights physics and complexity: you cannot make disks, networks or people perfect. Lowering MTTR is mostly within the architect's control: detection, automation, redundancy, small reversible deployments. This is why modern operations practice concentrates on recovery.

MTTR is an average and hides long tails. One eight-hour outage among many five-minute ones dominates the year, so track the distribution of incident durations, not only the mean.

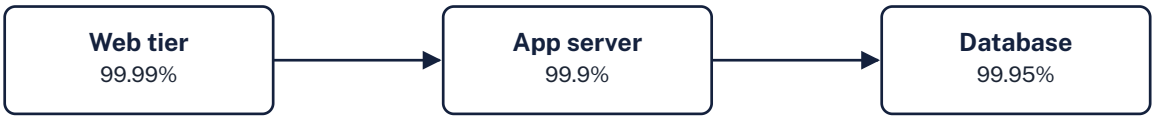
Components in series and in parallel

Real systems are chains of parts. If a request needs the web tier, the application server and the database, all three must work: the components are *in series*, and their availabilities multiply. Because every factor is below one, the product is always lower than the weakest link. Ten components at 99.9% each give 0.999^{10} = 0.990, only 99.0%.

If instead any one of several identical copies is enough, the copies are *in parallel* (redundancy). The pair fails only when every copy fails at once, so you multiply the unavailabilities: $A_{parallel}$ = $1 - (1 - A)^n$ for n identical copies. Two servers at 99% each give 1 - 0.01 x 0.01 = 99.99%.

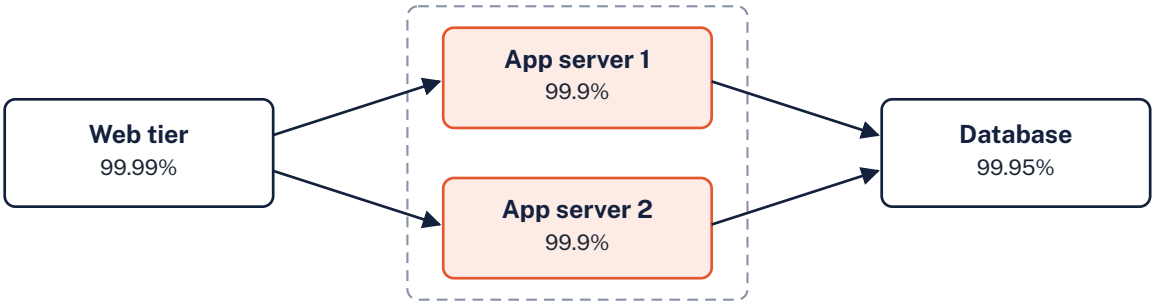
Series and parallel with numbers

A. Three components in series: every one must work



0.9999 x 0.999 x 0.9995 = 0.9984 (99.84%, about 14 hours down a year)

B. Same chain with two app servers in parallel: either one is enough



pair: 1 - (0.001 x 0.001) = 0.999999; chain: 0.9994 (99.94%, about 5.3 hours a year)

Duplicating the weakest stage lifts the whole chain from about 14 hours of downtime a year to about 5.3.

WORKED EXAMPLE

Worked example: where should the next pound be spent?

Continuing the diagram. The chain in part A is 99.84%. Which change brings it closest to 99.99%?

Add a second app server	0.9999 x 0.999999 x 0.9995 = 0.99940 (99.94%): about 5.3 hours a year
Then add a database replica with failover	Database pair: $1 - 0.0005^2 = 0.99999975$. Chain: 0.9999 x 0.999999 x 0.99999975 = 0.99990 (99.99%): about 53 minutes a year
What now limits the system	The single web tier (99.99%) contributes almost all of the remaining downtime

The arithmetic assumes the copies fail independently. Two servers in the same rack, on the same power feed, running the same buggy release, or both depending on one DNS provider, share common-mode faults and can fail together. Real gains are always smaller than the formula suggests.

It also assumes failover is instant and always works. A standby that has never been tested in a real failover is a hope, not a component.

A useful rule from Google's SRE authors (Treynor, Dahlin, Rau and Beyer, 2017): each critical dependency should offer roughly one more nine than the service that relies on it, or the service needs a way to cope without it.

Availability tactics: detect, recover, prevent

Bass, Clements and Kazman group availability tactics into three families. They are design options, not a checklist to apply in full; each has a cost.

The three families of availability tactics

1. **Detect faults**

   Heartbeat: a component sends a periodic "I am alive" message, and silence triggers action. Ping/echo: a monitor asks and expects a reply. Health checks: an endpoint such as `/health` that a load balancer or orchestrator polls (a liveness check asks "is the process running?"; a readiness check asks "can it serve traffic now?"). Monitoring of error rates and latency, sanity checks on values, and voting between replicas that compute the same result.

2

Recover: preparation and repair

Redundant spares in three temperatures: *active-active* (all copies serve live traffic), *hot or warm standby* (a copy kept in step and ready), and *cold spare* (started only when needed). Failover switches traffic to a healthy copy. Retry handles transient faults (with limits and back-off, or retries become a self-inflicted overload). Rollback returns to a known good state or release. Graceful degradation keeps the core service going without a failed feature: the shop still takes orders even if recommendations are down.

3

Recover: reintroduction

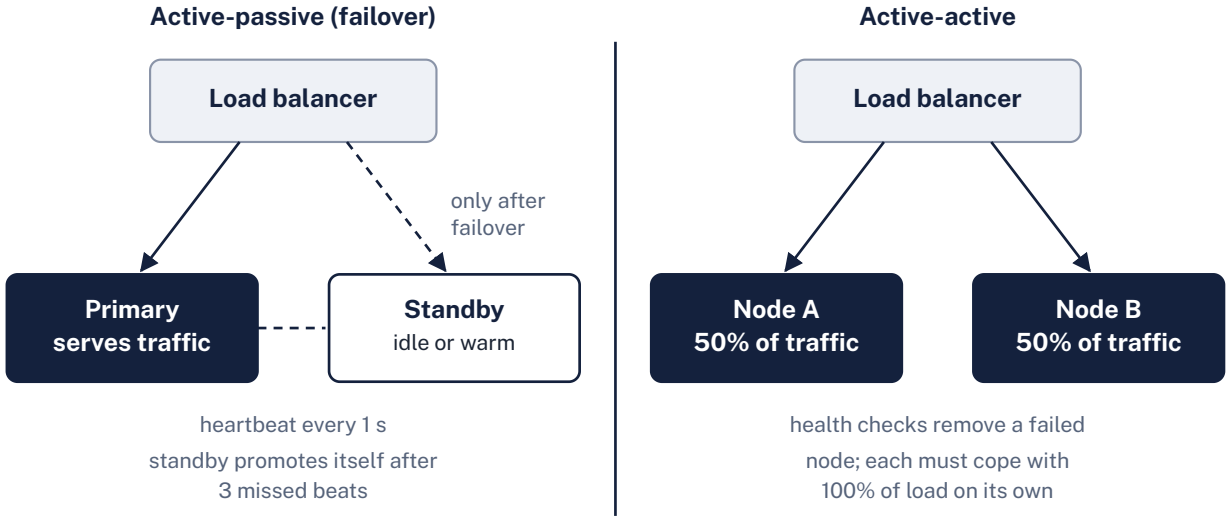
Bringing a repaired component back safely: run it in shadow mode, resynchronise its state, restart in escalating steps (a thread, then a process, then a machine).

4

Prevent faults

Removal from service (take a node out of rotation for maintenance or a scheduled restart before a known memory leak bites), transactions (all-or-nothing updates so a crash cannot leave half-written state), predictive models (replace a disk when its error counters climb), and exception prevention (stronger types, input validation).

Two shapes of redundancy



Active-passive pays for idle capacity and a failover gap; active-active uses all capacity but each node must survive the loss of the other.

Choosing between active-passive and active-active

Active-passive is simpler to reason about, especially for stateful components such as a relational database with one writable primary. Its costs are the idle standby, a failover gap (detection plus promotion, often tens of seconds), and the risk that the standby has quietly drifted out of step. A classic trap is *split brain*: the network between the two breaks, each believes the other is dead, and both accept writes. Guarding against it needs a quorum or an external arbiter.

Active-active uses all capacity and has no promotion step, but every node must be sized to carry the full load if its partner fails. Two nodes each running at 70% utilisation will be asked for 140% when one dies, and the survivor collapses: a redundant design that fails anyway. Active-active also forces you to solve data consistency across copies, which is a large topic in itself (see consistency, CAP and PACELC).

Disaster recovery: RTO and RPO

Redundancy within one data centre does not protect you from losing the whole data centre, a region, a cloud account or your data to ransomware. Disaster recovery planning uses two numbers that should come from the business, not from engineers. The recovery time objective (RTO) is the longest acceptable time to restore the service after a disaster. The recovery point objective (RPO) is the largest acceptable loss of data, measured in time: an RPO of 15 minutes means you may lose at most the last 15 minutes of changes.

The two numbers drive very different designs and costs. Nightly backups copied to another region give an RPO of up to 24 hours and an RTO of many hours (restoring terabytes takes time). Continuous asynchronous replication to a warm standby in another region might give an RPO of seconds and an RTO of minutes, at perhaps double the infrastructure cost. Synchronous replication can bring the RPO close to zero but adds the round-trip time between sites to every write. Cloud providers describe a ladder of strategies: backup and restore, pilot light, warm standby and multi-site active-active, each faster and more expensive than the one before. The details belong to multi-region and disaster recovery.

Whatever the strategy, a backup you have never restored is not a backup. Test restores on a schedule and time them; the measured restore time is your real RTO.

From targets to SLIs, SLOs and SLAs

Modern operations practice expresses availability with three related terms, covered properly in SLIs, SLOs and error budgets. A service level indicator (SLI) is the measurement, for example the proportion of checkout requests that succeed in under 800 ms. A service level objective (SLO) is the internal target for that measurement over a window, for example 99.9% over 30

days. A service level agreement (SLA) is a contract with customers that names consequences, such as service credits, if a level is missed; it is normally set looser than the SLO so the team has warning before the contract is breached.

The SLO turns availability into a budget. A 99.9% monthly SLO allows 0.1% of requests to fail; at 2 million requests a month that is 2,000 failed requests, or about 43 minutes of full outage. Teams spend this error budget on risk: releases, experiments, migrations. When it is gone they slow down and invest in reliability. This links the abstract nines in this section to day-to-day decisions.

> **IF YOU ARE NEW TO THIS**
>
> **If you are new to this**
>
> You do not need to memorise the nines table. Remember the method: one minus the percentage, times the period. 99.9% leaves 0.1% of a 30-day month, which is 0.001 x 43,200 minutes = 43.2 minutes. Every other row is the same sum.

Common mistakes

Promising nines without counting dependencies: a 99.99% service built on a single 99.9% database cannot meet its target. Adding redundancy without removing shared causes (the same release, the same configuration push, the same certificate expiring on every node at midnight). Measuring availability from inside the system, where everything looks green, while users cannot reach it because of DNS or a content delivery network. Planning failover but never rehearsing it; chaos engineering exists precisely to exercise these paths deliberately (chaos engineering). Finally, chasing a number for its own sake: a staff rota system used in office hours does not need five nines, and the money is better spent elsewhere.

To make availability testable, write it as a scenario (Section 2.3): “When the primary database server fails during peak hours, the system detects it within 10 seconds and resumes accepting orders within 60 seconds, losing no confirmed order.” Every part of that sentence becomes a design choice and a test.

CHECK YOUR UNDERSTANDING

▼ **How many minutes of downtime does a 99.95% target allow in a 30-day month?**

0.0005 x 43,200 = 21.6 minutes.

▼ **Two independent load balancers each have 99.5% availability and either can carry all traffic. What is the availability of the pair?**

1 - (0.005 x 0.005) = 1 - 0.000025 = 99.9975%.

▼ **Why might adding a second identical server give much less benefit than the parallel formula predicts?**

Because the copies may share common causes of failure, such as the same release, configuration, power, network or dependency, so they do not fail independently; failover itself may also be slow or broken.

▼ **A team's backups run every 6 hours and a full restore takes 3 hours. What are their effective worst-case RPO and RTO?**

RPO up to 6 hours of lost data, and an RTO of at least 3 hours plus the time to detect the disaster and decide to restore.

▼ **Try it: A chain runs gateway 99.95%, order service 99.9%, payment 99.9%. What is its availability, and downtime a year? Then add a second, independent order instance with working failover. Recalculate, and name the assumption that could spoil the result.**

In series: 0.9995 x 0.999 x 0.999 = 0.99750, about 99.75%, or 21.9 hours a year. Two order instances give 1 - 0.001 x 0.001 = 0.999999, so the chain is 0.9995 x 0.999999 x 0.999 = 0.99850, about 13.1 hours. Payment and the gateway now dominate. The assumption is independence: a shared bad release, power feed or database means both copies fail together, and failover must really work.

SOURCES

- Google SRE book: Embracing Risk (time-based and request-based availability): https://sre.google/sre-book/embracing-risk/
- Google SRE book: Service Level Objectives: https://sre.google/sre-book/service-level-objectives/
- SEI blog: Tactics and Patterns for Software Robustness (availability tactics): https://www.sei.cmu.edu/blog/tactics-and-patterns-for-software-robustness/
- AWS whitepaper: Disaster recovery options in the cloud: https://docs.aws.amazon.com/whitepapers/latest/disaster-recovery-workloads-on-aws/disaster-recovery-options-in-the-cloud.html
- Avižienis, Laprie, Randell and Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing (IEEE TDSC, 2004): https://doi.org/10.1109/TDSC.2004.2

SECTION 2.6 OF 2.8 · INTERMEDIATE · 22 MIN READ · LAST REVIEWED NOVEMBER 2026

Modifiability, testability and deployability: designing for change

BEFORE YOU READ THIS

This section builds on:

- 0.3 Organising code
- 0.8 How software gets built
- 2.3 Quality attribute scenarios

Terms it uses from earlier: Feature flag, Interface, Module, Unit test, Production

THE IDEA IN BRIEF

Think of a kitchen. If the cooker is built into the wall with the pipes buried in concrete, replacing it means demolishing the wall. If it plugs into a standard socket, you swap it in an afternoon. Software is the same: some parts will certainly change, so architects give them standard connections, make it easy to check that everything still works after a swap, and make delivering the new version routine rather than a risky event.

WHAT YOU NEED TO KNOW

- Modifiability is the cost and risk of change; specify it by saying what is likely to change, who makes the change, when it binds and what the change may cost.
- Modifiability tactics increase cohesion, reduce coupling (encapsulate, use an intermediary, restrict dependencies) and defer binding, but each later binding time adds machinery.
- Testability depends on controllability and observability; isolate dependencies behind interfaces and inject them so tests can use test doubles.
- Deployability rests on small batches, automation, independently deployable units, backward-compatible changes, gradual rollouts and fast rollback.
- DORA's current software delivery metrics are change lead time, deployment frequency, failed deployment recovery time, change fail rate and deployment rework rate.
- ISO/IEC 25010:2023 maintainability has five sub-characteristics: modularity, reusability, analysability, modifiability and testability.

Three qualities about change

Most of the cost of a successful system is incurred after its first release, as it is changed again and again. Three related qualities describe how well an architecture supports that. Modifiability is about the cost and risk of making a change. Testability is about how easily you can show, by testing, that the system (or a part of it) behaves as intended, and how readily faults reveal themselves. Deployability is about how quickly, safely and predictably a change can be put into production, and taken back out again.

They are linked in a loop. A change is only cheap if you can make it in one place (modifiability), know quickly that you have not broken anything (testability), and ship it without ceremony (deployability). A system that is modular but has no fast tests will still change slowly, because every change needs weeks of manual regression testing. A system with good tests but a quarterly release window will still deliver value slowly.

Modifiability: four questions about any change

Bass, Clements and Kazman suggest that modifiability is meaningless until you say what is expected to change. Planning for every possible change produces an over-engineered system full of unused extension points; planning for none produces a system where every change is a rewrite. Four questions sharpen the requirement.

What can change? Functions, platforms (a new database version, a move to another cloud), the environment (a new partner API, new regulation), qualities (twice the load), or capacity. **How likely is the change?** Spend design effort where change is probable: payment providers, tax rules and user interfaces change often; the concept of an order rarely does. **Who makes the change, and when?** A developer at coding time, a release engineer at deployment, an administrator through configuration, or an end user through settings. The moment at which a decision is fixed is called its binding time. **What does the change cost?** The effort, elapsed time and money to make, test and deploy it, plus the number of other elements affected and the risk of introducing defects.

Answering those questions gives a scenario of the kind introduced in Section 2.3: “A developer adds support for a new payment provider, changing no existing modules other than configuration, within 3 person-days including tests, with no increase in checkout errors.”

Modifiability scenarios with different sources and binding times

Who changes	What changes	When it binds	Response measure
Developer	Add a new delivery partner to a food-delivery platform	Coding time	Only the partner adapter changes; done, tested and deployed within 5 days
Product manager	Turn a new checkout flow on for 10% of users	Run time (feature flag)	Takes effect within 1 minute, without a deployment
Operations engineer	Point the system at a new database host	Deployment (configuration)	No code change; one pipeline run of under 15 minutes
Customer administrator	Add a custom field to an invoice	Run time (settings)	No supplier involvement; available immediately
Platform team	Upgrade the web framework to a new major version	Build time	Fewer than 20 files change outside an anti-corruption layer; under 2 weeks

Tactics for modifiability

The tactics in *Software Architecture in Practice* reduce the cost of change by limiting how far a change ripples. They rest on two old ideas, treated in depth in Chapter 3. Cohesion measures how strongly the responsibilities inside one module belong together; coupling measures how much one module depends on another. A change that is likely should touch one cohesive module, and that module should be loosely coupled to the rest.

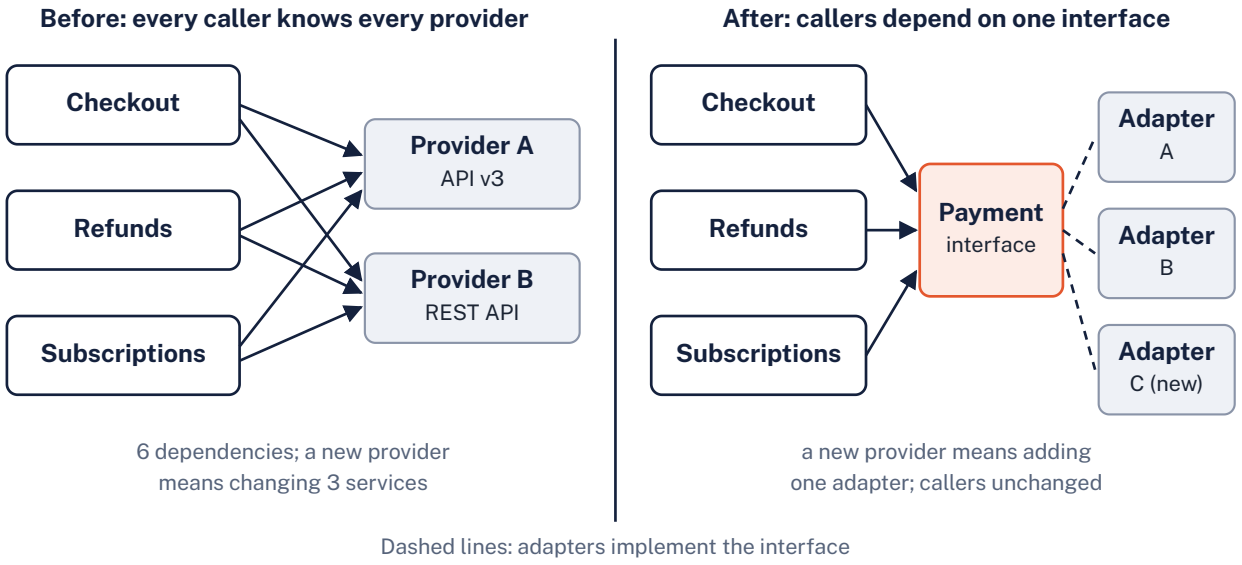
Increase cohesion. Split a module that does two unrelated jobs (a `CustomerService` that handles both login and loyalty points), and move responsibilities to where they belong, so that a likely change lands in one place.

Reduce coupling. *Encapsulate*: hide a module's internals behind an explicit interface, following David Parnas's principle of information hiding (1972), so callers cannot depend on what may change. *Use an intermediary*: put something between two parties so neither knows the other directly: an adapter, a message broker, a publish-subscribe channel, a repository in front of a database, or an API gateway. *Restrict dependencies*: decide which modules may call which (layering rules, a module visible only to its own package) and enforce it with tooling. *Abstract common services*: provide shared behaviour once, behind an interface, rather than copying it.

Defer binding. Fix a decision as late as is useful, so that the change can be made later and by someone further from the code: configuration files, environment variables, plug-ins discovered at start-up, service discovery and feature flags. Each later binding time adds machinery

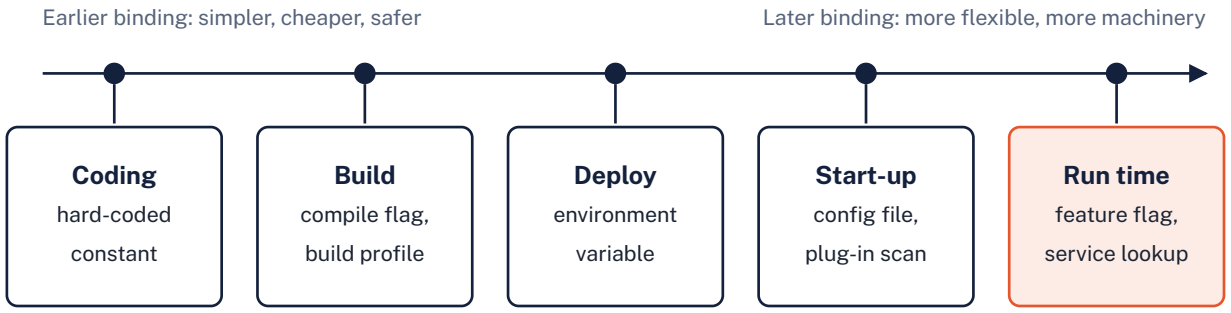
(configuration to validate, flags to remove, versioned plug-in interfaces), so defer only decisions that genuinely vary.

Using an intermediary to contain change



An interface with one adapter per provider turns a change that touched three services into the addition of one adapter.

Binding time



Moving a decision to the right lets it change later and without a release, but every step adds machinery to manage.

DECISION GUIDE

Decision guide: how late should this decision bind?

Use it when a stakeholder says "this might change". Start at the top and stop at the first yes.

Q1 **Is the change unlikely, or would it need new code and new tests anyway?**

- **Yes** Bind at coding time. Keep it simple and well encapsulated so the change stays local when it comes.
- **No** Go to the next question.

Q2 **Does the value differ only between environments (test, staging, production) or between customers' installations?**

- **Yes** Bind at deployment: environment variables or configuration injected by the pipeline, kept in version control.
- **No** Go to the next question.

Q3 **Do third parties or other teams need to add behaviour without changing your code?**

- **Yes** Bind at start-up or run time with a plug-in or extension interface, and version that interface carefully.
- **No** Go to the next question.

Q4 **Must non-developers switch behaviour on or off quickly, or must you release to a fraction of users first?**

- **Yes** Bind at run time with a feature flag, and give every flag an owner and a removal date.
- **No** Bind at build or deployment time; later binding would add machinery nobody needs.

WORKED EXAMPLE

Worked example: what is the modifiability scenario worth?

A food-delivery company integrates restaurants' point-of-sale systems. Today each new point-of-sale vendor needs changes in the menu, order and refund services: about 15 person-days of work plus a 5-day regression test cycle. Sales expects 8 new vendors over the next 12 months. The architect proposes a vendor adapter interface (as in the diagram above). Assume a fully loaded cost of £600 per person-day.

Cost per vendor today	15 + 5 = 20 person-days, so 20 x £600 = £12,000
Cost for 8 vendors today	8 x £12,000 = £96,000
One-off cost of the refactoring	Design and build the interface and migrate the 3 existing vendors: 30 person-days = £18,000
Cost per vendor afterwards	Write one adapter (3 days) plus contract tests against the interface (1 day): 4 person-days = £2,400
Cost for 8 vendors afterwards	£18,000 + 8 x £2,400 = £37,200
Saving in year one	£96,000 - £37,200 = £58,800
Break-even	Each vendor saves £12,000 - £2,400 = £9,600, so the £18,000 is repaid after £18,000 / £9,600 = 1.9, that is, by the second new vendor

The calculation is only as good as the prediction of change. If sales signs no new vendors, the refactoring is pure cost. This is why modifiability requirements must say what is likely to change.

Lead time often matters more than cost: cutting a vendor launch from about 4 weeks to under 1 week may win contracts, which no person-day sum captures.

The new adapter interface is itself a commitment. Changing it later affects every adapter, so it deserves careful design and versioning.

IF YOU ARE NEW TO THIS

Step by step: deciding when a choice is fixed

This section's hardest idea is binding time: the moment at which a decision is fixed, which decides who can change it later and how easily. Follow one decision through three binding times, using the section's diagram, in which checkout reaches payment providers through one Payment interface with an adapter for each provider.

- Adapter A and Adapter B each connect the Payment interface to one provider. Suppose the decision is which adapter checkout uses.
- Coding time: a developer writes 'use Adapter A' into the code. It is the simplest choice, but switching to B means a code change, tests and a new release, and only a developer can make it.
- Deployment time: the choice moves into configuration, such as an environment variable that the deployment pipeline supplies. Test and production can now use different providers, and an operations engineer can switch provider with one pipeline run and no code change.
- Run time: a feature flag sends 10% of payments through Adapter B, and a product manager can turn that share up or down within a minute, without a deployment.
- Each step later adds machinery. The configuration must be checked so that a mistyped value cannot stop payments; the flag needs an owner and a removal date; and while either adapter may be live, both must be kept tested.
- Adding a new Adapter C still means new code and new tests, whatever the binding time. Deferring binding changes when, and by whom, a choice between existing options is made.
- **The lesson:** ask who needs to change a decision and how often, then pick the earliest binding time that lets them. Binding any later only adds machinery that nobody uses.

Testability: controllability and observability

A component is testable when a test can put it into a known state, feed it chosen inputs and see what it did. Two properties capture that. **Controllability** is the ability to set a component's inputs and internal state from a test. **Observability** (in the testing sense) is the ability to see its outputs and relevant internal state. Code that reads the system clock directly, calls a real payment service, picks random numbers or writes to a shared database is hard to control; code that hides its results in private state or only in log text is hard to observe.

Architectural tactics for testability therefore aim to give tests control and visibility, and to limit complexity. **Isolate dependencies** behind interfaces, then use dependency injection: the component receives its collaborators (a clock, a payment gateway, a repository) from outside rather than creating them, so a test can pass in a substitute. Those substitutes are test doubles, a term popularised by Gerard Meszaros: a *stub* returns canned answers, a *fake* is a working lightweight implementation (an in-memory repository), a *mock* checks that expected calls were made, and a *spy* records calls for later assertions. **Provide test interfaces**: hooks to set state, record and replay traffic, or run in a sandbox environment. **Limit non-determinism**: avoid hidden concurrency and global state, and make time and randomness injectable. **Limit structural complexity**: fewer dependencies and no cycles mean smaller test set-ups.

Architecture shapes the testing strategy at higher levels too. In a system of services, each boundary needs contract tests that check both sides agree on the API, and a few end-to-end tests that exercise the real path. A design with clean boundaries lets most tests run fast and in isolation, while a tangled design forces slow, brittle end-to-end tests for everything.

Dependency injection makes a component controllable



The same OrderService runs against the real gateway in production and against a scripted fake in tests, because it depends only on the interface.

```
class OrderService:
    def __init__(self, gateway, clock):
        self.gateway = gateway   # anything with charge(amount, card)
        self.clock = clock       # anything with now()

    def place(self, order):
        if self.clock.now().hour ⩾ 23:
            return "kitchen closed"
        return self.gateway.charge(order.total, order.card)

# In a test: full control, no network, no waiting until 11 pm
class FakeClock:
    def __init__(self, t): self.t = t
    def now(self): return self.t

class FakeGateway:
    def __init__(self): self.calls = []
    def charge(self, amount, card):
        self.calls.append(amount)
        return "approved"
```

Constructor injection: the service is given its clock and gateway, so a test can supply fakes.

Deployability

Deployability measures how predictably and cheaply a change moves from a developer's commit into production, and how quickly it can be undone. It became a first-class quality only in the last decade, as continuous delivery showed that release frequency and stability are not opposites. The fourth edition of *Software Architecture in Practice* (2021) added a chapter on it.

The central idea is **small batches**. A release that contains one change is easy to test, easy to reason about and easy to roll back; a release of three months' work contains hundreds of interacting changes, so failures are likely and hard to diagnose. Small batches require **automation**: a deployment pipeline that builds, tests and deploys every change the same way, scripted infrastructure and no manual steps that someone can forget. Architectural tactics include independently deployable units (so one team's release does not wait for another's), backward-compatible interfaces and database migrations, rollback and roll-forward mechanisms, gradual rollouts (canary releases to a small share of traffic first, blue-green switching between two environments) and feature flags, which separate *deploying* code from *releasing* a feature to users. Pipelines and strategies are covered in CI/CD and deployment strategies.

The cost of ignoring deployability can be severe. In August 2012 the trading firm Knight Capital deployed new code manually to its eight order-routing servers but missed one. A reused configuration flag then activated obsolete code on that server, which flooded the market with

orders, producing about 4 million executions in roughly 45 minutes, and cost the firm about 460 million US dollars. An automated, verified deployment and the removal of dead code would each have prevented it.

Measuring delivery: the DORA metrics

The DORA research programme (DevOps Research and Assessment, described in *Accelerate* by Nicole Forsgren, Jez Humble and Gene Kim, 2018, and continued in annual reports now run within Google Cloud) found that a small set of measures predicts software delivery performance, and that the best performers are fast and stable at the same time. The current model has five software delivery performance metrics, grouped into throughput and instability. Their names have evolved: the old "mean time to restore" became failed deployment recovery time in 2023, and deployment rework rate was added in 2024.

Two cautions. First, these metrics describe a team's delivery system, not individuals; using them to rank people invites gaming. Second, they are outcomes. Architecture influences them strongly: teams that can deploy their part independently, test it in isolation and roll it back quickly score better, which is one reason the DORA research links loosely coupled architecture with higher delivery performance.

DORA software delivery performance metrics (dora.dev, current model)

Metric	Group	What it measures	Architectural levers
Change lead time	Throughput	Time from a change being committed to version control to it running in production	Fast automated tests, small independently deployable units, no hand-offs
Deployment frequency	Throughput	How often changes are deployed to production (or the time between deployments)	Independent deployability, automated pipeline, feature flags
Failed deployment recovery time	Throughput	Time to recover from a deployment that fails and needs immediate intervention	Automated rollback, canary releases, good monitoring, small changes
Change fail rate	Instability	Proportion of deployments that need immediate intervention, such as a rollback or hotfix	Testability, contract tests, backward-compatible changes
Deployment rework rate	Instability	Proportion of deployments that are unplanned and happen because of an incident in production	Quality of tests and reviews; observability that catches problems early

Maintainability in ISO/IEC 25010

ISO/IEC 25010:2023 groups these concerns under the characteristic maintainability, with five sub-characteristics: **modularity** (the system is made of discrete components so that a change to one has minimal impact on others), **reusability** (an asset can be used in more than one system or in building other assets), **analysability** (how easily you can assess the impact of a change, diagnose a deficiency or find the parts to modify), **modifiability** (the system can be changed without introducing defects or degrading quality) and **testability** (test criteria can be established and tests performed to determine whether they are met). Deployability has no sub-characteristic of its own in the standard; installability and replaceability sit under flexibility. The model is a checklist, as Section 2.2 explains; scenarios are what make each item concrete.

WATCH OUT

Common mistakes

- Building for imagined change: plug-in frameworks and abstraction layers for variations nobody has asked for. Every extension point has a cost in complexity and testing. Ask how likely the change is before deferring binding.
- Feature flags without an exit: hundreds of stale flags multiply the paths through the code that nobody tests. Give each flag an owner and a removal date.
- Microservices as a cure for slow delivery: splitting a system into services that must still be released together (a distributed monolith) makes deployability worse, not better.
- Mock-heavy tests that verify implementation details: they pass while the system is broken and fail whenever the code is refactored. Prefer fakes and contract tests at architectural boundaries.
- Measuring lead time from the start of a ticket in one team and from commit in another, then comparing them. Define each metric once and apply it consistently.

CHECK YOUR UNDERSTANDING

▼ **Why is "the system shall be easy to change" not a useful requirement?**

It does not say what will change, who will change it, when, or what cost is acceptable, so nobody can design for it or test whether it has been met.

▼ **What is the difference between a stub and a mock?**

A stub returns canned answers so the test can control inputs; a mock also checks that the expected calls were made, so the test verifies interactions.

▼ **Why do feature flags help deployability?**

They separate deploying code from releasing a feature, so incomplete or risky features can be deployed switched off, released gradually and switched off again without a new deployment.

▼ **A team deploys monthly and each release contains about 200 changes. Which DORA metrics are likely to suffer, and why?**

Change lead time (changes wait weeks for the release), deployment frequency, and probably change fail rate and recovery time, because large batches are more likely to fail and harder to diagnose and roll back.

▼ **Try it: Each new courier partner costs 10 person-days today. A 15-day refactoring would cut that to 4 days. Five partners are expected this year, at £600 a day. Is the refactoring worth it, and when does it break even?**

Without it: 5 x 10 = 50 days, £30,000. With it: 15 + (5 x 4) = 35 days, £21,000, a saving of 15 days or £9,000. Each partner saves 6 days, so it breaks even at 15 / 6 = 2.5, meaning by the third partner. It is worth doing if at least three partners seem likely. It also makes each integration less risky, but check the forecast first, since the saving depends on it.

SOURCES

- DORA: software delivery performance metrics: https://dora.dev/guides/dora-metrics/
- DORA: history of the software delivery metrics: https://dora.dev/guides/dora-metrics/history/
- Martin Fowler: Test Double: https://martinfowler.com/bliki/TestDouble.html
- Pete Hodgson: Feature Toggles (aka Feature Flags), martinfowler.com: https://martinfowler.com/articles/feature-toggles.html
- Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern: https://martinfowler.com/articles/injection.html

SECTION 2.7 OF 2.8 · INTERMEDIATE · 22 MIN READ · LAST REVIEWED NOVEMBER 2026

Security, usability, interoperability, cost and sustainability as qualities

BEFORE YOU READ THIS

This section builds on:

- 2.2 Quality models
- 2.3 Quality attribute scenarios
- 2.5 Availability and reliability

Terms it uses from earlier: ISO/IEC 25010, Quality attribute scenario, Tactic, Constraint, API (application programming interface)

THE IDEA IN BRIEF

When you choose a bank, you care that it keeps your money safe, that its app is easy to use, that it works with other banks, that its fees are fair and, perhaps, that it does not waste energy. None of these is a feature you can point to on a screen; each is a quality of the whole service. Architects have to turn each of them into something specific enough to design for and check, and they often have to balance one against another.

WHAT YOU NEED TO KNOW

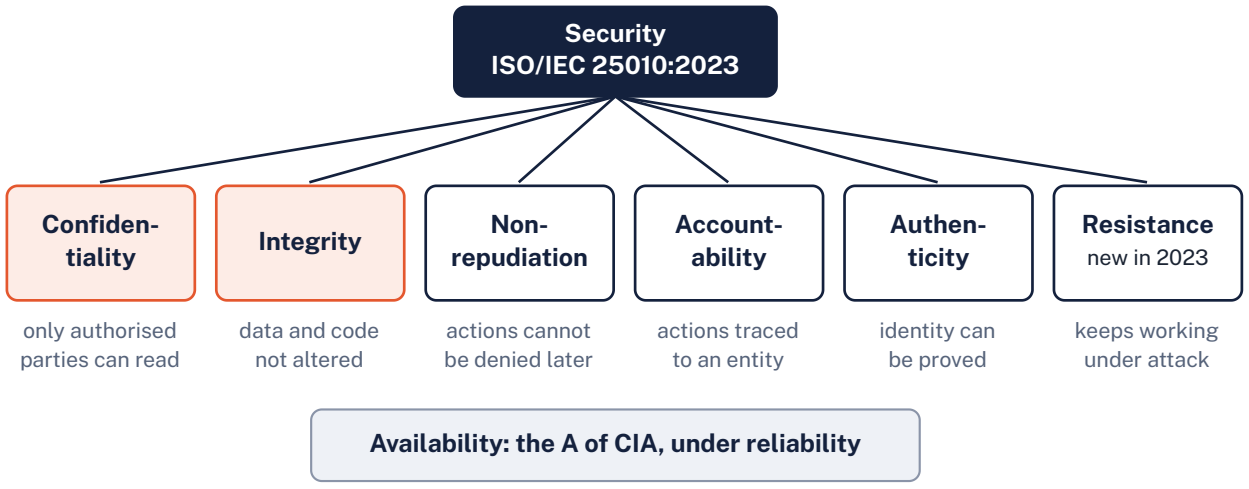
- Security starts from confidentiality, integrity and availability; ISO/IEC 25010:2023 lists confidentiality, integrity, non-repudiation, accountability, authenticity and resistance, with availability under reliability.
- Security scenarios name the attacker, the attack, the target and a measurable response such as detection time or data exposed; tactics resist, detect, react to and recover from attacks.
- Interaction capability (formerly usability) affects architecture through responsiveness, undo, cancel and offline working; accessibility to WCAG 2.2 level AA is increasingly a legal constraint.
- Interoperability needs agreement on meaning as well as format; industry standards and adapters reduce the cost.
- Cost and carbon are best expressed per unit of work; the SCI specification (ISO/IEC 21031:2024) computes ((E x I) + M) per R and counts only real reductions.
- Safety, observability, data residency, auditability and retention are qualities too, and should be captured early as testable constraints.

Security as a quality attribute

Security is the system's ability to protect data and functions from unauthorised access and use while still serving those who are authorised. Architects often start from the CIA triad: **confidentiality** (only authorised parties can read data), **integrity** (data and code are not altered without authorisation, and alterations are detectable) and **availability** (authorised users can use the service when they need it). Attacks map onto the three: stealing customer records breaks confidentiality, changing a payee's bank details breaks integrity, and a denial-of-service attack breaks availability.

ISO/IEC 25010:2023 refines the picture. Its security characteristic has six sub-characteristics: confidentiality, integrity, **non-repudiation** (actions or events can be proved to have taken place, so that a party cannot later deny them), **accountability** (actions can be traced uniquely to the entity that performed them), **authenticity** (the identity of a subject or resource can be proved to be the one claimed) and, new in the 2023 edition, **resistance** (the product keeps operating as intended while under attack by a malicious actor). Availability appears under reliability in that model, which is why it was covered in Section 2.5.

The security sub-characteristics



Confidentiality and integrity come from the CIA triad; the other four make accountability, proof of identity and behaviour under attack explicit.

Writing security scenarios

"The system must be secure" cannot be designed for or tested. A security scenario names a source (who attacks, with what access), a stimulus (what they try), an artifact (what is targeted), an environment (online, during an incident, with the attacker already inside the

network), a response and a measure. Measures for security are often about detection time, the amount of damage, or the time to restore, because "no attack ever succeeds" cannot be verified.

Three examples for an online bank. *Credential stuffing:* an external attacker tries leaked username and password pairs against the login API at 50 attempts per second from many addresses; the system detects the pattern within 2 minutes, challenges or blocks the source addresses, and accounts protected by multi-factor authentication are not taken over. *Insider tampering:* a support employee with legitimate access changes a customer's payee details; the change requires a second person's approval, and the audit log records who, what and when in a form that the employee cannot alter. *Stolen backup:* an attacker obtains a copy of last week's database backup; all personal data in it is encrypted with keys stored separately, so no customer data is readable.

Notice that each scenario implies different architecture: rate limiting and anomaly detection at the edge, a four-eyes approval workflow and tamper-evident logs, and encryption with separate key management. Finding these scenarios systematically is the job of threat modelling, covered in Chapter 8.

IF YOU ARE NEW TO THIS

Step by step: telling the security words apart

The idea that needs the most care here is telling apart the security sub-characteristics that sound alike, especially authenticity, accountability and non-repudiation. Take the insider-tampering scenario above, in which a support employee changes a customer's payee details, and match each safeguard in it to a sub-characteristic.

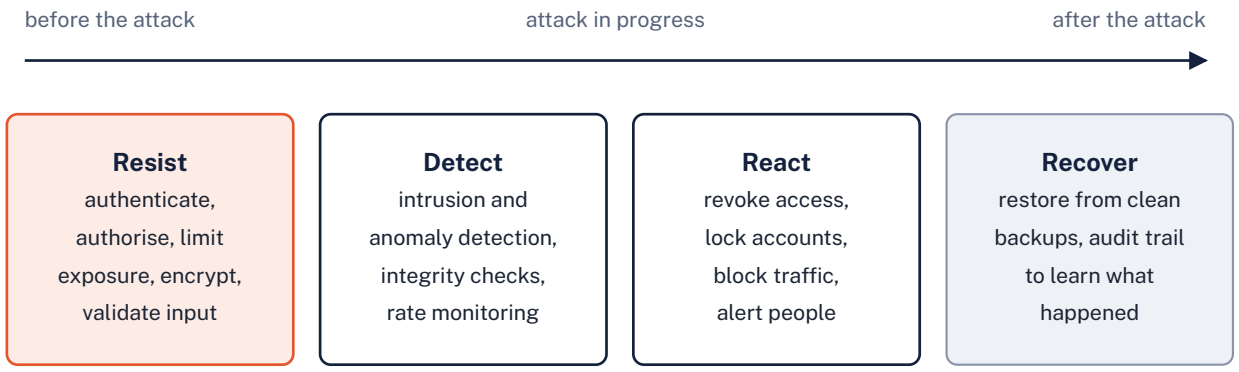
- The payee details must not change without authorisation, and any change must be detectable; in the scenario a second person must approve the change. That protects **integrity**.
- The audit log records who made the change, what it was and when, so it can be traced to that one employee. That is **accountability**: knowing who did it.
- The log can name the right person only if the bank can be sure that whoever logged in as the employee really is that employee, not someone using a stolen password. Being able to prove that a person or system is who or what it claims to be is **authenticity**.
- The log is kept in a form the employee cannot alter, so if they later say 'that was not me', the bank can show that it was. That is **non-repudiation**: proof that the action took place, so it cannot be denied.
- The other two belong to other scenarios. **Confidentiality** (only authorised parties can read data) is what the stolen-backup scenario protects with encryption. **Resistance**, carrying on as intended under attack, is what the credential-stuffing scenario tests.
- **The lesson:** confidentiality and integrity protect the data; authenticity, accountability and non-repudiation concern the people who act on it: proving who they are, tracing what they did and proving it later. Each calls for a different mechanism, so name the one you mean in each scenario.

Security tactics: an overview

Bass, Clements and Kazman group security tactics by when they act. **Resist attacks**: identify, authenticate and authorise actors, limit access and exposure (fewer open ports, fewer public endpoints, least privilege), encrypt data in transit and at rest, separate entities (sensitive data in its own store or network segment), change default settings and validate input. **Detect attacks**: intrusion detection, detecting denial of service, verifying message integrity and spotting unusual delays or patterns. **React to attacks**: revoke access, lock accounts after failed log-ins, inform operators and users. **Recover from attacks**: restore a known good state (the availability tactics again) and keep an audit trail to understand what happened and support non-repudiation.

This section only introduces the families. Principles such as defence in depth and least privilege, identity protocols, encryption, zero trust and supply-chain security are the subject of Chapter 8. The architectural point here is that security is a property of the whole structure, not a component you add at the end: where trust boundaries fall, which component holds which data, and how services authenticate one another are early, expensive-to-change decisions.

Tactics before, during and after an attack



Grouping from Bass, Clements and Kazman; detail in Chapter 8

Resist tactics reduce the chance of success; detect, react and recover tactics limit the damage when resistance fails.

Usability, interaction capability and accessibility

ISO/IEC 25010:2023 renamed usability *interaction capability*: the degree to which a product can be interacted with by specified users to exchange information through the user interface to complete specific tasks. Its sub-characteristics are appropriateness recognisability, learnability, operability, user error protection, user engagement (which replaced user interface aesthetics), inclusivity, user assistance and self-descriptiveness. Much of this is interaction design rather than architecture, but several usability concerns reach deep into the structure.

Responsiveness is the obvious one. Long-standing human-factors guidance, popularised by Jakob Nielsen, gives three limits: about 0.1 seconds feels instantaneous, about 1 second keeps the user's flow of thought, and about 10 seconds is the limit of attention. Meeting them may require caching, asynchronous processing with progress feedback, or optimistic updates. Undo and cancel are architectural: supporting “undo the last five actions” needs commands that can be reversed, and cancelling a long upload needs an operation that can be interrupted cleanly. Working offline in a mobile app requires local storage and synchronisation with conflict handling. These are much cheaper to design in than to retrofit.

Accessibility, the ability of people with disabilities to use the product, is increasingly a legal constraint rather than a preference. The Web Content Accessibility Guidelines version 2.2 (WCAG 2.2), a W3C Recommendation since October 2023, define testable success criteria at

levels A, AA and AAA; level AA is the usual legal and procurement benchmark. WCAG 2.2 added nine criteria, including minimum target sizes for pointer input and accessible authentication (no step of logging in may depend on a cognitive function test, such as remembering a password or transcribing a code, unless the user is offered an alternative method or help such as pasting from a password manager; a few other narrow exceptions apply). In the European Union, the European Accessibility Act (Directive (EU) 2019/882) has applied since 28 June 2025 to many consumer products and services, including e-commerce, banking services, e-books and passenger transport services. For an architect, accessibility is therefore a constraint to capture early: it affects the choice of front-end framework, the design system, the authentication flow and the testing pipeline (automated accessibility checks in continuous integration, plus manual testing with assistive technologies).

Interoperability

Interoperability is the degree to which two or more systems can exchange information and use the information they exchange. ISO/IEC 25010 places it under compatibility, alongside co-existence. It has two levels. *Syntactic* interoperability means the systems agree on format and protocol: both speak HTTP and JSON. *Semantic* interoperability means they agree on meaning: `"amount": 1050` is pence, not pounds; `status: "closed"` means paid, not cancelled. Most integration failures are semantic.

Industries reduce the cost of semantic agreement with shared standards: HL7 FHIR for exchanging healthcare records, ISO 20022 for financial messages, GTFS for public transport timetables. Architectural tactics include discovering services through a registry, tailoring interfaces with adapters and translators, and managing versions explicitly so that older partners keep working. An interoperability scenario might read: “A new hospital joins the network and exchanges patient summaries using FHIR R4 resources within 4 weeks, without changes to the core record service.”

Cost as a quality

Cost is rarely listed in quality models, yet it constrains every architectural decision. Separate **build cost** (the effort to create the system), **run cost** (infrastructure, licences, third-party services, support) and **change cost** (the effort for each future change, which modifiability is about). In the cloud, run cost is variable and directly shaped by architecture: chatty services pay for network transfer, over-provisioned clusters pay for idle capacity, and a careless query pattern can multiply a database bill.

A useful way to make cost a quality attribute is unit cost: cost per transaction, per active user or per order. Unlike a total monthly bill, it shows whether the architecture becomes more or less efficient as the business grows, and it can be written as a scenario: “At 3 million orders a

month, infrastructure cost stays below 1 pence per order." FinOps practices for managing this are covered in Chapter 10.

WORKED EXAMPLE

Worked example: unit cost and carbon per request for a food-delivery API

An order API runs on a cluster costing £24,000 a month and handles 3,000,000 orders a month. An analysis estimates its energy use at 1,200 kWh a month, on a grid averaging 250 g CO_2e per kWh, with 60 kg CO_2e of embodied hardware emissions allocated to it each month. The Software Carbon Intensity formula is SCI = ((E x I) + M) per R.

Cost per order	£24,000 / 3,000,000 = £0.008, so 0.8 pence per order
Operational emissions (E x I)	1,200 kWh x 250 g = 300,000 g = 300 kg CO_2e
Add embodied emissions (M)	300 + 60 = 360 kg CO_2e
Divide by the functional unit (R = one order)	360,000 g / 3,000,000 = 0.12 g CO_2e per order
Change: cache menus, cut energy by 25%	E = 900 kWh: (900 x 250 + 60,000) / 3,000,000 = 0.095 g per order
Change: run batch jobs in a region at 50 g per kWh	Lowers I for that share of the load; carbon-aware scheduling reduces the score without changing the code
Growth to 6,000,000 orders on the same cluster	Cost per order halves to 0.4 pence; carbon per order falls too, as long as energy grows less than in proportion (idle servers still draw power, so higher utilisation is more efficient)

Both metrics are rates per unit of useful work, so they reward efficiency rather than penalising growth.

The SCI specification counts only real reductions: buying offsets or renewable energy certificates does not lower the score.

Energy and grid figures are estimates; what matters is measuring them the same way over time so that changes can be compared.

Energy efficiency and sustainability

Software does not emit carbon itself, but the hardware it runs on consumes electricity and embodies the emissions of its manufacture. The Green Software Foundation's Software Carbon Intensity (SCI) specification, published by ISO as ISO/IEC 21031:2024, defines a rate: operational emissions (energy E times the carbon intensity I of the electricity) plus embodied emissions M, divided by a functional unit R such as a user, an API call or an order. It identifies three levers: **energy efficiency** (use less electricity for the same work), **hardware efficiency** (use fewer devices, and use them more fully and for longer) and **carbon awareness** (run flexible work when and where electricity is cleaner).

Architecture affects all three. Right-sizing and higher utilisation improve hardware efficiency; caching, efficient data formats and avoiding needless work improve energy efficiency; queues and batch processing make it possible to shift work in time or place. Sustainability often aligns with cost, but not always: keeping a warm standby in a second region improves availability and doubles the embodied emissions. Large AI models make the question more pressing, as Chapter 12 discusses.

Safety

Safety is freedom from unacceptable risk of harm to people, property or the environment. It differs from security: security protects the system from people with harmful intent; safety protects people from the system, including when nobody is malicious. ISO/IEC 25010:2023 added safety as a ninth characteristic, with sub-characteristics operational constraint, risk identification, fail safe, hazard warning and safe integration.

Safety-critical domains have their own standards, which impose architecture and process: IEC 61508 for functional safety in general, ISO 26262 for road vehicles, IEC 62304 for medical device software and DO-178C for airborne software. Typical tactics are a fail-safe default (a train stops when signals are lost), independent monitoring channels that can override the main controller, redundancy with voting, and strict separation of safety-critical components from everything else so that a fault in the infotainment system cannot reach the brakes. Note that fail-safe and highly available can conflict: a fail-safe system often stops, which is safe but unavailable.

Observability, an emerging quality

Observability is the degree to which you can understand a system's internal state from the outside, using the data it emits, well enough to answer questions you did not anticipate when it was built. It is not listed in ISO/IEC 25010, though analysability and accountability touch on it. In distributed systems it has become a quality in its own right, because recovery time, and

therefore availability, depends on how quickly people can find out what is wrong. Architecturally it requires structured logs, metrics and distributed traces with correlation identifiers that follow a request across services, designed in from the start. A scenario might be: “For any failed checkout, an on-call engineer can identify the failing service and the request's path within 5 minutes.” See observability in Chapter 9.

Legal and regulatory qualities

Laws and regulations create qualities that are easy to overlook until an audit. Data residency requires certain data to be stored or processed in a particular country or region; it may come from sector rules, public-sector contracts or customer demands. The EU GDPR itself does not require data to stay in the EU, but transfers outside it need a legal basis and safeguards, which in practice shapes where services and backups run. **Auditability** means being able to show, later, who did what and when: tamper-evident logs, retained for a defined period, with clocks synchronised. **Retention and deletion** rules require data to be kept for a minimum time (financial records) and deleted after a maximum time or on request (personal data), which is hard in systems with many copies, caches and backups. **Explainability** and human oversight are required for some automated decisions, as the EU AI Act chapter explains.

Treat these as constraints with acceptance criteria: "customer data for German public-sector clients is stored and processed only in EU regions, including backups and logs" is testable, and it rules out some managed services immediately. Privacy requirements are covered in privacy by design and the GDPR.

Summary: the qualities in this section and example measures

Quality	Example response measure	Typical architectural lever
Security (confidentiality)	No personal data readable from a stolen backup	Encryption at rest with separately managed keys
Security (accountability)	Every change to payee details attributable to one person, logs tamper-evident	Central audit log, write-once storage
Interaction capability	95% of first-time users complete checkout without help in under 3 minutes	Simple flows, fast responses, undo
Accessibility	All customer journeys meet WCAG 2.2 level AA	Accessible design system, automated checks in CI
Interoperability	New partner integrated using the published API standard within 4 weeks	Standard formats, adapters, versioned APIs
Cost	Infrastructure under 1 pence per order at 3 million orders a month	Right-sizing, caching, autoscaling
Sustainability	SCI below 0.1 g CO_2e per order	Efficiency, utilisation, carbon-aware scheduling
Observability	Failing service identified within 5 minutes for any failed request	Structured logs, metrics, distributed tracing
Data residency	All data for EU clients, including backups, stays in EU regions	Region-pinned storage and processing

CHECK YOUR UNDERSTANDING

▼ **What is the difference between accountability and non-repudiation?**

Accountability means actions can be traced to the entity that performed them; non-repudiation means there is proof strong enough that the entity cannot later deny the action.

▼ **Why does "undo the last five actions" count as an architectural requirement, not only a user-interface one?**

Because it requires operations to be modelled as reversible commands with stored history, which affects how state changes are structured throughout the system.

▼ **An API uses 800 kWh a month on a grid at 200 g CO2e per kWh, has 40 kg of embodied emissions allocated, and serves 2,000,000 requests. What is its SCI per request?**

(800 x 200 g + 40,000 g) / 2,000,000 = 200,000 / 2,000,000 = 0.1 g CO2e per request.

▼ **How does safety differ from security?**

Security protects the system and its data from people with harmful intent; safety protects people and the environment from harm the system could cause, even with no malicious actor.

▼ **Try it: A ticketing API costs £18,000 a month for 1,500,000 tickets and uses 800 kWh at 300 g CO2e per kWh, with 40 kg embodied emissions. Work out cost and SCI per ticket, then SCI if caching cuts energy 25%.**

Cost: £18,000 / 1,500,000 = 1.2 pence per ticket. Operational emissions: 800 x 300 g = 240 kg; adding 40 kg gives 280 kg, and 280,000 g / 1,500,000 = about 0.187 g per ticket. With caching: 600 kWh gives 180 kg, plus 40 kg is 220 kg, or about 0.147 g per ticket. Per-unit figures stay meaningful as volume grows, so track them rather than the total bill.

SOURCES

- W3C: Web Content Accessibility Guidelines (WCAG) 2.2: https://www.w3.org/TR/WCAG22/
- W3C WAI: What's new in WCAG 2.2: https://www.w3.org/WAI/standards-guidelines/wcag/new-in-22/
- Green Software Foundation: Software Carbon Intensity (SCI) specification: https://sci.greensoftware.foundation/
- Nielsen Norman Group: Response Times, the 3 Important Limits: https://www.nngroup.com/articles/response-times-3-important-limits/
- OWASP Application Security Verification Standard: https://owasp.org/www-project-application-security-verification-standard/

SECTION 2.8 OF 2.8 · INTERMEDIATE · 22 MIN READ · LAST REVIEWED NOVEMBER 2026

Eliciting, prioritising and trading off quality attributes

BEFORE YOU READ THIS

This section builds on:

- 1.5 Stakeholders and concerns
- 1.6 Decisions and trade-offs
- 2.3 Quality attribute scenarios

Terms it uses from earlier: Stakeholder, Concern, Quality attribute scenario, Percentile (p95, p99), Architecture Decision Record (ADR)

THE IDEA IN BRIEF

Planning a family holiday, everyone wants something different: one wants it cheap, another wants a pool, another wants short travel times. You cannot have everything, so you sit down together, list what each person really needs, vote on what matters most and agree what to give up. Architects do the same with a system's qualities: they gather everyone who cares, turn wishes into specific, measurable goals, rank them and write down the trade-offs they choose.

WHAT YOU NEED TO KNOW

- Quality requirements must be elicited from stakeholders, business goals, regulation, past incidents, competitors and measurements of the existing system, because they are rarely stated unprompted.
- The SEI Quality Attribute Workshop has eight steps; its outputs are architectural drivers and prioritised, refined scenarios, ranked by voting with a fixed vote budget.
- A utility tree refines utility into quality attributes, refinements and concrete scenarios, each rated (business importance, technical difficulty); (H,H) scenarios get the earliest design attention.
- Qualities conflict (performance and security, availability and consistency, modifiability and performance, cost and almost everything, usability and security), and the right resolution depends on context and business value.
- Record significant trade-offs in decision records and make each target measurable with a scale, a meter and levels, then track it over time.
- Fitness functions turn scenario response measures into automated checks that run as the system evolves.

Where quality requirements come from

Functional requirements usually arrive in a backlog. Quality requirements rarely do: stakeholders assume them ("of course it must be fast and secure") or do not know they need to state them. Architects therefore have to elicit them actively, from several sources.

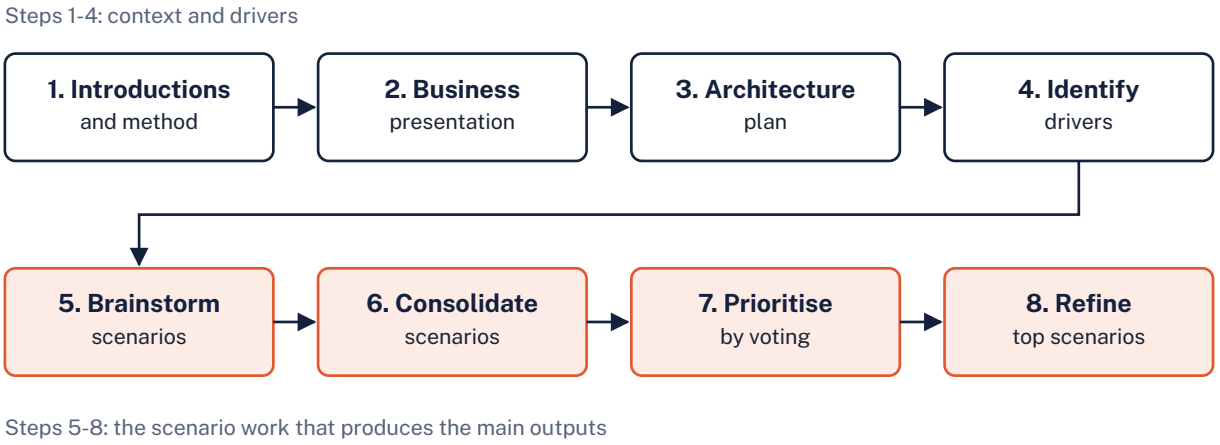
Stakeholders. Each group has different concerns: customers care about speed and ease of use, operations about monitoring and recovery, finance about run cost, support about diagnosing problems, compliance about audit trails. Chapter 1's stakeholder map is the starting point. **Business goals.** "Expand into three new countries next year" implies localisation, data residency and scalability; "cut customer support costs by 20%" implies usability and self-service. **Regulation and contracts.** Data protection, accessibility law, sector rules and customer SLAs create non-negotiable constraints. **Incidents.** Every past outage, breach or slow day reveals a quality that was under-specified; post-incident reviews are a rich source. **Competitors and the market.** If rivals load in one second, two seconds is a quality problem. **The existing system.** Its measured behaviour today (actual load, actual latency percentiles, actual change lead time) gives a baseline that turns "faster" into a number.

Interviews and questionnaires work, but quality attributes emerge best when stakeholders hear one another. A product owner who learns from the operations lead what an hour of downtime at Friday dinner time costs will prioritise differently. That is the idea behind structured workshops.

The Quality Attribute Workshop

The Quality Attribute Workshop (QAW), developed at the Software Engineering Institute (Barbacci and colleagues; the third edition of the method report appeared in 2003), is a facilitated session, typically a day long, held early, before the architecture is fixed. It brings 5 to 30 stakeholders together to produce prioritised, refined quality attribute scenarios. It has eight steps.

The eight QAW steps



The first four steps establish context and drivers; the last four produce, merge, rank and refine the scenarios.

What happens at each step

1. **1. Presentation and introductions**
   Facilitators explain the purpose and method; participants introduce themselves and their concerns.

2. **2. Business or mission presentation**
   A business representative describes the context, goals and high-level requirements: why the system exists and what matters most.

3. **3. Architectural plan presentation**
   A technical lead presents current plans, existing systems, known constraints and technology commitments, however early.

4. **4. Identification of architectural drivers**
   Facilitators distil the key architectural drivers heard so far (goals, qualities, constraints) and check them with the group.

5. **5. Scenario brainstorming**
   Participants propose scenarios in round-robin turns; facilitators make sure every driver has at least one, and that normal use, growth and exploratory (stress) scenarios are all represented.

6 **6. Scenario consolidation**
Near-duplicates are merged, but only when their proposers agree, so that votes are not split.

7 **7. Scenario prioritisation**
Each participant receives a vote budget of about 30% of the number of scenarios and spends it in two rounds, any number of votes per scenario. The tally ranks the scenarios.

8 **8. Scenario refinement**
The top four or five scenarios are written out in full: source, stimulus, artifact, environment, response and response measure, with the business goals they serve and open questions.

Teams that cannot spare a full day often run a shorter "mini-QAW", a lighter variant described by practitioners such as Michael Keeling, Will Chaparro and Thijmen de Gooijer, using a quality attribute taxonomy as a prompt and a few hours with a smaller group. The essentials survive: stakeholders together, scenarios not adjectives, explicit voting, and refinement of the winners.

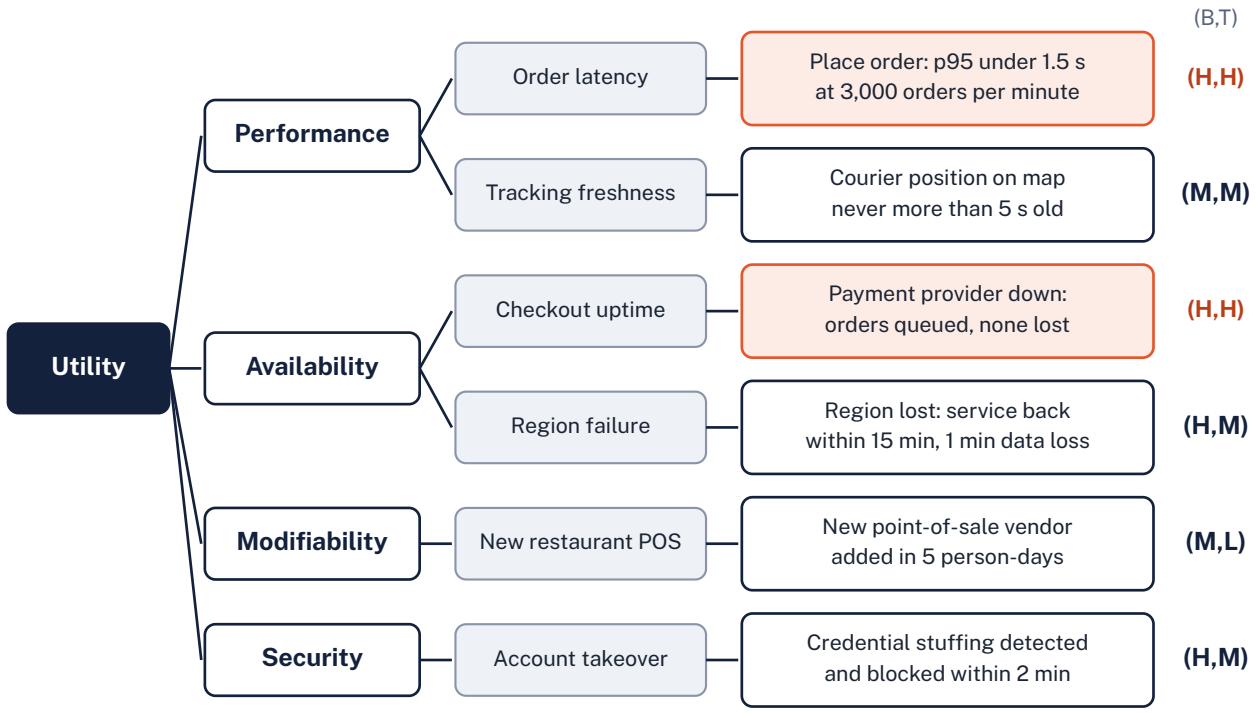
The utility tree

A utility tree is a way to organise and prioritise quality requirements, introduced as part of the Architecture Tradeoff Analysis Method (ATAM), which Rick Kazman, Mark Klein and Paul Clements developed at the SEI; the evaluation method itself is covered in Chapter 11. The root is *utility*, the overall goodness of the system. The second level holds quality attributes (performance, availability and so on). The third level refines each into more specific concerns (for performance: order latency, tracking freshness). The leaves are concrete, measurable scenarios.

Each leaf is rated with a pair: **business importance** (how much the scenario matters to the business: High, Medium or Low) and **technical difficulty** or risk (how hard it is to achieve with the architecture: High, Medium or Low). The pair is written (B,T), for example (H,M). Scenarios rated (H,H) are the architecture's top priorities: they matter most and are hardest, so they deserve early design effort, prototypes and analysis. (H,L) scenarios matter but are easy, so a standard approach will do; (L,H) scenarios are candidates to drop or postpone, because they are hard but not worth much.

The tree is useful even outside a formal evaluation. It makes trade-offs visible on one page, shows gaps (a quality with no scenarios), and stops the loudest stakeholder from setting priorities alone.

A utility tree for a food-delivery platform



Two scenarios rated (H,H), highlighted, are the architecture's top priorities; the (M,L) modifiability scenario can be met with ordinary good design.

WORKED EXAMPLE

Worked example: building the utility tree for a food-delivery platform

A food-delivery platform connects customers, 4,000 restaurants and 9,000 couriers in one country. The business plans to double order volume within 18 months. The Friday evening peak is 3,000 orders per minute today. A two-hour QAW produced 26 scenarios; with 12 participants, each had 8 votes (30% of 26 is 7.8, rounded up to an even 8). The top-voted scenarios became leaves of the tree.

Performance: order latency (H,H)	Customers place orders at the peak rate of 3,000 per minute; 95% of order confirmations return within 1.5 s and 99% within 3 s. High importance: abandoned baskets are lost revenue. High difficulty: the request crosses basket, pricing, payment and restaurant services
Capacity check for that scenario	3,000 orders/min = 50 orders/s. Doubling volume gives 100 orders/s. Little's law needs the mean time, not a percentile (Section 2.4), so the 1.5 s target gives only a cautious planning figure: if the mean were as high as 1.5 s, about 100 x 1.5 = 150 orders would be in progress at once; the real average is lower because the mean is well below the p95
Availability: checkout uptime (H,H)	When the payment provider times out during the peak, orders are accepted and queued for payment, no confirmed order is lost, and customers are told within 10 s. Difficult because it needs asynchronous payment, idempotent retries and a user experience for "pending"
Availability: region failure (H,M)	If the cloud region fails, service resumes in a second region within 15 minutes (RTO) with no more than 1 minute of order data lost (RPO). Important but technically well understood
Security: account takeover (H,M)	A credential-stuffing attack is detected and blocked within 2 minutes; no account protected by multi-factor authentication is taken over
Performance: tracking freshness (M,M)	A courier's position on the customer's map is never more than 5 s old under peak load
Modifiability: new POS vendor (M,L)	A new restaurant point-of-sale vendor is integrated in 5 person-days by adding one adapter (Section 2.6)
Resulting priorities	Prototype and load-test the order path and the queued-payment design first; use standard multi-region patterns for region failure; keep the adapter interface simple

Ratings are judgements agreed by the group, not calculations. Record who rated what and why, so the ratings can be challenged later.

The tree drives the work plan: the two (H,H) scenarios get architectural spikes in the first sprints, while the (M,L) scenario needs no special attention.

Courier tracking was rated only medium in importance, which surprised the engineers; the product owner explained that customers mostly check the estimated arrival time, not the map.

Prioritising when everything is "critical"

Every stakeholder will rate their own concern as high. Techniques that force real choices help. A fixed vote budget (as in the QAW) makes people spend scarce votes. Pairwise comparison ("if you could have only one, sub-second search or zero data loss on failover?") exposes the true order. Putting money or time next to each target ("four nines will cost an extra £9,000 a month and two months of work") changes many opinions. And distinguishing *constraints* (legal accessibility requirements, data residency) from *preferences* stops non-negotiables being traded away by accident.

Priorities are not fixed. A start-up seeking its first customers rightly ranks time to market and modifiability above five-nines availability; the same product, three years later with enterprise clients and contractual SLAs, must rerank. Revisit the utility tree when the business changes.

Conflicts between qualities

Qualities interact. Some reinforce each other: modularity helps testability, which helps deployability. Many conflict, because the tactic that helps one quality hurts another. The ATAM, described by Bass, Clements and Kazman, calls the points where an architectural decision affects several qualities trade-off points. Recognising the usual conflicts lets you raise them before they surprise anyone.

Common conflicts between qualities, and how context resolves them

Conflict	Why they pull apart	Ways to resolve it in context
Performance vs security	Encryption, authentication checks, input validation and audit logging add work to every request	Measure the real overhead (often small with modern TLS and hardware); cache authorisation decisions briefly; push expensive checks to the edge; never trade away security on data that would cause serious harm if exposed
Availability vs consistency	During a network partition a replicated system must choose between answering (possibly with stale data) and refusing	Decide per operation: a menu or courier position can be slightly stale; a payment or stock reservation cannot. See CAP and PACELC in Chapter 6
Modifiability vs performance	Layers, interfaces, intermediaries and late binding add indirection and hops	Keep the indirection where change is likely; optimise the measured hot path; accept a few deliberate shortcuts, documented in a decision record
Cost vs nearly everything	Redundancy, extra regions, more testing and faster hardware all cost money	Tie each quality target to its business value; use error budgets and unit cost; buy nines only where an outage costs more than preventing it
Usability vs security	Strong authentication, timeouts and confirmations add friction	Risk-based authentication (step up only for risky actions), passkeys and single sign-on, sensible session lengths; involve users in testing
Time to market vs maintainability	Shortcuts speed the first release and slow later ones	Take on technical debt deliberately, record it and schedule its repayment (Chapter 1)

The right resolution is always contextual. "Security beats performance" is a slogan, not a decision: an extra 2 ms of TLS handshake on a public website is irrelevant, while synchronous fraud checks on every card swipe at a busy stadium may be a real problem worth an architectural answer (an approve-first, check-after model with limits). Ask: which business goal does each side serve, what does each option cost, and who must agree?

IF YOU ARE NEW TO THIS

Step by step: one conflict, two answers

The step most people trip over here is that a conflict between two qualities is not settled once for the whole system. It is settled operation by operation, by what each is worth to the business. Follow the table's availability-versus-consistency row through the food-delivery platform.

- Suppose the platform keeps copies of some of its data on more than one machine, so that it can survive a failure, and passes each change from one copy to the others.
- Now suppose the network link between two copies breaks for a while: a network partition. A copy that cannot hear about the latest changes can either keep answering with what it has, which may be out of date, or refuse until the link returns. It cannot promise both. Chapter 6 returns to this choice.
- Courier position: a dot on the customer's map that is a few seconds old does little harm, and customers mostly check the estimated arrival time anyway. Availability wins: keep answering.
- Payment: acting on out-of-date information could mean charging a customer twice, or treating an order as paid when the payment never went through. Consistency wins: hold the payment back and show the order as pending rather than guess.
- So one platform gives opposite answers to the same conflict. Neither comes from a slogan such as 'consistency beats availability'; each follows from what a wrong answer would cost for that operation.
- **The lesson:** when two qualities conflict, ask which business goal each side serves for this particular operation, and decide there. Then record the choice and its reasons, as the next part of this section explains, so that nobody later reverses it without knowing why.

Recording decisions and tracking qualities

A trade-off that is not written down will be argued again, often by people who were not in the room. Record significant ones as architecture decision records (Chapter 4): the context, the qualities in tension, the options considered, the decision and its consequences, including what you have consciously given up.

Then make each quality target measurable and visible. Tom Gilb's Planguage offers a disciplined format: a *scale* (what is measured, in which unit), a *meter* (how it is measured), and levels such as *fail* (the threshold below which the requirement is not met), *goal* (the target) and *stretch* or *wish* (more ambitious levels). Applied to the order-latency scenario: scale, seconds from submit to confirmation at the 95th percentile; meter, real-user monitoring during the Friday peak; fail above 3 s; goal 1.5 s; stretch 1 s.

Finally, track the measures over time on a dashboard that stakeholders can see, alongside SLOs (Chapter 9), and revisit them whenever the business, the load or the team changes.

From scenarios to fitness functions

The strongest way to keep a quality requirement true is to check it automatically, every time the system changes. Neal Ford, Rebecca Parsons and Patrick Kua, in *Building Evolutionary Architectures* (2017; second edition with Pramod Sadalage, 2022), call such a check an architectural fitness function: an objective assessment of how well the system meets an architectural characteristic. A load test in the pipeline that fails the build if p95 order latency exceeds 1.5 s at 100 orders per second is a fitness function. So is a test that fails if any module outside `payments` imports the payment provider's library (protecting the modifiability scenario), a scheduled failover drill that measures recovery time, or a dependency scan that blocks known vulnerable libraries.

Fitness functions are the natural end point of the work in this chapter: a vague wish becomes a scenario (Section 2.3), the scenario gets a priority in the utility tree, and the scenario's response measure becomes an automated check. Not every quality can be automated cheaply (usability still needs people), but each one you automate stops silent erosion. Chapter 11 covers their design in depth: fitness functions and evolutionary architecture.

Checklist: running a quality attribute workshop

Adapted for a half-day or one-day session; a shorter "mini-QAW" can cover the same ground with a smaller group.

- [ ] **Invite the right mix**
  5 to 30 people: business owners, users or their representatives, operations, security, support, developers, and anyone who signs off compliance. Missing operations is the most common gap.
- [ ] **Send a short pre-read**
  The business goals, known constraints and a draft list of quality attributes (ISO/IEC 25010 as a prompt), so people arrive thinking.
- [ ] **Open with business goals, not technology**
  A senior stakeholder explains why the system exists, what success looks like and what failure would cost.
- [ ] **Agree the architectural drivers**
  A short list of the goals, qualities and constraints that will shape the architecture, checked with the group before scenarios start.
- [ ] **Brainstorm scenarios in rounds**
  Everyone contributes at least two, in turn, so quiet voices are heard. Include normal use, growth ("in two years") and stress ("when the payment provider is down").
- [ ] **Merge duplicates only with consent**
  Consolidation keeps votes from being split, but the proposer must agree that two scenarios really mean the same thing.
- [ ] **Vote with a fixed budget**
  Each person gets votes equal to about 30% of the scenario count, spent in two rounds. This exposes real priorities and avoids "everything is critical".
- [ ] **Refine the top few fully**
  Write the top four or five as six-part scenarios with measurable response measures, and note the business goal each one serves.
- [ ] **Record open questions and conflicts**
  Disagreements about priority or feasibility are valuable outputs; list them with an owner.

- [ ] **Publish the result within days**
  Share the prioritised scenarios, add them to the architecture documentation and the backlog, and plan when each will be checked.

WATCH OUT

Common mistakes

- Collecting qualities only from the business sponsor: operations, security and support concerns then surface after go-live, when they are expensive.
- Accepting adjectives: "scalable" and "robust" on a requirements list without a scenario behind them.
- Rating every leaf (H,H): if everything is top priority, nothing is. Push back until the ratings discriminate.
- Treating the utility tree as a one-off artefact: it should be revisited when goals or load change.
- Resolving conflicts silently in code: a developer who quietly drops audit logging to meet a latency target has made an architectural decision that nobody recorded.

CHECK YOUR UNDERSTANDING

▼ In a QAW with 20 consolidated scenarios, about how many votes does each participant get?

About 30% of 20, which is 6 votes, spent in two rounds of 3.

▼ A utility-tree leaf is rated (L,H). What does that suggest?

It matters little to the business but is hard to achieve, so it is a candidate to drop, defer or renegotiate rather than drive the architecture.

▼ Why is "security always beats performance" a poor way to resolve a conflict?

Because the right answer depends on the actual overhead, the harm the security control prevents and the business value of the performance; the conflict must be resolved per case with numbers and recorded.

▼ Give one example of a fitness function for a modifiability requirement.

An automated test in the build that fails if any module other than the payment adapter imports the payment provider's library, keeping the provider replaceable.

▼ Try it: Stakeholders call everything critical. Rate these (importance, difficulty) for a food-delivery utility tree: order confirmation within 1.5 s at peak; monthly report export within an hour; adding a payment provider within a month. Which gets earliest design attention?

Order confirmation is (H,H): lost baskets are lost revenue, and the request crosses several services under peak load, so it gets design attention first. A new payment provider might be (M,M): valuable, and moderately hard unless an adapter already exists. The report export is (L,L) or (M,L): easily run overnight. Ratings force a ranking without arguing that one quality is always worth more. Record the reasons and check them with stakeholders by vote.

SOURCES

- SEI: Quality Attribute Workshop collection: https://www.sei.cmu.edu/library/quality-attribute-workshop-collection/
- SEI: Architecture Tradeoff Analysis Method (ATAM) collection: https://www.sei.cmu.edu/library/architecture-tradeoff-analysis-method-collection/
- IREB RE Magazine: Discover Quality Requirements with the Mini-QAW: https://re-magazine.ireb.org/articles/discover-quality-requirements-with-the-mini-qaw
- Thoughtworks Technology Radar: Architectural fitness function: https://www.thoughtworks.com/radar/techniques/architectural-fitness-function

Further reading

- *Software Architecture in Practice (4th edition)*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. The source of quality attribute scenarios and tactics, with a chapter on each major quality and one on architecturally significant requirements.
  https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- *ISO/IEC 25010:2023 Systems and software engineering: Product quality model*, ISO/IEC JTC 1/SC 7 (2023), International Organization for Standardization. The standard product quality model with nine characteristics; its companion ISO/IEC 25019:2023 covers quality in use.
  https://www.iso.org/standard/78176.html
- *Characterizing Architecturally Significant Requirements*, Lianping Chen, Muhammad Ali Babar and Bashar Nuseibeh (2013), IEEE Software, volume 30, issue 2. An empirical study of how practitioners recognise the requirements that shape an architecture.
  https://pure.itu.dk/en/publications/characterizing-architecturally-significant-requirements/
- *The Tail at Scale*, Jeffrey Dean and Luiz André Barroso (2013), Communications of the ACM, volume 56, issue 2. A short, readable paper on why rare slow responses dominate large fan-out systems and the techniques Google uses to tame them.
  https://research.google/pubs/pub40801/
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. Clear treatment of load, percentiles, reliability and scalability in data systems, with many real-world examples.
- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. A practitioner's view of identifying and prioritising architecture characteristics and protecting them with fitness functions.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (eds) (2016), O'Reilly Media. Free to read online; the chapters on service level objectives and monitoring show how qualities are measured in production.
  https://sre.google/books/
- *Systems Performance: Enterprise and the Cloud (2nd edition)*, Brendan Gregg (2020), Addison-Wesley. The standard reference on performance analysis, from queueing and utilisation to the USE method.

Glossary

Accessibility

The degree to which people with disabilities, including those using assistive technologies such as screen readers, can use a product.

Amdahl's law
A formula (Gene Amdahl, 1967) showing that the speed-up from adding processors is limited by the part of the work that must run serially.

Architectural driver
A goal, quality attribute requirement or constraint that significantly shapes the architecture.

Architecturally significant requirement (ASR)
A requirement that has a measurable effect on a system's architecture: if it changed, the structure would have to change. Most ASRs are quality attributes or constraints.

Availability
The proportion of time, or of requests, for which a system is ready to deliver its service when asked, usually written as a percentage such as 99.9%.

Binding time
The moment at which a decision is fixed: when code is written, at build, at deployment, at start-up or while the system runs.

CIA triad
The three classic goals of information security: confidentiality (only authorised parties can read), integrity (no unauthorised changes) and availability (usable when needed).

Concrete scenario
A scenario written for one particular system, with specific actors, events, conditions and numbers, ready to be designed for and tested.

Constraint
A design decision that has already been made for you, with zero or very little freedom left, such as a required programming language, a regulation, a fixed deadline or a budget.

Data residency
A requirement that certain data be stored, and sometimes processed, only in a particular country or region.

Dependency injection
Giving a component the collaborators it needs (a database, a clock, a payment gateway) from outside instead of letting it create them, so they can be swapped, for example for fakes in tests.

Deployability
How quickly, safely and predictably a change can be put into production, and taken back out again if needed.

DORA metrics
The software delivery performance measures from the DORA research programme: change lead time, deployment frequency, failed deployment recovery time, change fail rate and deployment rework rate.

Elasticity
The ability to add and remove resources automatically as load rises and falls, so capacity follows demand and you pay only for what you use.

Elicitation
Actively drawing requirements out of stakeholders, documents, regulations and existing systems, rather than waiting for them to be stated.

Error
Part of a system's internal state that is wrong because a fault has been activated, such as a corrupted record. An error may or may not lead to a failure.

Failover
Switching work from a failed component to a healthy standby or replica, automatically or by hand.

Failure
An event in which the service a system delivers deviates from what it should be, in a way its users can observe.

Fan-out
A pattern in which one incoming request triggers several parallel requests to other services or servers, whose answers are then combined.

Fault
The cause of a problem, such as a bug, a failed disk or a misconfiguration. A fault may lie dormant until something activates it.

Fault tolerance
The ability of a system to keep delivering its service even though one of its parts has failed, usually through redundancy.

Functional requirement
A statement of what a system must do: a behaviour, calculation or piece of information it must provide, such as 'a customer can cancel an order before it is dispatched'.

FURPS+
A quality classification that originated at Hewlett-Packard as FURPS (functionality, usability, reliability, performance and supportability), later extended with '+' for constraints such as design, implementation, interface and physical limits.

General scenario
A system-independent template listing the typical sources, stimuli, environments, responses and measures for one quality attribute.

Graceful degradation
Continuing to provide the most important functions when part of a system fails, instead of failing completely, for example a shop that still takes orders while recommendations are offline.

Heartbeat
A periodic "I am alive" message sent by a component; if the messages stop, a monitor assumes the component has failed and acts.

Implicit requirement
A quality that every stakeholder assumes but nobody writes down, such as 'orders are never lost' or 'the system keeps working across a clock change'.

Interoperability
The degree to which systems can exchange information and correctly use what they exchange, which needs agreement on meaning as well as format.

ISO/IEC 25010
The international standard product quality model. The 2023 edition defines nine characteristics, from functional suitability to safety, each divided into sub-characteristics.

Little's law
The rule $L = \lambda \times W$: the average number of items in a stable system equals the average arrival rate multiplied by the average time each item spends in it.

Maintainability
In ISO/IEC 25010, how effectively and efficiently a product can be modified, covering modularity, reusability, analysability, modifiability and testability.

Mean time between failures (MTBF)
The average time from one failure of a repairable component or system to the next.

Mean time to failure (MTTF)
The average time a component runs correctly before it fails.

Mean time to repair (MTTR)
The average time from a failure until the service is restored. Some organisations read the R as recover or restore; always say which start and end points are measured.

Modifiability
How cheaply and safely a system can be changed: the cost, effort and risk of a change, and how far it ripples beyond the place where it is made.

Non-functional requirement (NFR)
A widely used but imprecise name for quality requirements and constraints. Many architects avoid it because these requirements qualify functions rather than standing apart from them.

Observability
How well people can understand what is happening inside a running system from the data it emits (logs, metrics, traces), including for problems nobody anticipated.

Percentile (p95, p99)
The value below which a given share of measurements fall. A p99 of 400 ms means 99% of requests took 400 ms or less and 1% took longer.

Quality attribute scenario
A short, testable statement of a quality requirement with six parts: source of stimulus, stimulus, artifact, environment, response and response measure.

Quality Attribute Workshop (QAW)
A facilitated SEI method in which stakeholders brainstorm, consolidate, vote on and refine quality attribute scenarios early in a project.

Quality in use
The quality a product delivers to people in a specific context of use (whether it is beneficial, free from risk and acceptable). Since 2023 it has its own standard, ISO/IEC 25019.

Quality model
A structured list of quality characteristics and their sub-characteristics, used as shared vocabulary and as a checklist when specifying or evaluating quality.

Queueing
Waiting for a busy resource. Queueing theory studies how waiting time grows with utilisation; it explains why response time rises sharply as a resource nears 100% busy.

Recovery point objective (RPO)
The largest amount of data, measured as a period of time, that the business can accept losing in a disaster, for example the last 15 minutes of changes.

Recovery time objective (RTO)
The longest time the business can accept between a disaster and the restoration of the service.

Redundancy
Keeping more than one copy of a component so that another can take over if one fails.

Reliability
The probability that a system performs correctly, without failure, for a stated period under stated conditions. A system can be highly available yet unreliable if it fails often but recovers quickly.

Resilience
The broader capacity of a system to absorb disruption, keep its essential service going (perhaps in a degraded form) and recover to normal afterwards.

Response measure
The number that makes a scenario testable, such as '99% within 800 ms', 'service restored within 2 minutes' or 'change completed in 3 person-days'.

Response time
The total time between a client sending a request and receiving the answer: the service time plus any waiting time in queues, plus network delays.

Rollback
Returning a system to an earlier known good state or release after a change causes problems.

Safety
Freedom from unacceptable risk of harm to people, property or the environment caused by the system, whether or not anyone acts maliciously.

Scalability
A system's ability to handle more load by adding resources, ideally with a predictable and roughly proportional increase in capacity and cost.

Service level agreement (SLA)
A contract with customers that states a service level and the consequences, such as service credits, if it is not met.

Service level indicator (SLI)
A carefully defined measurement of service behaviour, such as the proportion of requests that succeed in under 300 ms.

Service level objective (SLO)
An internal target for an SLI over a time window, such as 99.9% of requests successful over 30 days.

Service time
The time a resource spends actually working on a request, excluding any time the request spends waiting for that resource.

Software Carbon Intensity (SCI)
A Green Software Foundation specification, published as ISO/IEC 21031:2024, that expresses software's carbon emissions as a rate: ((energy x grid carbon intensity) + embodied emissions) per functional unit.

Stimulus
The event that a quality attribute scenario is about, such as a burst of requests, a server crash, a change request or an attack.

Sub-characteristic
A more specific quality within a characteristic of a quality model; for example, fault tolerance and recoverability are sub-characteristics of reliability in ISO/IEC 25010.

Tactic
A design decision that influences how a system responds to a quality stimulus, such as caching to cut latency or redundancy to survive a crash. Tactics are the building blocks of patterns.

Tail latency
The response times of the slowest requests, usually described by high percentiles such as p99 or p99.9. At scale the tail often decides what users experience.

Test double
A stand-in for a real collaborator used in a test, such as a stub with canned answers, a fake with a simple working implementation, or a mock that checks the calls it receives.

Testability
How easily a system or component can be shown, by testing, to behave as intended, which depends on being able to control its inputs and state and observe its outputs.

Throughput
The amount of work a system completes per unit of time, such as 1,200 requests per second or 40,000 orders per hour.

Trade-off point
An architectural decision that affects more than one quality attribute, improving one while harming another.

Unit cost
The cost of running a system divided by a unit of useful work, such as cost per order or per active user, so efficiency can be tracked as the business grows.

Universal Scalability Law (USL)
Neil Gunther's model of how capacity grows with added nodes, accounting for contention for shared resources and the cost of keeping data coherent. It predicts a peak beyond which adding nodes reduces capacity.

Utilisation
The fraction of time a resource is busy, usually written ρ (rho). A server handling 40 requests a second that each need 20 ms of work is 80% utilised.

Utility tree
A tree, from the ATAM, that refines overall utility into quality attributes and then concrete scenarios, each rated by business importance and technical difficulty.

WCAG
The Web Content Accessibility Guidelines from the W3C. Version 2.2 (2023) sets testable success criteria at levels A, AA and AAA; level AA is the usual legal benchmark.