

BE SOFTWARE ARCHITECTURE WISE · CHAPTER 1

What Software Architecture Is

This chapter explains what software architecture is and why it matters. It compares the standard definitions, separates architecture from design, shows through real incidents how structural decisions drive cost, quality and risk, and describes what architects do. It then covers stakeholders and their concerns, decisions and trade-offs, technical debt and erosion, and where architecture fits in modern iterative delivery. It is written for developers moving towards architectural responsibility and for anyone who works with architects, and it assumes only the building blocks from Chapter 0.

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

AT A GLANCE

You will learn to

- Explain the ISO/IEC/IEEE 42010:2022 and SEI definitions of software architecture and the ideas they share.
- Distinguish module, component-and-connector and allocation structures, and an architecture from its description.
- Classify decisions as architectural or design using reversibility, scope and effect on quality attributes.
- Analyse how structural decisions contributed to real incidents such as Knight Capital and HealthCare.gov.
- Describe the responsibilities, skills and kinds of architect, and how architects stay hands-on.
- Identify stakeholders and their concerns, and make conflicting concerns explicit.
- Evaluate architectural decisions as trade-offs, considering reversibility and timing.
- Explain technical debt, erosion and drift, and how to detect and manage them across the lifecycle.

Before you start

Chapter 0, or a working knowledge of components, databases, APIs, networks and the cloud.

Time

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

Key terms

Software architecture · Architecture description · Structure · Architecturally significant · Quality attribute · Cost of change · Stakeholder · Concern · Trade-off · Reversibility · Technical debt · Architecture erosion · Fitness function · Conway's law

Why this chapter comes first

Every software system has an architecture, whether anyone designed it or not. It is the set of fundamental structures that make up the system (its main parts, how they connect and what properties they have) and the significant decisions that produced them. Those structures decide far more than most people realise: how fast and reliable the system is, how much it costs to run, how many teams can work on it at once, and, above all, how expensive it will be to change next year. Features are what a system does; architecture largely decides how well it does them and what it costs to do something different.

Chapter 0 introduced the building blocks: programs, data stores, networks, APIs, servers and the cloud. This chapter steps back and asks what it means to arrange those blocks deliberately, who does it, and why it is worth the effort. It gives you the vocabulary that the rest of the course depends on, and a way of thinking that you can apply to any system, from a small booking app to a national payments platform.

What architecture is, and what it is not

The first two sections are about definitions, because muddled definitions lead to muddled work. Section 1.1 compares the definitions that matter: the international standard ISO/IEC/IEEE 42010:2022, which speaks of the fundamental concepts or properties of a system in its environment and the principles that govern its evolution; the Software Engineering Institute's definition, which speaks of the structures needed to reason about a system, made of elements, relations and their properties; and practitioners' definitions that stress shared understanding and decisions that are expensive to change. It shows what they have in common, and introduces three kinds of structure every system has: how its code is divided, what runs and talks to what, and how software maps onto infrastructure and teams.

Section 1.2 then tackles the most common confusion: where architecture ends and design begins. The answer is that there is no wall, only a continuum ordered by significance. A decision is architecturally significant when it is costly to reverse, wide in scope or decisive for a key quality, which is why a humble convention such as how money amounts are stored can be more architectural than a large box on a diagram. The section also separates the levels at which the word is used: enterprise, solution and software architecture, with data, security and infrastructure architecture cutting across them.

Why it matters and who does it

Section 1.3 makes the case for caring. It shows how quality attributes such as availability and performance emerge from structure rather than from individual lines of code, how the cost of change grows as more is built on a decision, and how architecture shapes risk, team organisation (through Conway's law), communication and estimation. Three real cases, told from primary sources, ground the argument: the Knight Capital trading loss of August 2012, the troubled HealthCare.gov launch of October 2013, and Amazon Prime Video's 2023 decision to move one monitoring service from distributed serverless steps to a single process, which reportedly cut its cost by over 90% and illustrates that architecture is about fit, not fashion.

Section 1.4 looks at the people. It describes what architects actually do, using the eight expectations set out by Mark Richards and Neal Ford and the tasks in the iSAQB Foundation curriculum; the different kinds of architect; why architecture is a role before it is a job title; why architects value breadth of knowledge; and the long-running debate about whether architects should still write code.

Section 1.5 widens the lens to stakeholders (everyone with an interest in the system, from users and developers to operations staff, data protection officers and regulators) and their concerns, and shows how conflicting concerns are made explicit rather than settled by whoever shouts loudest. Section 1.6 turns to decisions and trade-offs: the anatomy of a good

decision, the difference between decisions that can easily be reversed and those that cannot, the idea of deciding at the last responsible moment, and the cognitive traps that lead capable people astray.

Architecture over time

The last two sections are about time. Section 1.7 explains technical debt as Ward Cunningham originally meant it in 1992, how it differs from mere mess, and how architectures erode and drift away from their intended shape; it shows how to detect the damage and how to explain it in business terms. Section 1.8 places architecture in the delivery lifecycle: the old argument between big design up front and purely emergent design, and the practical middle ground of risk-driven, continuous and evolutionary architecture in which a team does just enough architecture, at the right time, and keeps checking that it still fits.

The one idea to take away

If you remember one thing from this chapter, make it this: architecture is the set of decisions that are expensive to get wrong, and the architect's job is to identify those decisions, make them deliberately with the right people, record why, and keep them under review, while making as many other decisions as possible cheap to change. Everything else in this course, from quality attributes in Chapter 2 to the design of AI-enabled systems in Chapter 12, is an application of that idea.

SECTION 1.1 OF 1.8 · BEGINNER · 21 MIN READ · LAST REVIEWED NOVEMBER 2026

What software architecture is: the standard definitions and what they share

THE IDEA IN BRIEF

Think of a hospital building. Its architecture is not the blueprint pinned on the wall; it is how the building is actually arranged: where the wards are, which corridors connect them, where the lifts go, which doors must stay locked, and the rules everyone follows when they extend it. Some choices, like where the lifts are, would cost a fortune to change; the colour of the curtains would not. Software architecture is the same idea for a software system: its main parts, how they connect, what properties they have, and the expensive-to-change decisions behind them.

WHAT YOU NEED TO KNOW

- ISO/IEC/IEEE 42010:2022 defines architecture as the fundamental concepts or properties of an entity in its environment, plus the governing principles for its realisation and evolution.
- Bass, Clements and Kazman define software architecture as the set of structures needed to reason about a system: software elements, the relations among them and the properties of both.
- Practitioner definitions add that architecture is the shared understanding of expert developers (Ralph Johnson) and the significant decisions, where significance is measured by cost of change (Grady Booch).
- The definitions share seven ideas: structures, elements, relations, properties, decisions, significance and environment.
- Systems have three kinds of structure: module structures (code), component-and-connector structures (runtime) and allocation structures (mapping to infrastructure, teams and files).
- Every system has an architecture whether or not it was designed deliberately or written down; an architecture description is a separate work product that expresses it.

Why the word needs defining at all

Ask ten experienced engineers what software architecture is and you will hear ten different answers: the boxes-and-arrows diagram on the wall, the choice of cloud provider, the database schema, the split into services, the rules about which code may call which, or simply "the hard

decisions". Each answer captures something real, and each, taken alone, misses something. A course that is going to teach architecture therefore has to start by being precise about what it is talking about.

This matters for practical reasons. If a team thinks architecture means "the diagram", it will stop paying attention once the diagram is drawn, even though the system keeps changing underneath it. If it thinks architecture means "the technology stack", it will argue about frameworks while ignoring how responsibilities are divided and how data flows. A good definition tells you what to look at, what to protect and what to discuss with others.

The definitions below come from an international standard, from the Software Engineering Institute (SEI) at Carnegie Mellon University, and from practitioners. They were written for different purposes, but they agree on more than they disagree, and the points where they agree are the core of the discipline.

The standard definition: ISO/IEC/IEEE 42010:2022

ISO/IEC/IEEE 42010 is the international standard for describing architectures. Its current edition, published in November 2022, replaced the 2011 edition. It defines architecture, in close paraphrase, as the fundamental concepts or properties of an entity in its environment, together with the governing principles for that entity's realisation and evolution and for its related life-cycle processes.

Three features of this definition deserve attention. First, it talks about an *entity*, not only a software system. The 2022 edition deliberately widened the scope so that the same ideas apply to software, systems of systems, products, product lines and whole enterprises. Second, it places the entity *in its environment*: you cannot understand an architecture without the users, other systems, organisation, laws and budgets around it. Third, it includes *governing principles* for how the entity is built and how it changes. Architecture is not only a snapshot of structure; it includes the rules that keep the structure coherent as the system evolves, such as "every service owns its own data" or "the domain code never depends on the web framework".

The standard also draws a sharp line between an architecture and an architecture description. The architecture is the set of fundamental properties the system actually has. The description is a work product (documents, models, diagrams, decision records) that expresses it. ISO/IEC/IEEE 42010 is mainly a standard about descriptions: it says what a good description should contain, such as identified stakeholders, their concerns and views that address those concerns. It does not tell you how to design an architecture. Section 1.5 returns to stakeholders and concerns, and Chapter 4 covers views.

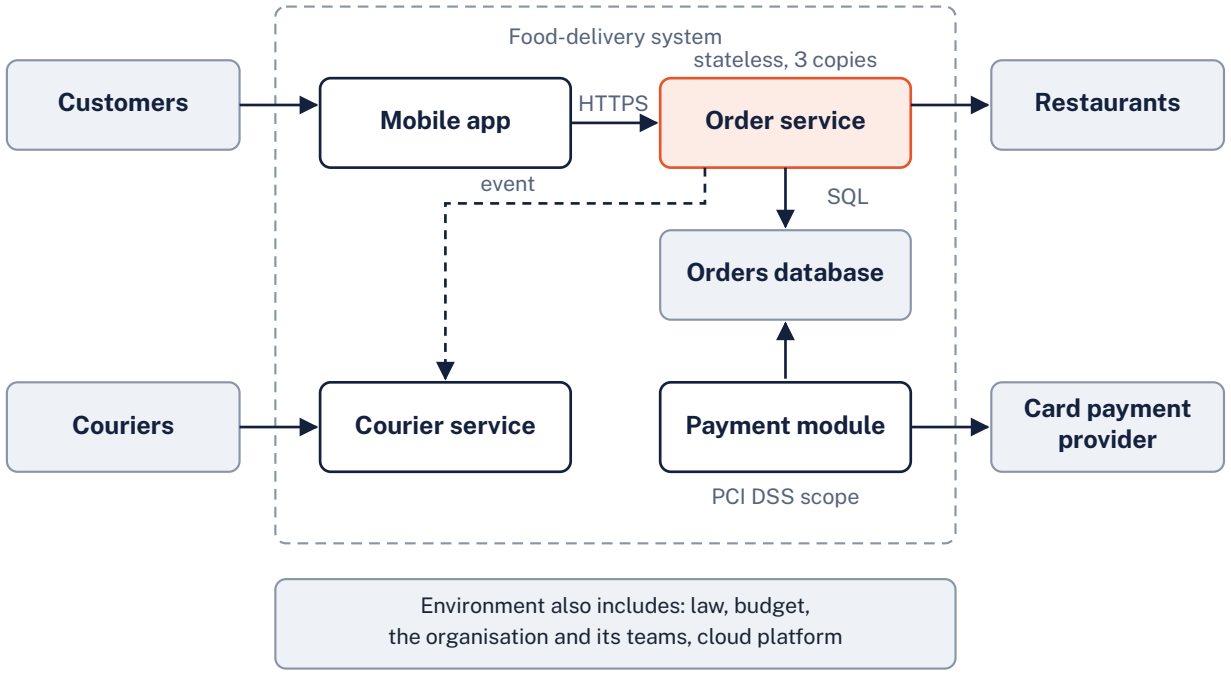
The SEI definition: structures, elements, relations and properties

The most widely taught definition comes from *Software Architecture in Practice* by Len Bass, Paul Clements and Rick Kazman (4th edition, Addison-Wesley, 2021), whose authors worked at the SEI. In their words, paraphrased: the software architecture of a system is the set of structures needed to reason about the system, and those structures consist of software elements, the relations among them and the properties of both.

Each part of that sentence does work. **Structures**, plural: a system has many structures at once (its code modules, its running processes, its deployment onto machines) and no single one is "the" architecture. **Needed to reason about the system**: the test of whether something is architectural is whether you need it to answer important questions, such as "Will this survive peak load?" or "Can two teams work on this independently?". **Elements** are the parts: modules, services, databases, queues. **Relations** are how they connect: *uses*, *calls*, *publishes to*, *is deployed on*. **Properties** are what makes elements and relations behave as they do: a service is stateless, a call is synchronous with a two-second timeout, a database holds personal data and must stay in the EU.

The phrase about reasoning also explains what is *not* architecture. The name of a local variable, the internal algorithm of a sorting routine or the layout of a settings screen usually do not help you reason about the system as a whole, so they belong to detailed design rather than to architecture. Section 1.2 examines where that line falls.

Elements, relations, properties and environment



Elements (boxes) connected by relations (arrows), with properties written beside them, inside a system boundary that sits in a wider environment.

Practitioner definitions: shared understanding, important stuff and decisions

Practitioners have offered shorter definitions that emphasise people and decisions rather than structures. Martin Fowler has popularised two from Ralph Johnson (one of the four authors of the 1994 *Design Patterns* book). The first is that architecture is the shared understanding that the expert developers have of how the system is designed. The second, from the same discussion, is deliberately open-ended: “Architecture is about the important stuff. Whatever that is.” Johnson also suggested that architecture is the set of decisions you wish you could get right early in a project, not because you are more likely to get them right early, but because they are so costly to get wrong.

Grady Booch, one of the creators of the UML, framed it in economic terms: architecture consists of the significant design decisions that shape a system, where significance is measured by the cost of change. That turns an abstract idea into a usable test. Choosing the colour of a button is cheap to reverse. Choosing between a relational database and a document store, or between one deployable unit and forty, is expensive to reverse once data, code and teams have grown around it. Booch also observed that all architecture is design, but not all design is architecture.

These definitions add something the structural ones leave implicit. Architecture lives partly in people's heads; a system whose senior engineers share a clear picture of how it works has a more coherent architecture than one where everyone holds a different picture, even if neither has a diagram. And architecture is shaped by decisions, many of them involving a trade off: choosing one option means giving up the advantages of another. Section 1.6 looks at decisions and trade-offs in detail.

The definitions compared

Source	Core idea (paraphrased)	What it emphasises	Most useful for
ISO/IEC/IEEE 42010:2022	Fundamental concepts or properties of an entity in its environment, plus governing principles for its realisation and evolution	Environment, principles, evolution; architecture distinct from its description	Contracts, standards work, describing architectures consistently across an organisation
Bass, Clements and Kazman (2021)	The set of structures needed to reason about the system: elements, relations and properties of both	Multiple structures; reasoning about qualities	Analysis and design: asking whether a structure will meet its goals
Ralph Johnson, via Martin Fowler	The shared understanding of expert developers; the important stuff, whatever that is	People, communication, judgement about importance	Team practice: keeping a common mental model alive
Grady Booch	The significant design decisions that shape a system, with significance measured by cost of change	Decisions and their economics	Deciding where to spend analysis effort

What the definitions share

Read side by side, the definitions converge on seven ideas, and together these make a working definition for the rest of this course.

Seven shared ideas

1. **Structures**
   A system has several structures at once, each showing it from a different angle. No single diagram is the whole architecture.

2. **Elements**
   The parts that make up those structures: modules, components, services, data stores, queues, servers.

3. **Relations**
   How the elements connect and depend on one another: calls, messages, shared data, deployment, ownership.

4. **Properties**
   The externally visible characteristics of elements and relations, such as being stateless, encrypted, asynchronous or replicated, which together determine the system's quality attributes.

5. **Decisions**
   The choices that produced the structures, with the reasons and rejected alternatives behind them. Without the reasons, people later undo good decisions by accident.

6. **Significance**
   Only some things are architectural: those that are costly to change, affect many parts, or determine a key quality. Architecture is selective by definition.

7. **Environment**
   The system sits among users, other systems, an organisation, laws and budgets. These shape the architecture and are shaped by it.

Three kinds of structure

The SEI's work on documentation, published as *Documenting Software Architectures: Views and Beyond* (Clements and colleagues, 2nd edition, 2010), groups structures into three kinds, and *Software Architecture in Practice* uses the same grouping. Knowing the three kinds protects you from the commonest mistake in architecture discussions: talking about one kind of structure while the other person is thinking of another.

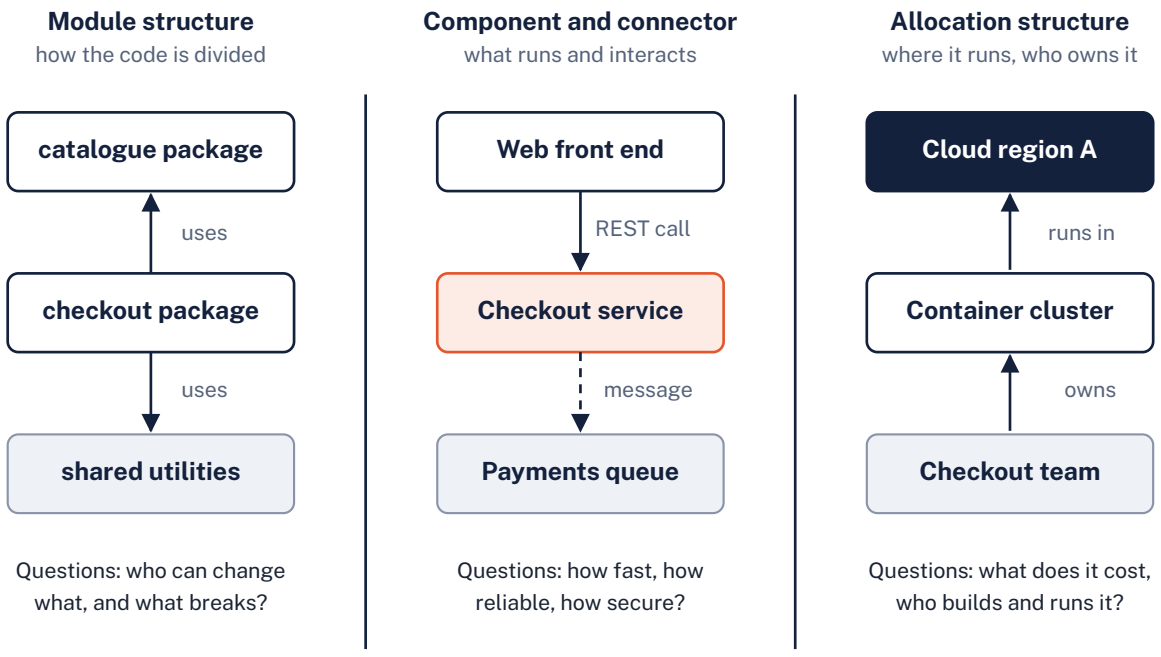
Module structures show how the code is divided into units of implementation, such as packages, libraries and namespaces, and how those modules depend on each other. They exist at development time. They answer questions such as: which team can change what without breaking others, and which code can be reused?

Component-and-connector structures show what exists when the system runs: processes, services, clients, databases, and the connectors between them, such as HTTP calls, message queues and event streams. They answer runtime questions: how fast is a request, what happens when one part fails, where is data held?

Allocation structures map software onto things outside the software: onto hardware and cloud infrastructure (deployment), onto teams (work assignment), and onto files and repositories (implementation). They answer questions about cost, operations and organisation: where does it run, who builds it, who is woken up at night when it fails?

One element can appear in all three. A checkout feature is a *package* in the module structure, a running *service* in the component-and-connector structure, and a set of *containers in a cloud region owned by the checkout team* in the allocation structures. The mapping between structures is not always one to one: a single module may be deployed as several services, or several modules may run in one process. That gap is exactly where many surprises hide.

One system, three kinds of structure



Module, component-and-connector and allocation structures show the same online shop from three angles and answer different questions.

Every system has an architecture

Because architecture consists of structures and the properties they actually have, every software system has an architecture, whether or not anyone designed it on purpose and whether or not it is written down. A spreadsheet macro that grew into a business-critical tool

has an architecture. So does a startup's codebase written in a hurry by three people. The question is never whether a system has an architecture, but whether that architecture is fit for its purpose, understood by the people who work on it, and deliberately shaped.

This leads to a useful distinction. The *intended* (or prescriptive) architecture is what the team believes or has decided the structure should be: the diagram on the wiki, the agreed rules. The *actual* (or descriptive) architecture is what the code and infrastructure really do. The two drift apart over time unless someone keeps them aligned, and that gap is one of the main sources of technical debt, covered in Section 1.7. An undocumented architecture is not absent; it is merely harder to see, reason about and change safely.

WHERE THIS CAME FROM

Where the definitions came from

Software architecture emerged as a named discipline in the early 1990s. Dewayne Perry and Alexander Wolf's 1992 paper *Foundations for the Study of Software Architecture* proposed that an architecture consists of elements, their form and the rationale behind them, and David Garlan and Mary Shaw's work, culminating in their 1996 book *Software Architecture: Perspectives on an Emerging Discipline*, catalogued recurring architectural styles. The first edition of *Software Architecture in Practice* appeared in 1998, and IEEE 1471, the ancestor of ISO/IEC/IEEE 42010, was published in 2000.

The ideas are older still. Edsger Dijkstra's layered THE operating system (1968) and David Parnas's 1972 paper on dividing systems into modules by hiding design decisions are both architectural thinking before the word was used.

WORKED EXAMPLE

Worked example: applying the definitions to a food-delivery app

A food-delivery company asks a new architect to "tell us what our architecture is". There is one out-of-date diagram. Using the shared ideas as a checklist, the architect produces a first description in a week.

Elements	Mobile app, web app for restaurants, order service, courier-matching service, payment module, a shared PostgreSQL database, a message broker, an external card payment provider.
Relations	Apps call the order service over HTTPS; the order service writes to the shared database and publishes "order placed" events; courier matching consumes those events; the payment module calls the provider's API.
Properties	The order service is stateless and runs as three copies; courier matching keeps state in memory and runs as a single copy; the database holds personal data and card tokens.
Environment	Two development teams, a food-safety regulator, data protection law, a fixed cloud budget, restaurants with unreliable tablets.
Significant decisions found	One shared database for all services; a single copy of courier matching; events used for courier dispatch but direct calls for payments.
Intended vs actual	The old diagram shows separate databases per service. In reality the courier service reads the order tables directly, so the two services cannot be changed independently.

The exercise surfaces two risks no one had stated: the single copy of courier matching is a single point of failure, and the shared database couples two teams together.

Neither risk was visible in the code of any one service. Both are properties of structures and relations, which is why they count as architecture.

The description took a week because nobody had recorded the decisions or their reasons. Recording them is the subject of Chapter 4.

WATCH OUT

Common misconceptions

- **"Architecture is the diagram."** A diagram is one partial description of one structure. The architecture is the set of properties the system actually has.
- **"Architecture is the technology stack."** Technology choices can be architectural, but how responsibilities, data and dependencies are divided usually matters more than which framework is used.
- **"Architecture is only for large systems."** Small systems have architectures too; they simply have fewer significant decisions, so less effort is needed to get them right.
- **"Architecture is decided once, at the start."** All the definitions include evolution. Architecture is shaped throughout a system's life, as Section 1.8 explains.

How professionals use the definitions

In practice, experienced architects rarely recite a definition, but they use each one as a lens. The ISO view reminds them to look outward at the environment and to separate what the system *is* from what the documents *say*. The SEI view gives them a vocabulary of structures, elements, relations and properties for analysis, and a test ("do we need this to reason about the system?"). Johnson's view reminds them that a shared mental model across the team is itself an asset to be maintained. Booch's view tells them where to spend their limited time: on the decisions that would be expensive to reverse.

A practical rule of thumb follows. When you are unsure whether something is architectural, ask three questions. Would it be expensive or risky to change later? Does it affect several parts of the system or several teams? Does it determine whether the system meets an important quality goal, such as speed, availability or security? If the answer to any of these is yes, treat it as architecture: decide it deliberately, record the reasons, and make sure the people affected understand it.

CHECK YOUR UNDERSTANDING

▼ **What is the difference between an architecture and an architecture description?**

The architecture is the set of fundamental properties and structures the system actually has. The description is a work product, such as diagrams and documents, that expresses it, and it can be incomplete or out of date.

▼ **Which kind of structure would you examine to find out which team is woken up when a service fails?**

An allocation structure, specifically the work-assignment or ownership mapping of software onto teams.

▼ **Using Booch's test, is the choice of logging library architectural?**

Usually not, if it is hidden behind a small interface and cheap to swap. It becomes architectural if every module calls it directly, making it expensive to change.

▼ **Why does the ISO definition mention the environment?**

Because an architecture can only be judged against the users, systems, organisation, laws and budgets around it; the same structure can be good in one environment and poor in another.

▼ **Try it: A food-delivery manager says: "We have no architecture, just code and one old diagram." Respond, then sort these into structure types: the orders package imports payments; orders publish to a broker; dispatch runs on three servers run by Team B.**

Every system has an architecture, designed or not; what they lack is an up-to-date description of it. The package import is a module structure, because it concerns code. Publishing to a broker is a component-and-connector structure, because it concerns the running system. Dispatch on servers run by Team B is an allocation structure, mapping software to infrastructure and people. No single diagram shows all three, so a good description needs several.

SOURCES

- ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description (ISO): https://www.iso.org/standard/74393.html
- Software Architecture in Practice, Fourth Edition (SEI, Carnegie Mellon University): https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- Software Architecture Guide (Martin Fowler): https://martinfowler.com/architecture/
- Documenting Software Architectures: Views and Beyond, Second Edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/

SECTION 1.2 OF 1.8 · BEGINNER · 20 MIN READ · LAST REVIEWED NOVEMBER 2026

Architecture, design and code: significance, scope and levels

THE IDEA IN BRIEF

When a house is planned, some choices fix everything else: how many floors, where the stairs go, where the load-bearing walls stand. Others can be changed on a weekend: paint, shelves, light fittings. Both are design, but only the first kind is architecture, because changing it later means knocking the house about. Software is similar. Choosing how a system is split into parts, or how money amounts are stored everywhere, is architecture because it is costly to undo. Choosing how one function is written inside one part is design.

WHAT YOU NEED TO KNOW

- Architecture and design form a continuum: all architecture is design, but only the significant design decisions are architecture.
- A decision is architecturally significant when it is hard or expensive to reverse, has wide scope, determines a key quality attribute, constrains others or involves important trade-offs.
- Level of abstraction alone is a poor test: low-level conventions such as identifier or money formats can be architectural, and some high-level boxes are trivial.
- Strategic decisions set long-term direction for many people; tactical decisions solve bounded, immediate problems, but repeated tactical choices can become architectural.
- Enterprise architecture covers a whole organisation (TOGAF is the best-known framework), solution architecture one business problem across systems, and software architecture one system; data, security and infrastructure architecture cut across all levels.
- Decisions should be made at the lowest level with enough context and accountability, with wider consultation for decisions that affect many teams.

A continuum, not a wall

If architecture is the set of significant decisions and structures (Section 1.1), then design is everything else: the many smaller decisions about how each part works inside. The difficulty is that there is no clean line between them. Choosing to split a food-delivery platform into an ordering system and a courier system is clearly architectural. Naming a variable is clearly not. Between those extremes lies a long middle ground of decisions (an API's error format, a caching rule, how a module is divided into packages) whose status depends on context.

Grady Booch put it neatly: all architecture is design, but not all design is architecture. Architecture and detailed design are points on one continuum of design decisions, ordered by how significant they are. That is why the most useful question is not "is this architecture or design?" in the abstract, but "how significant is this decision here, and how much care does it deserve?".

A second, popular way to draw the line is by level of abstraction: architecture is the high-level view ("boxes and arrows"), design is the low-level view ("classes and methods"). That is a reasonable first approximation but it fails in both directions. A low-level detail can be architectural: the format of an identifier shared by twenty services, or the choice of a time zone convention for stored dates, may cost months to change. And a high-level box can be trivial: whether an internal admin tool is a separate service may not matter at all.

What makes a decision architecturally significant

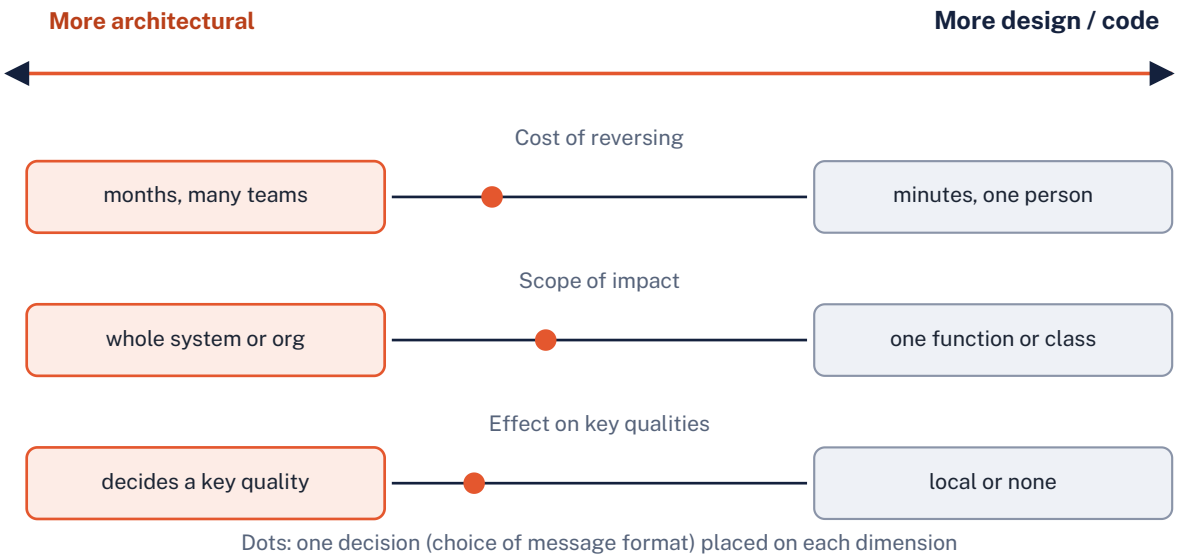
A decision is architecturally significant when it has one or more of the following characteristics. These are not formal rules from a single standard; they summarise what the SEI, Martin Fowler, Mark Richards and Neal Ford and others say in different words.

Signs that a decision is architecturally significant

Characteristic	Question to ask	Example
Hard or expensive to reverse	If this turns out wrong in a year, what will it cost to change?	Choosing an event-sourced data model rather than storing current state
Wide scope	How many components, teams or systems does it touch?	Defining the format of customer identifiers used across all services
Determines a key quality	Does it decide whether we meet a performance, availability, security or cost goal?	Putting a cache in front of the product catalogue to meet a 200 ms response target
Sets a constraint for others	Does it limit what other teams may choose?	Mandating that all services publish events to one shared broker
Involves significant trade-offs	Are we giving up something important to get something else?	Accepting eventual consistency between orders and stock to keep checkout available
High novelty or risk	Has anyone here done this before? What if the technology fails us?	Adopting a database the organisation has never operated in production

Mark Richards and Neal Ford, in *Fundamentals of Software Architecture*, describe the architecture/design spectrum in similar terms: a decision sits further towards architecture the more strategic it is (long-term, many people involved), the more effort it would take to change, and the more significant its trade-offs. Martin Fowler offers a compact version: architecture is the stuff that is hard to change, which also means that one of an architect's goals is to make more things easy to change. Making a decision cheap to reverse, for example by putting it behind an interface, effectively moves it from architecture towards design. This idea, reversibility, is explored further in Section 1.6.

Placing a decision on the continuum



Dots: one decision (choice of message format) placed on each dimension

Three dimensions place a decision between architecture and design; a decision can score differently on each.

Strategic and tactical decisions

A related distinction is between strategic and tactical decisions. **Strategic** decisions set direction for a long time and for many people: adopting a cloud provider, choosing between a monolith and distributed services, deciding that customer data will be mastered in one system. They usually need input from several stakeholders and are worth writing down with their reasons. **Tactical** decisions solve an immediate problem within a bounded area: which library to use for parsing dates in one service, how to structure a particular database query, how to name the endpoints of an internal API.

Strategic does not mean slow. Many strategic decisions can and should be made quickly once the options are clear, and some are reversible if they are structured well. Tactical does not mean unimportant: a poor tactical decision repeated in fifty places (say, every service writing

its own retry logic with no back-off) can add up to an architectural problem. Architects watch for tactical decisions that are about to become strategic through repetition, and turn them into a shared principle or platform capability.

Architectural or design? Example decisions with reasoning

Decision (for an online shop)	Verdict	Reasoning
Split the system into separately deployable services for catalogue, checkout and delivery	Architectural	Affects every team, deployment, data ownership and failure behaviour; very costly to reverse.
Use PostgreSQL rather than a document database as the main store for orders	Architectural	Shapes the data model, transactions and operations skills; migration later is expensive and risky.
Use a specific JSON library inside the checkout service	Design	Local to one service and replaceable in a day if hidden behind the service's own code.
Require every service to emit structured logs with a shared correlation ID	Architectural	A cross-cutting concern: it only works if everyone follows it, and it determines how quickly incidents can be diagnosed.
Name the method that calculates delivery fees	Design (code)	Purely local; renaming is automated by tools.
Choose the format of order numbers shown to customers and shared with the warehouse partner	Architectural	Low-level, but it crosses system and company boundaries and is printed on parcels; changing it needs coordination with an external party.
Use the Strategy pattern to support several delivery-fee rules	Design	Shapes one component's internals; other components are unaffected.
Cache product prices for 60 seconds	Depends	Design if prices change rarely and staleness is harmless; architectural if a stale price creates a legal or financial problem, because it then decides a key quality (correctness).

Levels of architecture

The word architecture is also used at several levels of scope, and job titles follow these levels. Confusing them causes many misunderstandings: an enterprise architect and a software architect may both be "the architect", yet their work barely overlaps.

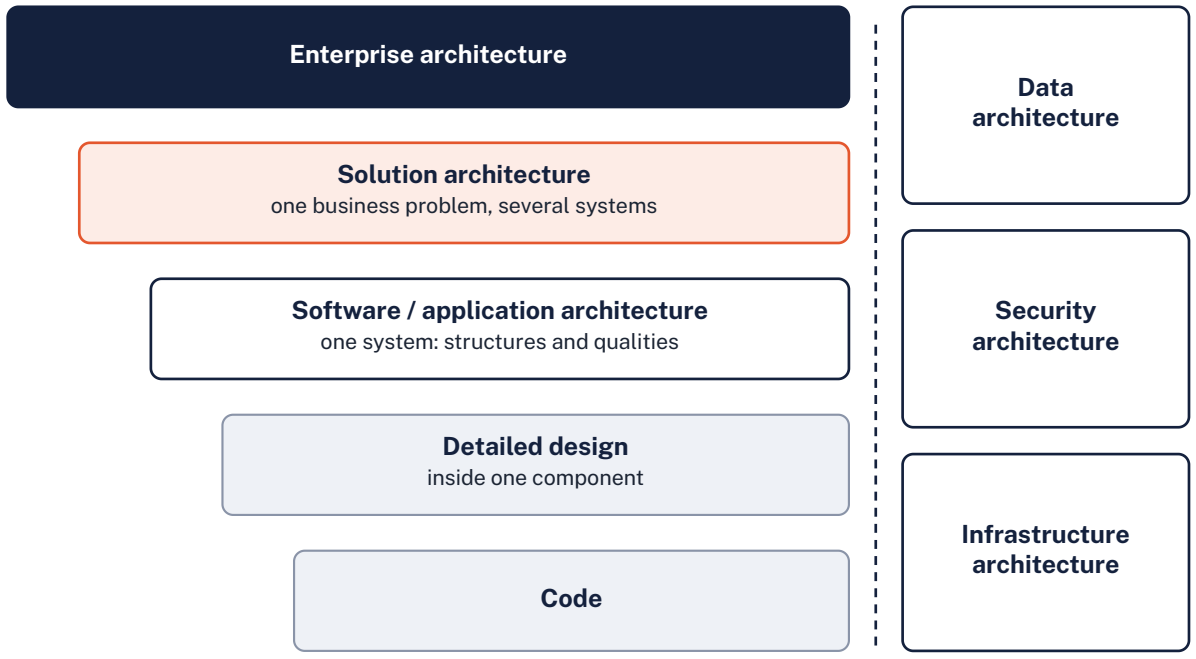
Enterprise architecture looks at a whole organisation: its business capabilities (what it must be able to do, such as "assess claims" or "onboard customers"), the information it holds, its portfolio of applications and its technology platforms, and how all of these should change to support strategy. Typical outputs are capability maps, target-state roadmaps, technology standards and decisions about which systems to retire. The most widely used framework here is TOGAF, published by The Open Group, whose current version is the TOGAF Standard, 10th Edition (2022). It provides a method (the Architecture Development Method) and a body of content for planning and governing enterprise change. Organisations use it to varying degrees; many adopt parts of it and some find it heavyweight, and this course describes it only so you can recognise it.

Solution architecture addresses one business problem or programme, typically spanning several applications, teams and vendors: for example, a hospital introducing online appointment booking that must connect a new web front end, the existing patient administration system, an SMS provider and the identity service. The solution architect makes the pieces fit together, often under commercial and contractual constraints.

Software or application architecture, the main subject of this course, concerns the structures and qualities of one system: its decomposition, its runtime interactions, its deployment, its data and its evolution.

Three specialisms cut across all these levels. **Data architecture** covers how data is modelled, owned, stored, moved, governed and used for analytics (Chapter 7). **Security architecture** covers identity, access, protection of data, threat models and how security controls are placed (Chapter 8). **Infrastructure architecture** (often now cloud or platform architecture) covers compute, networks, storage, environments and deployment platforms (Chapter 10).

Levels and cross-cutting specialisms



Each level constrains the one below; specialisms cut across every level

Scope narrows from enterprise to code; data, security and infrastructure architecture apply at every level.

Who makes which decisions

In a healthy organisation, decisions are made at the lowest level that has enough context and is accountable for the consequences. Architecture is not the sole property of people with "architect" in their title. A reasonable default division looks like this, although every organisation tunes it.

Kind of decision	Usually made by	Usually consulted
Technology standards and platforms for the organisation (e.g. approved cloud provider, identity platform)	Enterprise architects, CTO, architecture board	Solution and software architects, security, finance
How several systems and vendors combine to solve a business problem	Solution architect	Product owners, enterprise architects, vendor architects
The major structure of one system: decomposition, communication style, data ownership	Software architect or the team's technical lead, with the team	Operations, security, other teams affected
Design inside a component: classes, patterns, algorithms	The development team	Architect only if a principle or quality is at stake
Code-level choices: naming, local refactoring	Individual developers, via code review	Nobody else, normally

Two warning signs show that the division is wrong. If architects decide internal class structures, they become a bottleneck and the team stops thinking. If teams independently make decisions with system-wide effects (each choosing its own message broker, say), the organisation pays for the inconsistency later. Writing down which decisions need wider consultation, and making it easy to record them, is covered under decisions and trade-offs and, in depth, in Chapter 4's treatment of Architecture Decision Records.

WORKED EXAMPLE

Worked example: classifying decisions for a ride-hailing app

A ride-hailing start-up lists five pending decisions. The technical lead uses the three signs (cost to reverse, scope, effect on key qualities) to decide which need a written record and wider review.

1. Store trip locations in a time-series database	Reverse: high (data migration). Scope: trips, billing, analytics. Quality: performance of live tracking. Verdict: architectural; write it up and review with data and operations staff.
2. Use a particular HTTP client library in the pricing service	Reverse: low. Scope: one service. Quality: none significant. Verdict: design; the pricing team decides.
3. Calculate fares on the server, never on the phone	Reverse: medium. Scope: apps and back end. Quality: security and integrity of pricing. Verdict: architectural; it is a governing principle.
4. Represent money as integer minor units (pence, cents) everywhere	Reverse: very high once data and partner APIs use it. Scope: every service. Quality: correctness. Verdict: architectural despite being low-level.
5. Use dark mode colours in the driver app	Reverse: trivial. Scope: one screen set. Verdict: design.

Two of the three architectural decisions (3 and 4) are not about boxes and arrows at all; they are principles and conventions. Level of abstraction alone would have misclassified them.

Decision 4 is the kind most often missed: it looks like a coding detail but becomes very expensive to change once other systems depend on it.

Questions people ask

Is choosing a programming language an architectural decision?
Often, yes. It affects hiring, available libraries, performance characteristics, and how easily code can be shared between components. In a system of many independent services, the language of one small service may be a local choice; the default language for the whole organisation is architectural.

Is a database schema architecture or design?
It depends on who uses it. A table used only inside one service, behind its own API, is design. A schema read directly by several services, reports and partner exports is architectural, because every change must be coordinated across all of them. Many teams make schemas private to one service precisely to move them back into design.

Does agile development mean there is no architecture?
No. Every system has an architecture; agile approaches change when and how it is decided, favouring small, reversible steps and deferring irreversible decisions until enough is known. Section 1.8 covers this.

Who decides whether a decision is significant?
Usually the team and its architect or technical lead together, using agreed criteria such as the signs in this section. Writing those criteria down avoids two failure modes: debating every choice, and discovering after the fact that a team made a system-wide decision alone.

How professionals apply the distinction

Experienced teams turn the continuum into a lightweight routine. They agree a short list of the kinds of decision that need a written record and wider consultation, for example anything that changes a public API, introduces a new data store or technology, crosses team boundaries, or affects a stated quality goal. Everything else is decided by the people doing the work, through normal code review. They revisit the list when something slips through or when reviews feel like bureaucracy.

They also look actively for ways to shrink the set of architectural decisions, because every decision that becomes cheap to reverse is one less that needs to be right first time. Hiding a vendor behind an interface, making a database private to one service, using feature flags for risky behaviour and deploying in small increments all move decisions from the architectural end of the continuum towards the design end.

GOING DEEPER

The intension/locality idea

Amnon Eden and Rick Kazman proposed a more formal distinction in 2003, based on two criteria. *Intension* asks whether a statement is general (it describes a whole class of possible implementations) or specific (it describes one implementation); *locality* asks whether it can be checked in one part of the system or constrains the whole. Architectural statements are general and *non-local*: they constrain the whole system ("all components communicate through the message bus"), so they can be violated anywhere. Design statements are general but local ("use the Observer pattern in this module"). Implementation statements are specific and local ("this class has these methods"). The test is useful when writing rules: a statement that must hold everywhere is a candidate for an architectural principle and for an automated check, a fitness function.

WATCH OUT

Mistakes to avoid

- Treating everything as architecture, which produces review boards that slow every change and teams that stop taking ownership.
- Treating nothing as architecture, which lets system-wide properties emerge by accident from hundreds of local choices.
- Assuming a decision is design because it is low-level. Data formats, identifiers, time and money representations and API contracts are frequently architectural.
- Assuming a decision is architectural because it involves a big component. If it is easily reversed and local in effect, it does not need ceremony.

CHECK YOUR UNDERSTANDING

▼ **Why can a choice about the format of an identifier be architectural?**

Because once many services, databases and external partners rely on it, changing it requires coordinated work across all of them, so its cost of change and scope are high.

▼ **What is the difference between enterprise and solution architecture?**

Enterprise architecture covers the whole organisation's capabilities, information, applications and technology; solution architecture addresses one business problem that usually spans several systems and teams.

▼ **How can an architect move a decision from architecture towards design?**

By making it cheaper to reverse, for example hiding a technology behind an interface so it can be replaced locally.

▼ **Try it: A ticketing team lists three decisions: name a helper function; store all money as integer pence across every service; give each service its own database or share one. Which are architectural? Use reversal cost, scope and effect on qualities.**

The helper name is local design: cheap to reverse, tiny scope, no effect on qualities. The money format looks low-level but is architectural, because it spreads through every service and stored record, is costly to change later and determines correctness. Shared or separate databases is plainly architectural: very hard to reverse, wide scope, and it shapes deployability and consistency. Those two deserve a written record and wider review; the first does not.

SOURCES

- Who Needs an Architect? Martin Fowler, IEEE Software, 2003 (PDF): https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- Software Architecture Guide (Martin Fowler): https://martinfowler.com/architecture/
- The TOGAF Standard (The Open Group): https://www.opengroup.org/togaf
- iSAQB CPSA-Foundation curriculum: https://public.isaqb.org/curriculum-foundation/curriculum-foundation-en.html

SECTION 1.3 OF 1.8 · BEGINNER · 19 MIN READ · LAST REVIEWED NOVEMBER 2026

Why architecture matters: cost of change, quality, risk and real failures

THE IDEA IN BRIEF

Imagine two kitchens that cook the same menu. In one, every station is laid out so cooks rarely cross paths; in the other, the only fridge is behind the dishwasher. Both serve the same dishes on a quiet night. On a busy Saturday, or when the menu changes, the second kitchen jams. Software is similar: the features may look identical, but the arrangement of the parts decides how fast, reliable and cheap the system is, and how painful it is to change. Some of the costliest software failures in history were problems of arrangement, not of individual lines of code.

WHAT YOU NEED TO KNOW

- Architecture largely determines quality attributes such as performance, availability, security and cost, because these emerge from how parts are arranged and interact.
- The cost of changing a decision grows as code, data, integrations and teams are built on top of it, so early structural choices shape the cost and speed of all later work.
- Architecture governs risk, how work can be split between teams (Conway's law), communication and the ability to estimate change.
- Knight Capital lost more than USD 460 million on 1 August 2012 after a manual deployment left one of eight servers running old code that a reused flag reactivated.
- GAO found that HealthCare.gov launched in October 2013 without verification that it met performance requirements, after unsettled requirements, weak oversight and incomplete testing.
- Prime Video's 2023 move of one monitoring service from distributed serverless steps to a single process reportedly cut its infrastructure cost by over 90%, showing that architecture is about fit, not fashion.

The decisions that are expensive to get wrong

Section 1.1 defined architecture as the set of structures and significant decisions that shape a system, with significance measured by cost of change. This section asks the practical follow-up: so what? Why should a team, a product owner or a finance director care about architecture, rather than simply about features?

The short answer is that the architecture decides, more than any other technical factor, what the system will be *like*: how fast, how reliable, how secure, how cheap to run, and above all how easy to change. Features are what a system does. Architecture determines how well it does them and what it will cost to do something different next year. Many large software failures and many expensive successes trace back to a handful of structural decisions.

Architecture determines quality attributes

A quality attribute is a measurable property of how well a system does its job: performance, availability, security, modifiability, scalability, usability, cost-efficiency and others. Chapter 2 makes these precise. The key point here is that most quality attributes are *emergent*: they arise from how the parts are arranged and how they interact, not from any one piece of code.

Consider availability. If an online shop's checkout must make five synchronous calls in sequence to services that are each available 99.9% of the time, the chain as a whole can be no more available than roughly 99.5%, however well each service is written, because the probabilities multiply. Adding a queue so that some of those steps happen later, or caching results so that some calls are not needed, changes the arithmetic. Neither change is visible inside any single function; both are architectural. Performance, security and cost behave in the same way. A secure login page does not make a system secure if an internal service with no authentication is reachable from the internet.

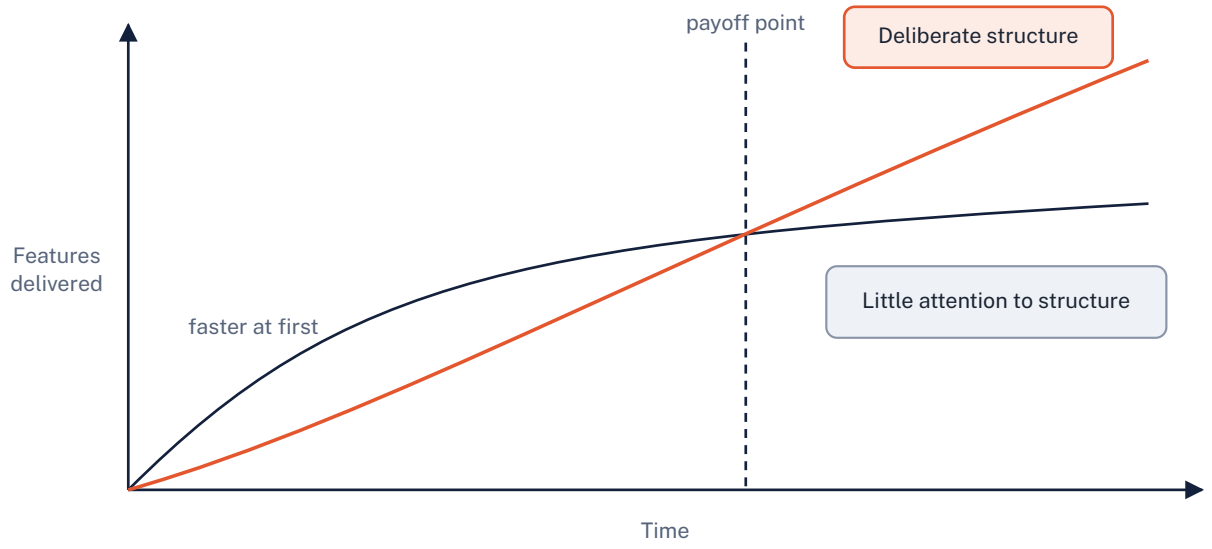
Functionality, by contrast, is usually largely independent of structure. You can implement the same features of a banking app as one deployable unit or as fifty services; users see the same buttons. What changes is everything else: how quickly a fix can be released, what happens when one part fails, how many engineers the system can absorb, and what the cloud bill looks like.

Cost and speed of change

Most of a successful system's cost is incurred after its first release, as it is changed to meet new needs. The structure therefore matters most because it governs how expensive each future change will be. In a well-structured system, a change in tax rules touches one module. In a tangled one, it touches twenty, each needing testing and coordination.

Martin Fowler's *design stamina hypothesis* (2007) captures the effect. A team that pays little attention to internal structure often delivers faster at first. Over time, the accumulated tangle slows every change, and a team that invested in structure overtakes it. Fowler calls this a hypothesis because it is hard to measure rigorously, and he suggests the crossover usually comes within weeks rather than months. The slowdown is the interest on technical debt, which Section 1.7 covers.

The design stamina hypothesis



A conceptual sketch after Martin Fowler: less attention to structure is faster at first, but deliberate structure sustains the pace of delivery.

WORKED EXAMPLE

Worked example: the cost of one decision over time

An online shop stores money amounts as floating-point numbers. Floating-point arithmetic cannot represent many decimal values exactly (0.1 + 0.2 is not exactly 0.3), so totals occasionally differ from what accountants expect by a penny. The figures below are illustrative assumptions, not industry data, chosen to show how the cost of change grows.

Month 1: decide differently	Two developers agree to store integer minor units (pence) instead. Effort: about half a day. Cost at an assumed £600 per person-day: about £300.
Year 3: what has been built on the decision	35 services handle money; 120 database columns store it; 200 million historical order rows; 4 partner integrations (payment provider, warehouse, accounting, marketplace) exchange it.
Year 3: code changes	35 services at about 3 person-days each to change, test and release = 105 person-days.
Year 3: data migration	Convert columns, migrate history, reconcile totals: about 20 person-days, plus a planned maintenance window.
Year 3: partner coordination	4 partners at about 10 person-days each, including their testing and contract changes = 40 person-days.
Year 3: end-to-end testing and reconciliation with finance	About 30 person-days.
Year 3: total	About 195 person-days, or roughly £117,000, plus the risk of errors during migration and months of elapsed time. That is several hundred times the month-1 cost.

The decision itself never changed in size. What grew was everything built on top of it: code, data, partners and habits.

This is why Ralph Johnson described architecture as the decisions you wish you could get right early, and why architects look for low-level conventions with wide scope.

An alternative strategy is to make such decisions cheaper to reverse, for example by putting all money handling behind one shared type or library from the start, so that a later change is local.

Risk, teams, communication and estimation

Risk. Architecture is where most technical risks are either contained or amplified. A design with a single point of failure, an untested integration between independently built parts, or an unproven technology at its core carries risk that no amount of careful coding removes. Much architectural work is risk management: identifying what could go badly wrong and shaping the structure so that it either cannot happen or does limited damage when it does.

Splitting work between teams. Melvin Conway observed in 1968 that organisations which design systems tend to produce designs that mirror their own communication structures. This is now called Conway's law, and it works in both directions. A system whose parts have clear boundaries and narrow interfaces lets several teams work in parallel with little coordination. A system where everything depends on everything forces every team into every meeting. Architecture and organisation design are therefore intertwined, as Chapter 3's section on Conway's law explains.

Communication. An architecture gives people a shared vocabulary and a shared map. New engineers learn the system faster, product managers understand why some requests are cheap and others are expensive, and operations staff know what depends on what during an incident.

Estimation and planning. Decomposition into known parts makes estimation possible: a change can be traced to the components it touches, and each can be estimated by the team that knows it. Without a clear structure, estimates are guesses, because nobody can say with confidence how far a change will ripple.

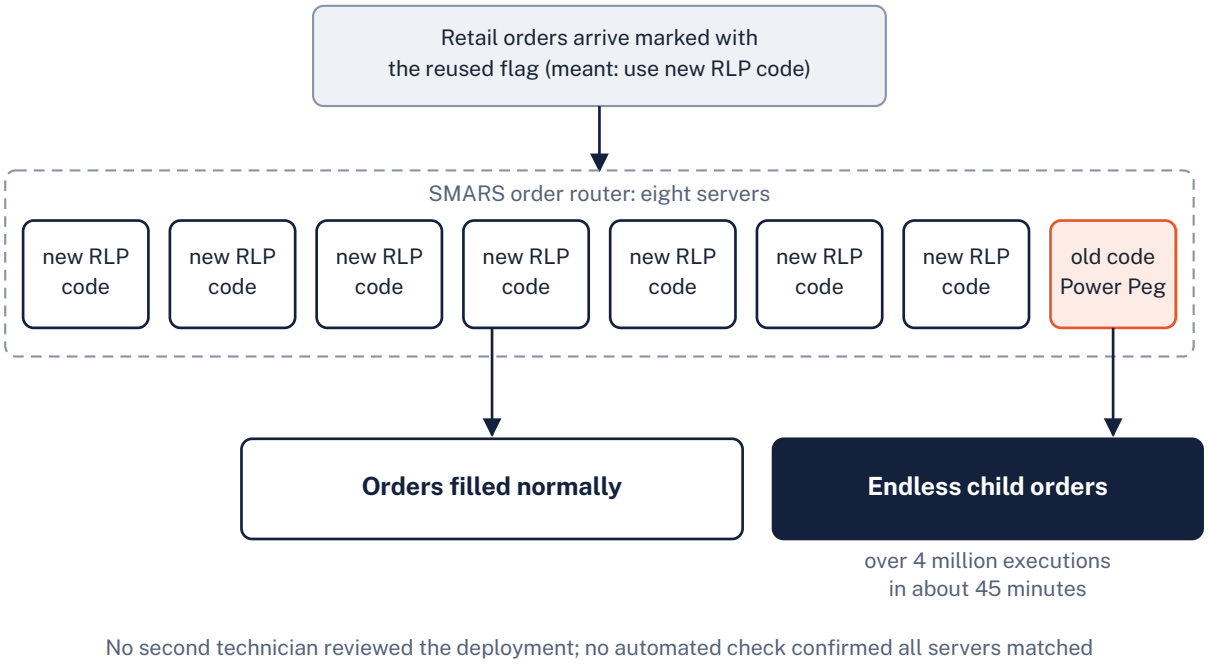
Case study: Knight Capital, 1 August 2012

Knight Capital was one of the largest market makers in US equities. According to the US Securities and Exchange Commission's order of 16 October 2013, in the first 45 minutes of trading on 1 August 2012 Knight's automated order router (called SMARS) sent millions of unintended orders to the market while processing 212 small retail orders. It received more than 4 million executions in 154 stocks, over 397 million shares, and was left with large unwanted positions. Knight lost more than USD 460 million, needed emergency outside investment within days, and the SEC later fined it USD 12 million for violating market-access rules.

The SEC order describes the mechanism. Years earlier, Knight had stopped using an old function called Power Peg, but its code remained in the router and could still be triggered: it was dead code that was not dead. In 2012 Knight wrote new code for a new NYSE programme, the Retail Liquidity Program, and that new code reused a flag that had formerly switched on Power Peg. The new code was deployed manually to the router's eight servers. A technician did not copy it to one of them, no second technician reviewed the deployment, and Knight had no

written procedure requiring such a review. When the flag was switched on, seven servers ran the new code; the eighth interpreted the flag as "run Power Peg" and sent orders without recognising that they had already been filled. Automated emails warning of a problem were sent before the market opened but were not acted on.

How one stale server caused the Knight Capital loss



One reused flag, one stale server and no deployment check: the failure lay in structure and process, not in a single line of code.

It would be unfair to call this simply a coding bug, and equally unfair to call it purely an operations error. The architectural lessons are about structure and deployability: obsolete functionality left reachable in a production system; one flag given two meanings in different versions of the code; a deployment process with no automated verification that all servers ran the same version; and, as the SEC emphasised, inadequate controls to limit the damage once something went wrong. Each is a decision about how the system is built and released, not about a single line of code.

Case study: the HealthCare.gov launch, October 2013

HealthCare.gov, the US federal health insurance marketplace, opened for enrolment on 1 October 2013. Many people could not create accounts or complete applications, and the site suffered outages for weeks. The US Government Accountability Office (GAO) and the Department of Health and Human Services' Office of Inspector General (OIG) later investigated, and their findings are the best source for what went wrong.

The GAO's July 2014 report (GAO-14-694) found that the Centers for Medicare and Medicaid Services (CMS) issued contracts before key requirements were settled, that requirements kept changing, that oversight was weak, and that CMS launched the system without verifying that it met its performance requirements. Obligations on the main marketplace contract rose from USD 56 million to more than USD 209 million between 2011 and early 2014. A second GAO report in March 2015 (GAO-15-238) found that the systems had not been fully tested before launch, and that problems at launch included inadequate capacity planning, software coding errors and planned functions that had not been implemented. The OIG's 2016 case study identified the absence of clear leadership and poor coordination between policy and technical work as central causes. After intensive corrective work, the site was able to serve high consumer demand within about two months.

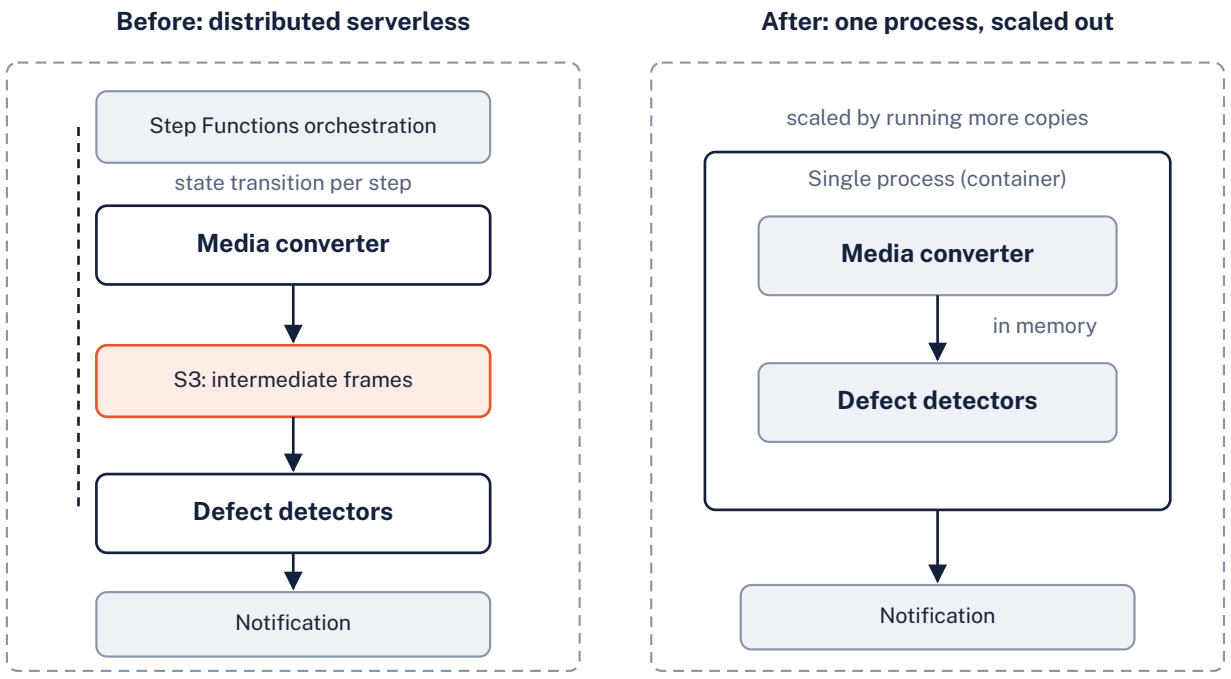
The HealthCare.gov story is mostly about programme management, but its architectural lesson is direct. The marketplace was built by several contractors as separate parts that had to work together, including a data hub connecting to other federal agencies, and nobody verified end to end, under realistic load, that the assembled whole met its capacity and performance goals. Quality attributes such as capacity are properties of the whole system, so they can only be assured by someone who owns the whole system's structure and tests it as a whole.

Case study: Prime Video and architecture as fit, not fashion

In March 2023, an engineer on Amazon Prime Video's Video Quality Analysis team published a post on the Prime Video Tech blog about a service that monitors video streams for quality defects. The first version had been built from distributed serverless components: AWS Step Functions orchestrated the steps and AWS Lambda functions did the work, with intermediate video frames passed between steps through Amazon S3 storage. The team reported that this design hit a hard scaling limit at around 5% of the expected load and would have been too expensive at full scale, largely because of the per-step orchestration charges and the many storage requests.

The team moved the media conversion and defect-detection steps into a single process running in containers on Amazon ECS, passing frames in memory instead of through storage, and scaled by running more copies of that process. The post reported that infrastructure costs fell by over 90% and that the service could scale further. The post was widely shared as "Amazon abandons microservices", which it did not say: the change concerned one service, and the author noted that microservices and serverless components do work at high scale, but that the choice between them and a monolith has to be made case by case.

Prime Video's monitoring service, before and after



The same steps, regrouped: moving data in memory inside one process removed per-step orchestration and storage costs.

The lesson is not that monoliths beat serverless. It is that architecture is about fit. The serverless design was quick to build and appropriate for an early, smaller workload. When the workload's defining characteristic (very high volumes of data passed between steps for every second of every stream) became dominant, a structure that moved that data across the network many times was the wrong fit, and a different structure fitted better. A good architect asks what the workload, the team and the budget need, not what is fashionable.

WATCH OUT

Reading case studies carefully

- Public accounts simplify. Use primary sources (regulators' orders, audit reports, the engineers' own posts) rather than retellings, which often exaggerate or add details.
- Most real failures have several causes: technical structure, process, organisation and incentives. Blaming one engineer or one line of code is almost always wrong.
- A decision that failed in one context may be right in another. Ask what the context was before drawing a general rule.

GOING DEEPER

Why architecture is hard to justify, and how to do it

The benefits of good architecture are mostly things that do *not* happen: the outage avoided, the rewrite not needed, the team not blocked. That makes them easy to discount. Effective architects translate structural choices into business terms: lead time for changes, cost per transaction, hours of downtime per year, the number of teams that can work in parallel, the size of the risk being carried. Chapter 11 returns to building business cases for architectural work.

CHECK YOUR UNDERSTANDING

▼ Why can a well-written service still be part of an unreliable system?

Because availability emerges from how services depend on each other. A chain of synchronous calls multiplies failure probabilities, so the whole can be less available than any part.

▼ What made the Knight Capital failure architectural rather than a simple bug?

Reachable obsolete code, a flag with two meanings across versions, and a manual deployment with no automated check that all servers matched: all structural and release decisions, not one faulty line.

▼ What is the main lesson of the Prime Video case?

That an architecture must fit its workload and context. A design suitable at one scale and data pattern can become a poor fit, and changing structure can transform cost.

▼ Why does the cost of changing a decision grow over time?

Because more code, stored data, integrations, partners and habits come to depend on it, so changing it means changing all of them.

▼ Try it: A new ticketing service stores prices as floating-point numbers. Changing to integer pence today takes two developers one day. The lead says: "We'll do it next year, it's only a rounding issue." What is your argument, and to whom?

Fix it now. Today nothing depends on the format; in a year it will be in stored bookings, reports, integrations and other teams' code, so the same change needs a data migration, coordinated releases and reconciliation. Early structural choices set the cost of all later work. Put it to the lead and the finance owner as a business point: one day now against weeks later, while totals that are out by a penny go on annoying accountants meanwhile.

SOURCES

- SEC press release 2013-222: SEC charges Knight Capital with violations of market access rule: https://www.sec.gov/news/press-release/2013-222
- GAO-14-694: Healthcare.gov: Ineffective Planning and Oversight Practices Underscore the Need for Improved Contract Management: https://www.gao.gov/products/gao-14-694
- GAO-15-238: Healthcare.gov: CMS Has Taken Steps to Address Problems, but Needs to Further Implement Systems Development Best Practices: https://www.gao.gov/products/GAO-15-238
- Marcin Kolny: Scaling up the Prime Video audio/video monitoring service and reducing costs by 90% (Prime Video Tech, 2023; the original post, which may no longer be online): https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90
- InfoQ: Prime Video moves from serverless to ECS, cutting costs (report on the original post, 2023): https://infoq.com/news/2023/05/prime-ec2-ecs-saves-costs
- Design Stamina Hypothesis (Martin Fowler): https://martinfowler.com/bliki/DesignStaminaHypothesis.html

SECTION 1.4 OF 1.8 · BEGINNER · 19 MIN READ · LAST REVIEWED NOVEMBER 2026

What an architect does: responsibilities, skills and kinds of architect

THE IDEA IN BRIEF

On a building project, the architect does not lay every brick, but decides the things that would be ruinous to get wrong, explains them to builders and clients, checks on site that the plans are followed, and adjusts when the ground turns out to be softer than expected. A software architect does the same for a software system: makes or guides the big decisions, explains them, keeps checking they are followed and still make sense, and needs to know a little about a lot of technologies and a good deal about people and the business.

WHAT YOU NEED TO KNOW

- Architects take responsibility for significant decisions being made well, understood and kept true as a system changes; architecture is a role before it is a job title.
- Richards and Ford describe eight expectations: make decisions, continually analyse, keep current, ensure compliance, have diverse exposure, know the business domain, have interpersonal skills and navigate politics.
- The iSAQB curriculum lists architects' tasks as clarifying requirements, designing structures and cross-cutting concepts, communicating, accompanying implementation and evaluating, done iteratively.
- Kinds of architect include enterprise, solution, software or application, data, security and cloud or infrastructure; staff and principal engineers often carry the same responsibilities.
- Architects value technical breadth (knowing many options and their trade-offs) over deep expertise in a few, as shown by the knowledge pyramid.
- Architects should usually stay hands-on without owning critical-path code: proofs of concept, fitness functions, bug fixes and code reviews.

What architects are for

If architecture is the set of significant decisions and structures of a system (Section 1.1), the architect is whoever takes responsibility for those decisions being made well, understood by others and kept true as the system changes. That sounds simple, but the role is one of the least consistently defined in software. In one company an architect is a senior engineer who still writes code daily; in another, a member of a central team who reviews designs and maintains standards; in a third, a person who mostly talks to business leaders and vendors.

Behind the variety there is a common core. Architects make or guide decisions that are costly to reverse, look after the qualities of the whole system rather than one part, and connect technical choices to business goals. The rest of this section describes what that involves, what skills it takes and how the role is organised.

Eight expectations of an architect

Mark Richards and Neal Ford, in *Fundamentals of Software Architecture* (O'Reilly, 1st edition 2020, 2nd edition 2025), describe eight core expectations of an architect, whatever the job title or company. They are paraphrased below, with an example of each from a food-delivery company.

The eight expectations (after Richards and Ford)

Expectation	What it means	Example
Make architecture decisions	Define decisions and principles that guide technology choices, usually by guiding the team rather than dictating specific products.	Deciding that restaurant integrations go through one adapter layer, rather than choosing each restaurant's library.
Continually analyse the architecture	Reassess whether the architecture still fits as technology and business change, and look for decay.	Noticing that the courier-matching service now takes 40% of all deploy failures and asking why.
Keep current with trends	Follow developments in technology and practice, because architectural decisions are long-lived.	Understanding new managed database offerings before the next capacity review.
Ensure compliance with decisions	Check that agreed decisions and principles are actually followed, ideally with automated checks.	Adding a build check that fails when the app calls the payment provider directly.
Have diverse exposure and experience	Know a wide range of technologies, frameworks and platforms well enough to compare them.	Comparing message brokers on ordering, durability and operating cost.
Have business domain knowledge	Understand the business well enough to talk to stakeholders in their terms.	Knowing how restaurant payouts and refunds work before redesigning payments.
Possess interpersonal skills	Lead, collaborate, mentor and resolve disagreement; most architecture work is done through other people.	Facilitating a session where two teams agree an event contract.
Understand and navigate politics	Gain support for decisions in an organisation where budgets, priorities and ownership are contested.	Securing agreement from finance and operations before retiring an old reporting system.

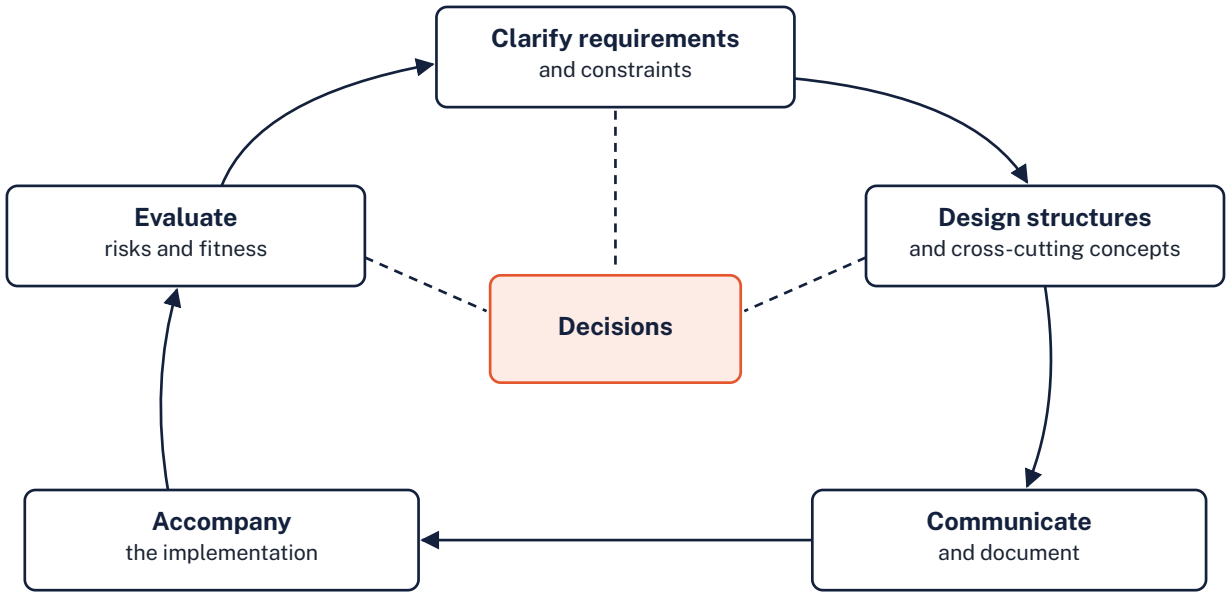
Notice how few of the eight are purely technical. Richards and Ford stress that nearly every decision an architect makes will be challenged by someone (developers, other architects, product owners, project managers), so negotiation and persuasion are core skills, not extras. An architecture that is correct on paper but not understood or accepted will not be built as intended.

The iSAQB view of architects' tasks

The International Software Architecture Qualification Board (iSAQB), whose Foundation Level curriculum (CPSA-F) is widely used in Europe, describes architects' tasks and responsibilities in a similar way. In summary, architects: clarify and scrutinise requirements and constraints; decide how to decompose the system into building blocks with their dependencies and interfaces; decide on cross-cutting concerns such as persistence, communication and user interface; communicate and document the architecture; accompany its realisation and implementation; analyse and evaluate it, especially for risks to meeting the requirements; and explain the consequences of architectural decisions to other stakeholders.

The curriculum also stresses that these tasks are iterative: architects work in cycles, revisiting requirements and decisions as feedback arrives, rather than designing once and handing over. Section 1.8 looks at how this fits into iterative and continuous delivery.

The architect's work as a cycle



Every task feeds and revisits decisions; the cycle repeats throughout the system's life

After the iSAQB curriculum: clarifying, designing, communicating, accompanying and evaluating repeat, and all of them revolve around decisions.

Kinds of architect

Titles follow the levels of architecture described in Section 1.2. Exact responsibilities vary widely between organisations, so treat the table below as typical rather than definitive.

Kind	Focus	Typical outputs
Enterprise architect	The whole organisation: capabilities, application portfolio, technology standards, alignment with strategy	Capability maps, target-state roadmaps, standards, investment advice
Solution architect	One business problem or programme spanning several systems, vendors and teams	Solution designs, integration designs, vendor evaluations
Software or application architect	The structure and qualities of one system or a group of closely related systems	Component structures, decision records, quality scenarios, technical guidance to teams
Data architect	How data is modelled, owned, stored, moved, governed and analysed	Data models, data ownership maps, pipeline and platform designs
Security architect	Identity, access, protection of data and the placement of security controls	Threat models, security patterns, control requirements
Cloud or infrastructure architect	Compute, network, storage, environments and platforms	Landing zones, network designs, platform standards, cost models

A role or a job title?

Architecture is a set of responsibilities before it is a job title. In many organisations, especially smaller ones and those organised around autonomous product teams, nobody is called "architect". The architectural work is still done, by technical leads, senior engineers or the team collectively. What matters is that someone clearly owns the significant decisions and the system-wide qualities. When nobody does, architecture is still produced, but by accident.

Many technology companies use a technical career track for senior individual contributors, with levels such as staff engineer and principal engineer. Will Larson's *Staff Engineer* (2021) describes four common archetypes of such roles, one of which is explicitly called the Architect: a person responsible for the direction and quality of a critical technical area. Others (the Tech Lead, the Solver and the Right Hand) also carry substantial architectural responsibility. In these organisations, the architect's work is done by people who are also engineers.

Martin Fowler's 2003 article *Who Needs an Architect?* contrasts two caricatures: an architect who makes all the important decisions alone, and one who works closely with the team, mentors developers and helps them make good decisions themselves. Fowler argued strongly

for the second. Most practitioners now agree that architects who dictate from a distance tend to produce designs that teams quietly work around.

Conceptual integrity: why someone must hold the whole

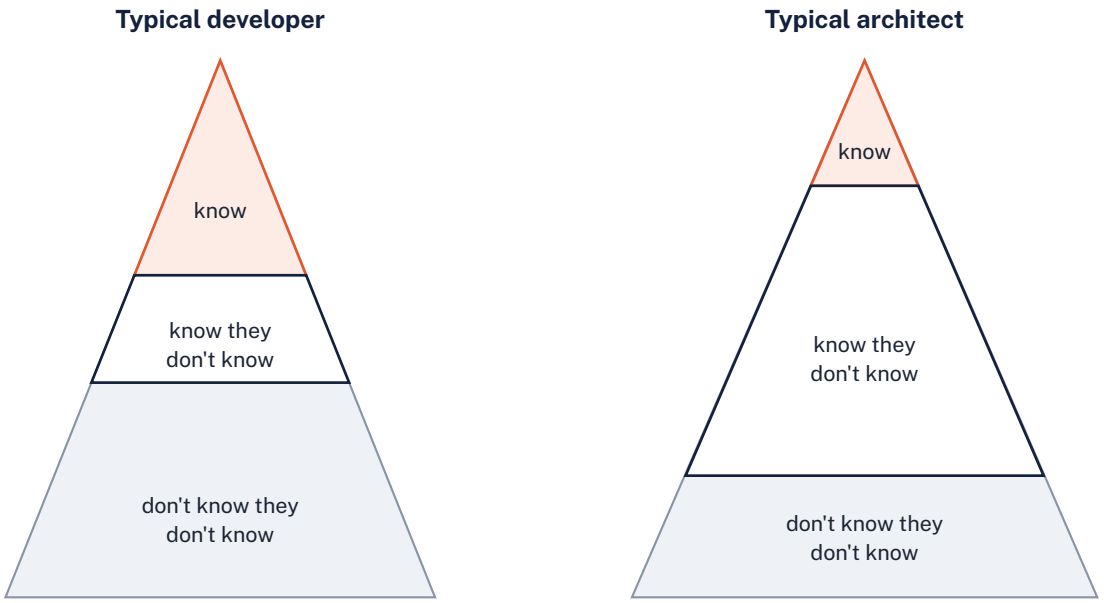
Fred Brooks gave the role its classic justification in *The Mythical Man-Month* (1975): "I will contend that conceptual integrity is the most important consideration in system design." A system should feel as if one mind designed it, with one set of ideas used consistently, even if that means leaving out good but unrelated features. He compared it with the unity of Reims cathedral, whose builders gave up some of their own ideas for the whole. His remedy was to separate architecture from implementation, and he accepted the charge of aristocracy. Modern teams share this work more widely, but the need remains: if the food-delivery platform has three ways of handling a refund, someone should notice and care.

Technical breadth versus depth

Developers are usually rewarded for depth: knowing one language, framework or platform extremely well. Architects need a different profile. Richards and Ford describe a knowledge pyramid with three layers: things you know (where you have real expertise), things you know you do not know (you know they exist and roughly what they are for, but could not build with them today), and things you do not know you do not know. Expertise in the top layer must be maintained, which takes time. The middle layer is cheap to grow and is what lets you recognise options.

An architect's value lies largely in technical breadth: knowing that five ways exist to solve a problem, what each is good and bad at, and which deserves a closer look. An architect who knows only one database will recommend it every time. Breadth comes at a price, because time spent widening the middle layer is not spent deepening the top; architects therefore deliberately let some old expertise fade and invest in knowing about more things. Richards and Ford warn of the opposite failure, the architect whose views were formed by one bad experience years ago and who rejects every option that resembles it.

The knowledge pyramid



Depth (top) still matters, but breadth (middle) is what lets an architect compare options

After Richards and Ford: an architect deliberately widens the middle layer, things known to exist but not mastered.

Should architects write code?

Practitioners genuinely disagree here, and the disagreement is useful because both sides are right about something.

The case for coding. Architects who code understand the real cost of their decisions, notice when a principle is painful in practice, keep their technical judgement current and keep the respect of the developers who must implement their designs. Architects who stop coding for years risk designing systems that are elegant on a whiteboard and awkward in an editor.

The case against. An architect's time is spread across many teams, meetings and decisions. An architect who owns critical-path features becomes a bottleneck: the team waits for code that is delayed by the architect's other duties. Their attention is also needed on cross-team concerns that no single team will look after.

The common resolution. Most experienced voices, including Richards and Ford, recommend that architects stay hands-on in ways that do not put them on the critical path. Examples include building a proof of concept to test a risky option before the team commits to it; writing fitness functions, automated checks that verify architectural rules such as "the domain module must not depend on the web framework"; fixing bugs and doing small technical-debt work; building tools and automation for the team; and reviewing code regularly, which shows how decisions are actually being implemented.

```
import pathlib, re

FORBIDDEN = re.compile(r"^\s*(from|import)\s+flask\b", re.M)

def test_domain_does_not_depend_on_web_framework():
    offenders = []
    for path in pathlib.Path("shop/domain").rglob("*.py"):
        if FORBIDDEN.search(path.read_text(encoding="utf-8")):
            offenders.append(str(path))
    assert not offenders, f"Domain code imports Flask: {offenders}"
```

A simple fitness function an architect might write: fail the build if any domain module imports the web framework.

A week in the life of a software architect (illustrative)

Day	Main activities	Expectation exercised
Monday	Planning with two product teams; spotting that a new loyalty feature needs data the order service does not expose; drafting options	Make decisions; know the business domain
Tuesday	Workshop with the data protection officer and security on storing customers' addresses; writing up a decision record	Interpersonal skills; ensure compliance
Wednesday	Two-day proof of concept of a managed event-streaming service to test ordering guarantees under load	Diverse exposure; keep current
Thursday	Code reviews on the payments team; adding a fitness function after spotting a forbidden dependency; one-to-one mentoring	Ensure compliance; interpersonal skills
Friday	Reviewing incident reports and lead-time metrics for signs of architectural decay; meeting finance about cloud costs; reading	Continually analyse; navigate politics; keep current

WORKED EXAMPLE

Worked example: one decision, five tasks

A hospital's appointment-booking system must send reminders by SMS. The software architect works through the iSAQB tasks for this one need.

Clarify requirements	Asks how many reminders per day (about 20,000), how quickly they must go out, what happens if the SMS provider is down, and whether messages may contain clinical details (they may not, for privacy).
Design	Proposes that the booking service publishes "appointment booked" events and a separate notification component sends messages from a queue, so an SMS outage never blocks bookings.
Communicate	Writes a one-page decision record with the options considered (direct call versus queue) and the reasons; presents it to the team and the information governance lead.
Accompany implementation	Pairs with a developer on the first version, reviews the retry logic, and adds a fitness function that blocks clinical fields from the message template.
Evaluate	Three months later checks delivery times and failure rates; finds reminders queue for 30 minutes during a provider outage and confirms with clinicians that this is acceptable.

Most of the architect's effort went into clarifying, communicating and checking, not into drawing the design.

The decision was recorded with its reasons, so a future team can revisit it if the context changes.

Questions people ask

How do I become an architect?

Usually by doing architectural work before having the title: owning cross-cutting decisions on your team, writing decision records, widening your technical breadth deliberately, learning the business domain and practising explaining trade-offs to non-technical people. Certifications such as the iSAQB CPSA-F can structure that learning but do not replace experience.

Does every team need an architect?
Every team needs architectural responsibility to be clearly owned, but not necessarily a dedicated person. Small teams often give it to a technical lead; larger organisations may have architects who each support several teams.

What is the difference between a technical lead and an architect?
A technical lead is usually responsible for one team's technical delivery and spends much of their time coding; an architect is usually responsible for decisions and qualities that span teams or systems. The roles overlap heavily, and in many organisations the technical lead is the architect for that team's system.

How much time should an architect spend in meetings?
There is no correct figure, but a useful test is whether the meetings produce decisions, shared understanding or removed obstacles. Gregor Hohpe's image of the architect riding an elevator between the boardroom and the engine room is a reminder that both ends need regular visits.

WATCH OUT

Anti-patterns in the architect role

- **The ivory tower:** designs handed down without involvement in implementation, which teams ignore or work around.
- **The bottleneck:** every decision, however small, must pass through the architect, so delivery slows and teams stop thinking.
- **The frozen expert:** judgement anchored in technologies or bad experiences from years ago.
- **Résumé-driven architecture:** choosing technologies because they are interesting to the architect rather than right for the problem.

CHECK YOUR UNDERSTANDING

▼ **Why do Richards and Ford include politics among an architect's expectations?**

Because almost every architectural decision will be challenged by someone, and a decision that cannot win support across budgets, priorities and teams will not be implemented as intended.

▼ **Which layer of the knowledge pyramid should an architect deliberately grow, and why?**

The middle layer, things you know you do not know, because knowing that options exist and what they are good for lets an architect compare solutions.

▼ **Name two ways an architect can stay hands-on without becoming a bottleneck.**

Any two of: building proofs of concept, writing fitness functions, fixing bugs or small debt items, building tools and automation, and reviewing code.

▼ **Try it: A food-delivery architect, the strongest coder present, offers to write the core payment module herself because it is too risky for anyone else. Is that a good use of the role? What would you advise her to do instead?**

No. Owning critical-path code makes her a bottleneck, and when architecture work pulls her away the team is stuck on the most important module. She should stay hands-on in other ways: build a proof of concept of the risky part, review the code, add fitness functions that protect the rules, and fix some bugs. A strong developer then owns the module, which spreads knowledge and leaves her able to carry responsibility for the wider decisions.

SOURCES

- iSAQB CPSA-Foundation curriculum (LG 01-04: tasks and responsibilities of software architects): https://public.isaqb.org/curriculum-foundation/curriculum-foundation-en.html
- Who Needs an Architect? Martin Fowler, IEEE Software, 2003 (PDF): https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- Staff archetypes (Will Larson, StaffEng): https://staffeng.com/guides/staff-archetypes/
- The Architect Elevator (Gregor Hohpe): https://architectelevator.com/
- Brooks: The Mythical Man-Month, chapter 4 (sample chapter): https://ptgmedia.pearsoncmg.com/images/0201835959/samplechapter/chap4.html

SECTION 1.5 OF 1.8 · BEGINNER · 20 MIN READ · LAST REVIEWED NOVEMBER 2026

Stakeholders, their concerns and how architecture serves them

THE IDEA IN BRIEF

Think about building a house. The family who will live in it want space and light. The bank lending the money wants it to be worth the loan. The planning office wants it to fit the street. The builder wants drawings that can actually be built, and the neighbours want no noise at night. All of them are stakeholders, and what each one worries about is a concern. A software system has its stakeholders too: users, payers, developers, the people who keep it running, regulators. Architecture is the art of hearing all of them and deciding openly where their wishes collide.

WHAT YOU NEED TO KNOW

- A stakeholder is any role, person, organisation or class of them with an interest, right, share or claim in a system, including people who never use it directly, such as regulators and operators.
- A concern, in ISO/IEC/IEEE 42010, is any matter of relevance or importance to a stakeholder: qualities, costs, risks, constraints, feasibility and organisational questions all count.
- Concerns are not yet requirements: capture them in the stakeholder's words, then sharpen them into measurable quality attribute scenarios, constraints or functional requirements.
- The power/interest grid helps plan how to engage each stakeholder, but it says nothing about whose concerns are most important; low-power groups can have the highest stakes.
- Conflicting concerns are normal; the architect makes them explicit, quantifies the options, names who decides and records the rationale.
- Concerns organise architecture documentation: each viewpoint frames certain concerns, and each view shows how the system addresses them.

Who the stakeholders are

A stakeholder is anyone with a legitimate interest in a system: people who use it, pay for it, build it, run it, regulate it or are affected by it. ISO/IEC/IEEE 42010:2022, the international standard for architecture description, defines the term broadly as a role, position, individual, organisation or class of these that has an interest, right, share or claim in the thing being described. Two features of that definition matter in practice. First, a stakeholder is often a *class*

rather than a person: "patients" or "the on-call engineers" rather than a named individual. Second, a claim or a right is enough: a regulator who never logs in to your system is still a stakeholder because it can stop you operating.

The list is longer than most teams expect. For a typical commercial system it includes:

Users (the people who operate the software day to day, often several distinct groups); **customers** (whoever pays, who may not be a user at all); the **product owner** or product manager who decides what gets built; the **sponsor** who funds the work and answers for its success; **developers** who must understand and change the code; **testers** who must be able to set up data and observe behaviour; **operations staff and site reliability engineers (SREs)** who deploy, monitor and recover it at three in the morning; the **security team**; the **data protection officer** where personal data is processed (under the UK and EU GDPR some organisations must appoint one); **legal and procurement**, who care about licences, contracts and liability; **finance**, who care about running costs as much as build costs; **regulators and auditors**; and **third-party integrators**, the other teams and companies whose software calls yours or is called by it.

Some stakeholders are absent from every meeting: the developers who will maintain the system in five years, the support desk that takes the complaints, people with disabilities who will use the interface, and members of the public affected by its decisions. Part of an architect's job is to speak for the ones who are not in the room.

Concerns: what stakeholders care about

In 42010 a concern is any matter of relevance or importance to a stakeholder. The word is deliberately wide. A concern can be a quality ("will the booking page stay fast on Monday mornings?"), a cost ("what will this cost to run per month?"), a risk ("what happens if the SMS provider goes down?"), a constraint ("must patient data stay in the UK?"), a feasibility question ("can our team build this with the skills we have?") or an organisational matter ("which team owns the integration with the GP system?").

Concerns are not yet requirements. They are usually expressed vaguely, in the stakeholder's own vocabulary, and they are often worries rather than specifications. "I don't want another outage like last winter" is a concern; "the booking service is available 99.9% of each calendar month, measured at the load balancer" is a requirement derived from it. Capturing the concern in the stakeholder's words first, and only then sharpening it, keeps you honest about where a requirement came from and why it matters.

One concern is often shared by several stakeholders, and one stakeholder usually has many concerns. Security is a concern of the security team, the data protection officer, the sponsor (who fears headlines) and patients (who fear their records leaking). That many-to-many

relationship is exactly why 42010 treats concerns as the hinge between people and architecture: decisions answer concerns, and views are organised around the concerns they address.

Typical stakeholders and their concerns

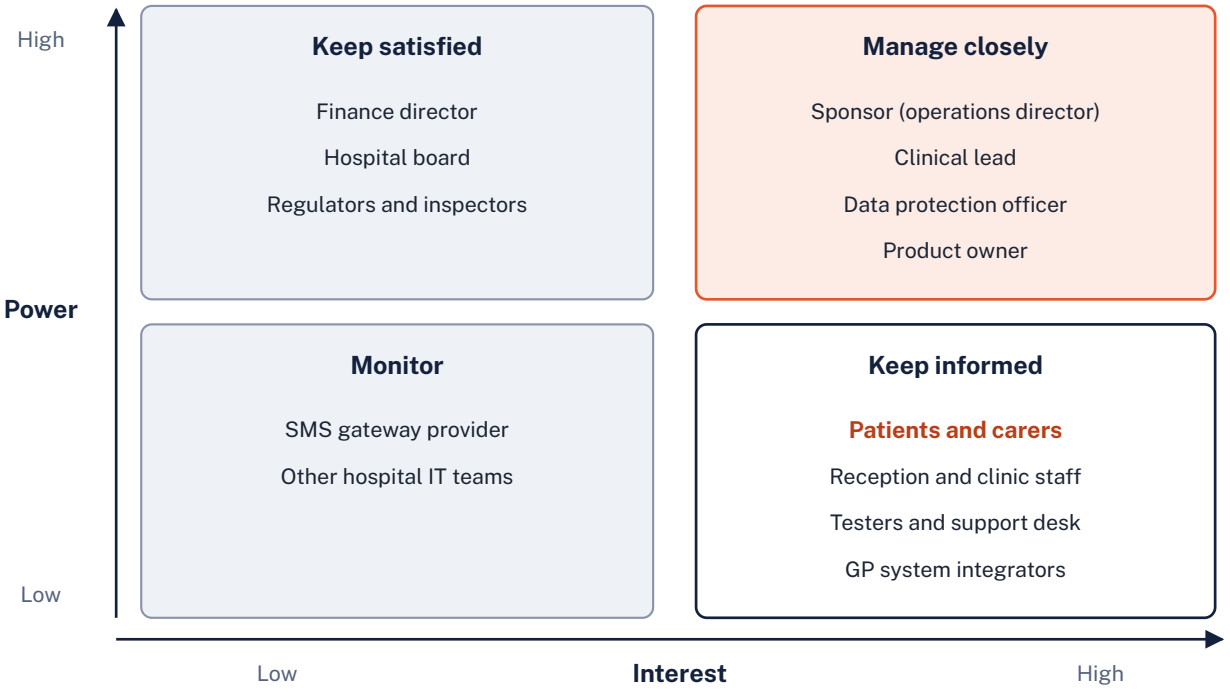
Stakeholder	Typical concerns	What the architecture must give them
Users	Speed, ease of use, accessibility, the system being there when needed	Response-time and availability targets; an interface layer that can meet accessibility standards
Customers and sponsors	Business value, time to market, total cost, risk to reputation	A credible delivery plan, cost model and list of the main risks
Product owner	How fast new features can be added, how options stay open	Modular structure aligned with the business domain; few hard-to-reverse decisions
Developers	Understandability, build and test speed, clear ownership	Clean module boundaries, a fast local build, documented conventions
Testers	Testability, controllable test data, observable behaviour	Seams where dependencies can be replaced; test environments that resemble production
Operations and SRE	Deployability, monitoring, recovery, capacity, on-call load	Health checks, logs, metrics and traces; safe rollback; runbooks
Security team	Attack surface, identity, secrets, vulnerability response	Trust boundaries, authentication design, patching and dependency strategy
Data protection officer	Lawful basis, data minimisation, retention, subject rights, transfers	A data inventory and data-flow view; deletion and export mechanisms
Legal and procurement	Licences, contracts, liability, vendor lock-in	A list of third-party components and services with their terms
Finance	Build cost, running cost, cost predictability	A cost model per environment and per unit of usage
Regulators and auditors	Compliance with sector rules, evidence of control	Audit trails, traceability from requirement to test, documented decisions
Third-party integrators	Stable interfaces, versioning, documentation, test sandboxes	Published API contracts and a versioning policy

Mapping stakeholders: the power/interest grid

With twenty or more stakeholder groups you cannot give each the same attention. A widely used tool is the power/interest grid, usually attributed to Aubrey Mendelow, whose paper on stakeholders and environmental scanning dates from 1981 (it is often cited as 1991). Each stakeholder is placed according to how much power they have over the system (to fund, approve, block or reshape it) and how much interest they take in it. The quadrants suggest an engagement strategy: **manage closely** (high power, high interest), **keep satisfied** (high power, low interest), **keep informed** (low power, high interest) and **monitor** (low power, low interest).

The grid is about engagement, not about whose concerns count. Patients of a hospital have little formal power over a booking system's design but the highest stakes of anyone, and some legal duties (accessibility, data protection) exist precisely to protect people with little power. Treat the grid as a reminder of whom to talk to and how often, and keep a separate judgement about which concerns are critical. Positions also move: a regulator with low interest becomes the most interested stakeholder of all the week after a data breach.

A power/interest grid for a hospital booking system



Patients sit in the low-power quadrant yet have the highest stakes, which is why the grid guides engagement and not priority.

A lightweight stakeholder analysis

1. **Identify**
   Brainstorm stakeholder classes with the sponsor and the team, then check the list against a standard set (users, payers, builders, operators, protectors, regulators, integrators). Ask each group you find: who else should we talk to?

2. **Elicit concerns**
   Interview or workshop each important group. Record concerns in their own words, then note the quality, constraint or risk behind each one.

3. **Map and prioritise**
   Place stakeholders on the power/interest grid to plan engagement. Separately, rank the concerns by business impact and by how strongly they constrain the architecture.

4. **Make conflicts explicit**
   Where two concerns pull in opposite directions, write the conflict down, name the people who must agree, and agree who decides.

5. **Trace and revisit**
   Link each significant decision to the concerns it addresses, and repeat the analysis at major milestones, because stakeholders and their concerns change.

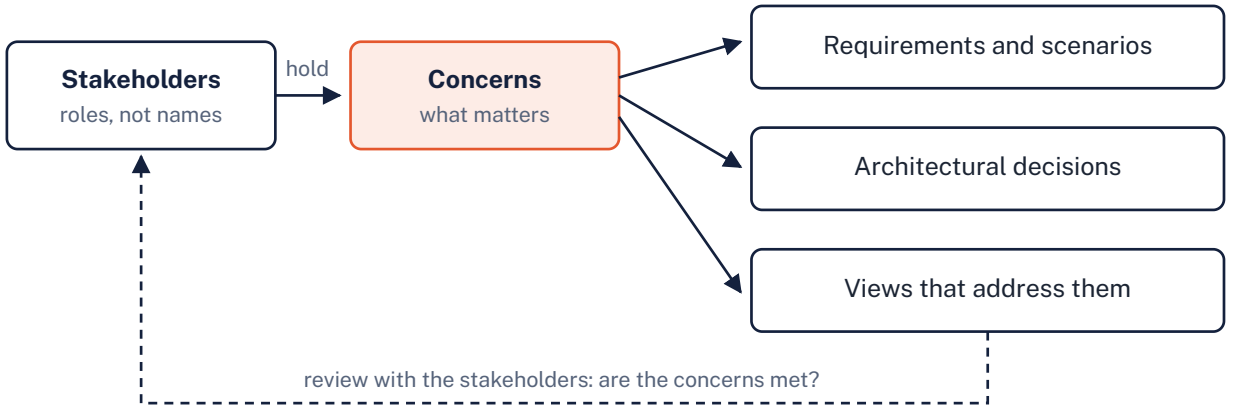
Conflicting concerns and how to make them explicit

Concerns conflict all the time, and an architecture that pleases everyone usually means a conflict has been hidden rather than resolved. The security team wants every login protected by multi-factor authentication; the patient representative points out that many elderly patients have no smartphone. Finance wants a single small database server; operations wants a standby replica in a second data centre. The product owner wants a new feature every week; the testers want time to regression-test the integration with the hospital's patient administration system.

An architect's contribution is not to pick a winner quietly but to make the conflict visible, quantified and owned. In practice that means four things. **State both concerns precisely**, ideally as testable scenarios, so the argument is about numbers rather than adjectives. **Show the options and their consequences** side by side: for example, MFA for staff and for patients who opt in, with a one-time code by text message or landline call as a fallback. **Name the decision owner**: some trade-offs belong to the sponsor or to a risk committee, not to the architect. **Record the outcome and its rationale**, so that when someone asks in two years why patients can log in without MFA, the answer exists. The next section, Decisions and trade-offs, covers the mechanics of weighing such options.

Some conflicts dissolve on inspection. The security team may not need MFA for every patient action, only for viewing test results; the patient group may be content with it there. Asking each side *why* they hold the concern, and what outcome they actually fear, often reveals a design that satisfies both.

Where concerns go



Concerns are the link between people and architecture: they become requirements, drive decisions and decide which views you draw.

How concerns feed requirements and documentation

Concerns are the raw material for two later activities. The first is requirements engineering. A concern about speed becomes a quality attribute requirement, which Chapter 2 teaches you to write as a measurable scenario (source, stimulus, artifact, environment, response and response measure): see Writing quality attribute scenarios. A concern such as "patient data must stay in the UK" becomes a *constraint*, a decision already made for you. Concerns that turn out not to affect the structure of the system stay as ordinary functional or project requirements.

The second is documentation. 42010 organises an architecture description around concerns: a viewpoint names the concerns it frames and the stakeholders it serves, and a view applies that viewpoint to your system. The operations team's concerns call for a deployment view showing servers, regions and failover; the data protection officer's call for a data-flow view showing where personal data travels and where it rests; developers need a module view showing what depends on what. A diagram that does not answer any stakeholder's concern is decoration. Chapter 4 covers this properly in Views and viewpoints.

Traceability ties the two together. If you can follow a line from a stakeholder, through a concern, to a requirement, a decision and a view, you can answer the two questions every reviewer eventually asks: why is it built like this, and have we forgotten anyone?

WORKED EXAMPLE

Worked example: stakeholders of a hospital appointment-booking system

A regional hospital trust wants an online system for patients to book, change and cancel outpatient appointments, replacing telephone booking. It must integrate with the existing patient administration system (PAS) and send reminders by text message. Here is the first-pass stakeholder analysis an architect might produce in the first fortnight.

Patients and carers	Concerns: easy booking on a phone, accessibility (screen readers, large text), reminders, privacy. Architecture impact: responsive web front end that meets WCAG 2.2 AA; no app install required; minimal data shown on screen.
Reception and clinic staff	Concerns: no double bookings, clinic lists that match reality, fewer phone calls. Impact: the PAS remains the single source of truth for slots; the booking system must not hold its own copy that can drift.
Clinical lead	Concerns: urgent referrals must not be crowded out by routine bookings; clinical safety. Impact: slot rules owned by clinicians and configurable without a release; a clinical safety case (in England, under the DCB0129 standard) covering the system.
Data protection officer	Concerns: health data is special category data under the UK GDPR; minimisation; retention; audit. Impact: a data-flow view, encryption in transit and at rest, audit logs of who viewed what, a retention policy.
Operations team	Concerns: Monday-morning peaks, the PAS's limited capacity, out-of-hours support. Impact: a queue or rate limiter in front of the PAS; caching of slot availability with short expiry; monitoring of PAS response times.
Finance director	Concerns: fixed budget, predictable running cost, saving on call-centre staff. Impact: managed cloud services sized for the peak; a cost model per thousand bookings.
PAS supplier (integrator)	Concerns: no unsupported load on its system; use of its published interface only. Impact: integration through the supplier's interface with agreed rate limits; a contract test suite.
Main conflict found	Operations wants to cache slot availability to protect the PAS; reception staff fear double bookings from stale data. Resolution proposed: cache for display only (30 seconds), but confirm every booking against the PAS in real time, with a clear message if the slot has just gone.

Seven stakeholder groups produced at least five decisions with structural consequences (single source of truth, rate limiting, audit logging, configurable rules, accessible front end) before any feature had been designed in detail.

The most consequential conflict sat between two low-profile groups, operations and reception staff, neither of whom would have dominated a steering meeting.

Each row can be traced forward to a quality attribute scenario, a decision record and a view, and backward to a named class of people.

WATCH OUT

Common mistakes

These are the ways stakeholder work most often goes wrong.

- Treating "the business" as one stakeholder. Sales, finance, compliance and customer support want different, often conflicting, things.
- Forgetting the operators. Systems designed only around users and developers are routinely hard to deploy, monitor and recover.
- Confusing loudness with importance. The person who attends every meeting is not necessarily the one whose concern will sink the project.
- Relying on proxies. A product owner can represent users, but cannot always tell you what an accessibility user or a night-shift nurse actually experiences. Talk to some real ones.
- Doing the analysis once. Stakeholders arrive and leave, regulations change, and a new integration partner brings new concerns. Revisit the list at each major milestone.

Questions people ask

Is the development team really a stakeholder? We are building it.
Yes. Developers live with the architecture longer than almost anyone, and their concerns (build time, testability, clear ownership) translate directly into delivery speed. 42010's definition makes no distinction between builders and users.

What is the difference between a stakeholder concern and a requirement?
A concern is the underlying interest, often vague ("I want it to be reliable"). A requirement is a precise, verifiable statement derived from one or more concerns ("99.9% monthly availability"). Several requirements may serve one concern, and some concerns turn into constraints instead.

Who decides when two powerful stakeholders disagree?
Usually not the architect alone. The architect frames the options and consequences; the decision belongs to whoever is accountable for the outcome, typically the sponsor or a governance board. Agreeing in advance who decides which kind of trade-off saves weeks of argument.

CHECK YOUR UNDERSTANDING

▼ Name three stakeholders of a typical commercial system who rarely attend design meetings.

For example: the developers who will maintain it in future years, the support desk, users with disabilities, operations staff on call at night, and members of the public affected by its decisions.

▼ A sponsor says "it must be secure". Is that a concern or a requirement?

A concern. It needs to be refined into specific, testable requirements, for example about authentication, encryption or audit logging, before it can guide decisions.

▼ Why can one concern, such as security, belong to several stakeholders at once?

Because different people fear different consequences of the same quality: the security team fears attacks, the data protection officer fears breaches of personal data, the sponsor fears reputational damage and users fear their data leaking.

▼ What links a stakeholder concern to an architecture view?

A viewpoint: it states which concerns and stakeholders it serves and how to model them, and the view applies it to the system.

▼ Try it: Marketing, powerful and keen, wants an animated seat map for sale day. A small disability group says screen readers cannot use it. A manager places both on the power/interest grid and says marketing wins. Critique that reasoning and say what to do.

The grid only helps plan how to engage people; it says nothing about whose concern matters most, and low-power groups can have the highest stakes. Here some customers could not buy at all. Record both concerns in their own words, then make the conflict explicit: for example a keyboard and screen-reader list alongside the map, with its cost. Quantify the options, name who decides and record the reasoning, including any accessibility duties.

SOURCES

- ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description (ISO): https://www.iso.org/standard/74393.html
- ISO/IEC/IEEE 42010 summary (arc42 quality model): https://quality.arc42.org/standards/iso-42010
- Stakeholder analysis for UX projects (Nielsen Norman Group): https://www.nngroup.com/articles/stakeholder-analysis/

- arc42 section 3: Context and scope: https://docs.arc42.org/section-3/

SECTION 1.6 OF 1.8 · INTERMEDIATE · 24 MIN READ · LAST REVIEWED NOVEMBER 2026

Architectural decisions and trade-offs: reversibility and timing

BEFORE YOU READ THIS

This section builds on:

- 1.2 Architecture versus design
- 1.5 Stakeholders and concerns

Terms it uses from earlier: Stakeholder, Concern, Architecturally significant

THE IDEA IN BRIEF

Choosing a car is a trade-off: a bigger engine costs more fuel, a smaller car is easier to park but carries less. Nobody gets everything. Some choices are easy to undo, like picking a paint colour you can change later; others, like buying the house next to a motorway, are very hard to undo. Software architecture is full of choices like these. Good architects make the easy-to-undo ones quickly, take care over the hard-to-undo ones, decide at the right time rather than too early or too late, and write down why they chose, so the people who come after them understand.

WHAT YOU NEED TO KNOW

- Every significant architectural choice is a trade-off: it improves some qualities at the expense of others, and the downsides must be found and stated.
- A well-formed decision records its context, the options considered, the criteria, the decision with its rationale, the consequences and its status.
- Reversibility is the key property of a decision: two-way doors should be decided quickly and locally, one-way doors deliberately and with the people who bear the consequences.
- The last responsible moment is the point at which delay would remove an important option; deciding then uses the most knowledge without letting the decision be made by default.
- Abstractions at boundaries, feature flags, incremental migration and spikes make decisions more reversible, at a cost worth paying only where uncertainty and reversal cost are both high.
- Weighted decision matrices and Architecture Decision Records make reasoning visible and open to challenge; agree criteria and weights before scoring, and test how sensitive the result is to them.

Everything is a trade-off

Every significant architectural choice buys something by giving something else up. Splitting a system into independently deployable services buys team autonomy and independent scaling, and pays in network latency, operational complexity and harder data consistency. Caching buys speed and pays in staleness and invalidation logic. A managed cloud database buys freedom from patching and backups and pays in cost and dependence on one provider. Mark Richards and Neal Ford put this at the head of *Fundamentals of Software Architecture* (2020; second edition 2025) as their first law: “Everything in software architecture is a trade-off.” A corollary they draw is that if you think you have found something that is not a trade-off, you have probably not yet identified the trade-off.

Their second law is that *why* is more important than *how*. Anyone can look at a running system and see how it is built: three services, a message broker, a document database. What cannot be recovered from the code is why: which alternatives were considered, which concerns won, which risks were accepted. Without the why, later engineers either preserve choices that no longer make sense or undo choices whose reasons still apply. The second edition adds a third law: most architectural decisions are not binary but sit on a spectrum between extremes. You rarely choose between "monolith" and "microservices"; you choose how many deployable units, split along which lines, sharing what.

Thinking in trade-offs is the habit that most clearly separates architectural thinking from enthusiasm. When someone proposes a technology, the useful questions are not "is it good?" but "what does it cost, what does it make harder, and are those costs acceptable given our concerns?" The previous section showed where those concerns come from.

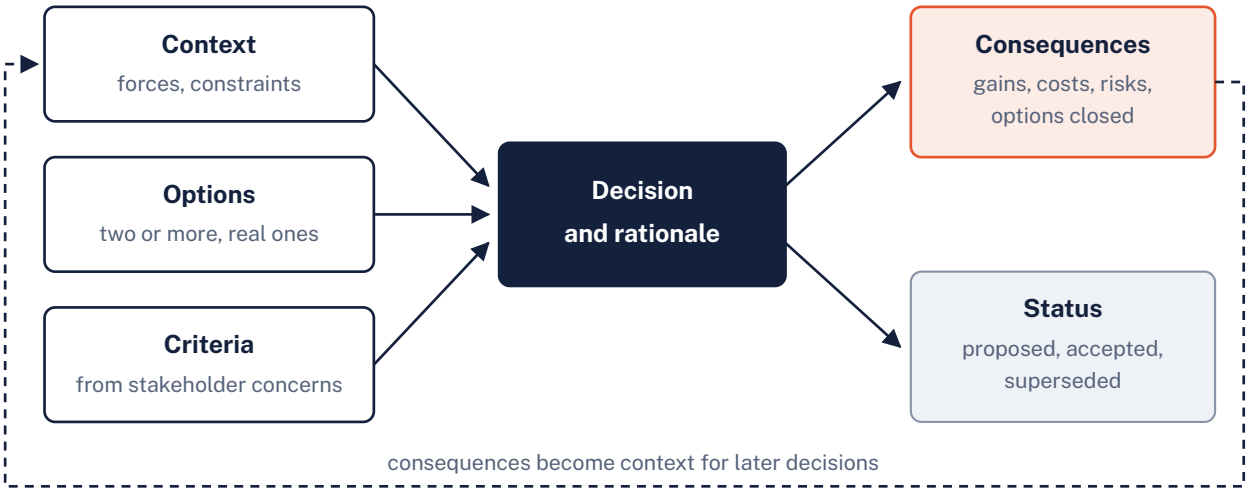
The anatomy of a decision

An architectural decision is more than its outcome. A decision that can be explained, reviewed and revisited has six parts, and naming them is the first step to making decisions well.

Context: the forces at play. The problem being solved, the quality attribute requirements that bear on it, the constraints (budget, deadlines, regulations, existing systems, team skills) and the assumptions being made. **Options**: at least two genuine alternatives, often including "do nothing" or "defer". A decision with only one option considered is an announcement, not a decision. **Criteria**: how options will be judged, derived from stakeholder concerns and weighted by importance. **Decision and rationale**: the option chosen and, crucially, why it beat the others on the criteria. **Consequences**: what becomes easier, what becomes harder, what new risks appear, and which future options are now closed. **Status**: proposed, accepted, deprecated or superseded, plus who made the decision and when.

Consequences deserve particular attention because they are where honesty is tested. Every option has downsides; writing them down protects the team from surprise and gives the next decision-maker a fair account. The consequences of one decision also become part of the context for the next: once you have chosen asynchronous messaging, you must decide how to handle duplicate and out-of-order messages.

Anatomy of an architectural decision



Context, options and criteria feed a decision; its consequences and status are recorded and become context for later decisions.

One-way doors and two-way doors

The most useful single property of a decision is its reversibility: how cheaply and safely it can be undone. In his 2015 letter to Amazon shareholders, Jeff Bezos distinguished **Type 1** decisions, which are consequential and irreversible or nearly so ("one-way doors"), from **Type 2** decisions, which are changeable and reversible ("two-way doors"). Type 1 decisions deserve slow, careful deliberation and consultation. Type 2 decisions should be made quickly by individuals or small groups with good judgement, because if they turn out wrong you can walk back through the door. His warning was that large organisations tend to apply the heavyweight Type 1 process to everything, which makes them slow and timid.

The idea translates directly into architecture. A one-way door in software is a decision that would require rewriting large parts of the system, migrating large amounts of data, renegotiating contracts or retraining many people to undo: the primary programming language of a large codebase, the data model of a system of record, the public API that hundreds of partners integrate with, a multi-year cloud commitment. Two-way doors include the choice of a logging library behind your own interface, the internal structure of one module, the tuning of a cache, or which of two equivalent message formats to use inside one service.

Two cautions. First, reversibility is a spectrum, not a binary: most decisions sit somewhere between "trivial to undo" and "impossible to undo", and the cost of reversal usually grows as more code, data and people come to depend on the choice. A decision that is a two-way door in month one can become a one-way door by month twelve. Second, you can often change which kind of door you are facing, by design, as the section on reversibility below shows.

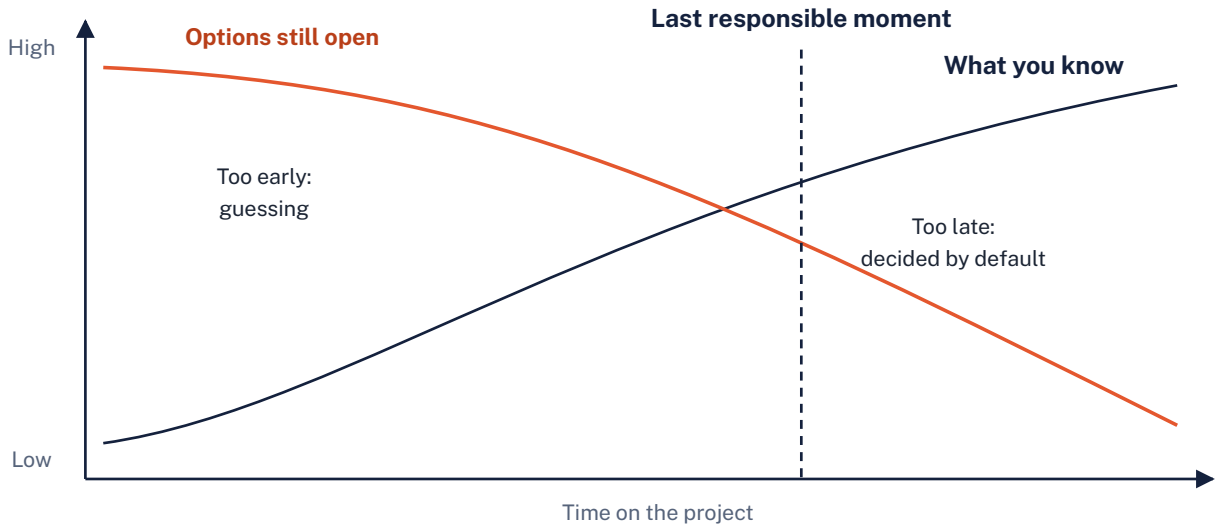
Timing: the last responsible moment

When should a decision be made? Mary and Tom Poppendieck, in *Lean Software Development: An Agile Toolkit* (2003), popularised the idea of deciding at the last responsible moment: the moment at which failing to decide would eliminate an important alternative. Before that moment you are deciding with less information than you could have; after it, the decision is made for you by default, usually badly.

The principle cuts both ways. It argues against deciding too early: fixing the database technology in week one, before you understand the data's shape and access patterns, is guessing. It equally argues against deciding too late. If the team has started writing persistence code against an unspoken assumption, the decision has already been made, just not consciously. If procurement needs six weeks to approve a cloud provider, the last responsible moment for that choice is at least six weeks before you need the environment.

Practitioners disagree about how far to push this. Critics point out that "last responsible moment" is easily misused as a licence to procrastinate, that some decisions (team structure, programming language) shape everything that follows and so must be made early, and that indecision has its own cost: teams cannot work in parallel without agreed interfaces. Advocates reply that the principle never meant "as late as possible" but "as late as is responsible", and that identifying the moment is itself an act of planning. Both positions agree on the practical core: know which decisions are pending, know when each must be made, and use the time before then to learn.

Deciding too early, too late and at the right time



Knowledge grows as the project proceeds while open options close; the last responsible moment is the latest point before an important option disappears.

IF YOU ARE NEW TO THIS

Step by step: when to choose the order service's database

This section's hardest idea is the last responsible moment: a decision should be made neither as early nor as late as possible, but at the latest point before waiting would remove an important option. Follow one decision through time: the database for the food-delivery company's new order service, scored later in this section's worked example.

- Week one. The team could pick a database straight away, but it does not yet know how orders will be saved and read. Choosing now would be a guess: deciding too early.
- So the team waits, and uses the time to learn. It finds, for example, that each order and its payment must be saved together, all or nothing, as one transaction: exactly the kind of fact a database choice should rest on.
- Waiting has a deadline, though. Suppose any new data store needs a three-week security review, and the database must be running for testing in week nine. That makes week six, at the latest, the last responsible moment for the choice.
- There is a quieter deadline too. If developers start writing the code that saves orders, and that code quietly assumes one particular database, the choice has been made by default: nobody decided it, but changing it now means rewriting that code.
- So the team names the decision, gives it an owner and a date before either deadline, decides then with what it has learnt, and records the assumptions behind it.
- **The lesson:** the last responsible moment does not mean 'as late as possible'. Know which decisions are pending and when each must be made, use the time before then to learn, and decide before the choice is made for you.

Designing for reversibility

The cheapest way to make a hard decision is often to turn it into an easier one. Several techniques lower the cost of changing your mind later, at the price of some effort now.

Abstraction at the boundary. Put a narrow interface of your own between your core logic and a technology you might replace, such as a payment provider, a message broker or a large language model API. Swapping the technology later then touches one adapter instead of the whole codebase. The idea goes back to David Parnas's information hiding (1972) and reappears in hexagonal architecture (Chapter 5). **Feature flags.** Ship new behaviour switched off, turn it on for a small share of users, and switch it off again in seconds if it misbehaves; this makes releases two-way doors. **Incremental migration.** Run old and new side by side and move traffic gradually rather than switching in one step. **Spikes and proofs of concept.** Spend a few days building the riskiest part with a candidate technology before committing to it. **Contractual exits.** Prefer shorter commitments, standard data formats and the ability to export your data.

This is *options thinking*, borrowed from finance: you pay a small premium now (an adapter, a flag, a spike) to buy the right, but not the obligation, to change course later. Like any option, it is only worth buying when the premium is small relative to the value of the flexibility. Wrapping every library in an abstraction "in case we change it" is a classic way to add cost and indirection for flexibility nobody uses. Buy options on the decisions that are both uncertain and expensive to reverse; for everything else, decide and move on.

DECISION GUIDE

Decision guide: decide now or defer?

Use this for any decision that might be architecturally significant. Work down the questions in order.

Q1 **Is other work blocked, or will an important option disappear soon if nobody decides (for example a procurement lead time or a team starting to build on an assumption)?**

- **Yes** You are at the last responsible moment. Decide now with the information you have, record the decision and its assumptions, and set a date to review it.
- **No** Go to the next question.

Q2 **Is the decision cheap and safe to reverse (a two-way door)?**

- **Yes** Decide quickly, at the lowest sensible level (often by the team that owns the code), record it briefly and move on. Do not spend a week debating it.
- **No** It is a one-way door. Go to the next question.

Q3 **Can you cheaply make it more reversible, for example with an interface around the technology, a feature flag or an incremental migration path?**

- **Yes** Buy that option first, then treat the remaining decision as a two-way door.
- **No** Go to the next question.

Q4 **Would information you can realistically obtain before the deadline (a spike, a load test, a conversation with a stakeholder) change the choice?**

- **Yes** Defer deliberately: name the decision, the owner, the date by which it must be made and the experiment that will inform it.
- **No** More waiting will not help. Run a structured comparison now (criteria, weights, options), involve the stakeholders who own the consequences, and write an Architecture Decision Record.

Cognitive traps

Architects are human, and architectural decisions are made under uncertainty, time pressure and social pressure: ideal conditions for cognitive biases. Knowing the common ones by name makes them easier to catch in yourself and to challenge kindly in others.

Traps that distort architectural decisions

Trap	What it looks like	Countermeasure
Anchoring	The first option mentioned, or the first estimate given, becomes the reference point that every other option is judged against.	Generate several options independently before discussing any; ask people to write estimates down before sharing.
Sunk cost	"We have spent eight months on this framework, we cannot drop it now", even though the past spend is gone whatever you decide.	Judge options only on future costs and benefits; ask what you would choose if starting today.
Groupthink	A cohesive team converges quickly, dissent feels disloyal, and nobody voices the obvious objection.	Assign someone to argue the other side; collect views in writing before the meeting; invite an outside reviewer.
Resume-driven development	A technology is chosen because it is exciting to learn or looks good on a CV, not because it fits the problem.	Require every choice to be justified against the agreed criteria; budget learning time separately.
Hype and bandwagon	"Everyone is moving to X" stands in for analysis; conference talks from very large companies are applied to small teams.	Ask what problem X solved for its adopters and whether you have that problem at that scale.
Confirmation bias	Evidence for the favoured option is sought and remembered; evidence against it is explained away.	Write down in advance what result would change your mind, then run the spike or test.

WORKED EXAMPLE

Worked example: a weighted decision matrix

A food-delivery company is building a new order service. Orders involve payment and must never be lost or double-charged. The team of six knows PostgreSQL well and has no dedicated database administrator. Three options are on the table: (A) a managed PostgreSQL service from their cloud provider, (B) the provider's proprietary managed document database, and (C) PostgreSQL self-hosted on virtual machines. The team builds a weighted decision matrix: it agrees criteria and weights from the stakeholder concerns (5 = most important), then scores each option from 1 (poor) to 5 (excellent).

Transactional integrity (weight 5)	A: 5, B: 3, C: 5. Multi-row transactions are central to orders and payments; the document store supports them with more restrictions.
Operational effort (weight 4)	A: 4, B: 5, C: 2. Self-hosting means patching, backups and failover by a team with no database specialist.
Team familiarity (weight 3)	A: 5, B: 2, C: 4.
Cost at expected load (weight 3)	A: 3, B: 4, C: 4.
Scaling headroom (weight 2)	A: 3, B: 5, C: 3. Expected growth is about three times in two years, well within a single large PostgreSQL instance.
Reversibility and lock-in (weight 2)	A: 4, B: 2, C: 5. Standard PostgreSQL can move between providers; the proprietary API cannot.
Weighted totals (maximum 95)	A: 25 + 16 + 15 + 9 + 6 + 8 = **79**. B: 15 + 20 + 6 + 12 + 10 + 4 = **67**. C: 25 + 8 + 12 + 12 + 6 + 10 = **73**.
Sensitivity check	Raise scaling headroom to weight 5 (as if expecting fifty-fold growth): A 88, B 82, C 82. Lower operational effort to weight 1 (as if a platform team ran databases for them): A 67, B 52, C 67, a tie between A and C.
Decision	Option A, managed PostgreSQL. It wins under the agreed weights and stays first or joint first under plausible changes. Recorded consequences: a higher monthly bill than self-hosting; dependence on the provider's maintenance windows; a plan to keep schema and queries standard so that option C remains a fallback.

The numbers do not make the decision; they make the reasoning visible. Anyone who disagrees can point to a specific weight or score instead of arguing about the whole.

The sensitivity check is the most informative step: it shows which assumptions the result depends on. Here the outcome hinges mainly on the team's lack of database operations capacity.

Matrices can be gamed by choosing criteria and weights after the favoured option is known. Agree criteria and weights before scoring, ideally with the stakeholders whose concerns they represent.

Recording decisions: a preview of ADRs

The second law says the why matters more than the how, so the why must be written down. The most widely used lightweight format is the Architecture Decision Record (ADR), proposed by Michael Nygard in a 2011 blog post. An ADR is a short text file, kept with the code in version control, that records one decision using the anatomy described above: title, status, context, decision and consequences. ADRs are never edited to change history; a later decision supersedes an earlier one and links to it, so the log tells the story of how the architecture evolved.

Chapter 4 covers templates, tooling and the habits that keep ADRs useful in Architecture Decision Records. The example below records the decision from the worked example.

```
ADR 7: Use managed PostgreSQL for the order service

Status: Accepted (2026-10-12)

Context
Orders and payments need multi-row transactions and must never be lost.
The team knows PostgreSQL well and has no database administrator.
Expected growth is about 3x in two years.

Decision
We will use the cloud provider's managed PostgreSQL service.

Consequences
+ Backups, patching and failover are handled by the provider.
+ Standard SQL keeps self-hosting or another provider open as a fallback.
- Monthly cost is about 30% higher than self-hosting at current load.
- Maintenance windows are set by the provider.
Revisit if write volume exceeds what one primary instance can handle.
```

A short ADR for the order-service database decision, following Nygard's structure

WATCH OUT

Common mistakes

- Debating two-way doors at length while making one-way doors in passing ("we'll just use the shared customer table for now").
- Considering only one option. If you cannot name a credible alternative, you have not understood the decision.
- Hiding the downsides. Consequences that list only benefits will be found out, and trust in the architect goes with them.
- Deciding by seniority or volume rather than by criteria agreed in advance.
- Never revisiting. Decisions rest on assumptions; record them, and review the decision when an assumption changes.

CHECK YOUR UNDERSTANDING

▼ What is the difference between Richards and Ford's first and second laws?

The first says every architectural choice is a trade-off. The second says the reasoning behind a choice (why) matters more than its implementation (how), because the why cannot be recovered from the code.

▼ Give one example of a two-way door and one of a one-way door in a typical web application.

Two-way: the logging library used behind your own interface, or a cache's expiry setting. One-way: the data model of a system of record, or a public API that many partners already depend on.

▼ Why can deciding too early be as harmful as deciding too late?

Too early means deciding with less information than you could have had, often by guessing. Too late means the decision is effectively made by default through assumptions already built into code or plans, or an option has expired.

▼ What does a sensitivity check add to a weighted decision matrix?

It shows whether the winning option still wins when weights or scores change within plausible ranges, revealing which assumptions the decision depends on.

▼ **Try it: Weights: reliability 5, team skills 3, cost 2. Database A scores 4, 5, 3; database B scores 5, 2, 4. Which wins? Now make the weights 5, 2, 3. What does this tell you about the decision?**

With 5, 3, 2: A = 20 + 15 + 6 = 41 and B = 25 + 6 + 8 = 39, so A wins narrowly. With 5, 2, 3: A = 20 + 10 + 9 = 39 and B = 25 + 4 + 12 = 41, so B wins. The result is sensitive to the weights, so the matrix does not settle it alone. Agree the weights before scoring, then discuss how much team skills truly matter, and record that judgement.

SOURCES

- Documenting Architecture Decisions, Michael Nygard (2011): https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
- Architectural Decision Records (adr.github.io): https://adr.github.io/
- Amazon.com 2015 letter to shareholders (SEC filing): https://www.sec.gov/Archives/edgar/data/1018724/000119312516530910/d168744dex991.htm
- Feature Toggles (aka Feature Flags), Pete Hodgson on martinfowler.com: https://martinfowler.com/articles/feature-toggles.html

SECTION 1.7 OF 1.8 · INTERMEDIATE · 27 MIN READ · LAST REVIEWED NOVEMBER 2026

Technical debt, architecture erosion and drift

BEFORE YOU READ THIS

This section builds on:

- 0.3 Organising code
- 0.8 How software gets built
- 1.3 Why it matters

Terms it uses from earlier: Module, Dependency, Cost of change, Version control

THE IDEA IN BRIEF

Imagine you move house in a hurry and throw everything into boxes without labels. You moved on time, which was the point, but now every time you need something you waste ten minutes searching. Those wasted minutes are interest. Unpacking and labelling properly is paying back the debt. Software works the same way: shortcuts, or designs that made sense before you understood the problem better, make every later change slower. A little of this is fine and sometimes wise. The danger is letting it pile up until the team spends most of its time searching through boxes.

WHAT YOU NEED TO KNOW

- Ward Cunningham coined the debt metaphor in 1992 to explain shipping code that reflects current understanding and then refactoring as understanding grows; it was never a licence for careless code.
- Principal is the cost of fixing the problem; interest is the extra cost paid on every change while it remains, so debt in code that never changes costs nothing.
- Fowler's quadrant classifies debt as deliberate or inadvertent and prudent or reckless; prudent debt is a normal by-product of delivering and learning, reckless debt is a skills or discipline problem.
- Architectural debt, in structure rather than code, is the most expensive kind because repaying it touches many modules, teams and often data.
- Perry and Wolf (1992) distinguished erosion, caused by violations of the architecture, from drift, caused by insensitivity to it.
- Hotspots (complex and frequently changed code), change coupling, dependency cycles and delivery trends make debt visible; a register, protected capacity and refactoring alongside features keep it under control.

What Ward Cunningham actually meant

The term technical debt comes from Ward Cunningham's experience report on the WyCash portfolio management system, presented at the OOPSLA conference in 1992. He wrote that shipping first-time code is like going into debt: a little debt speeds development, as long as it is paid back promptly with a rewrite, and the danger comes when it is not repaid: “Every minute spent on not-quite-right code counts as interest on that debt.” Whole engineering organisations, he warned, can be brought to a standstill under the debt load.

The metaphor is often misread as permission to write sloppy code. Cunningham later explained (in a short 2009 video) that this was never his meaning. His point was about *understanding*. A team ships code that reflects its current, partial understanding of a problem, because that is the fastest way to learn from real use. As understanding improves, the code no longer matches what the team now knows, and every change has to work around the mismatch. Repaying the debt means refactoring the code so that it reflects what has been learnt. That only works if the code was written cleanly enough to be refactored in the first place. On this reading, debt is a by-product of learning, not of carelessness.

Since then, usage has broadened. Today "technical debt" covers any internal-quality shortfall that makes future change slower or riskier, whether it arose from learning, deliberate shortcuts, ignorance or plain neglect. The broader meaning is useful as long as you remember that the different kinds call for different responses.

Principal and interest

The financial vocabulary is what makes the metaphor powerful in conversations with non-engineers. The principal is the effort needed to fix the problem: to restructure the module, separate the database, remove the cycle. The interest is the extra effort paid on every piece of work that touches the problem while it remains. Martin Fowler's illustration: if a feature would take four days in clean code but takes six because of cruft, the two extra days are interest.

Three consequences follow. First, **debt that is never touched costs nothing**: a messy module that nobody changes accrues no interest, so repaying it may be wasted effort. Second, **interest compounds**: workarounds become foundations for more workarounds, new developers copy the patterns they see, and the cost of the eventual repair rises. Third, **repayment is an investment decision**: it is worth paying the principal when the interest you will save over a sensible horizon exceeds it. That turns an emotional argument ("the code is a mess") into an economic one ("this module costs us eleven days a quarter").

IF YOU ARE NEW TO THIS

Step by step: when a mess costs money and when it does not

The idea that needs the most care here is that technical debt is judged by the interest it charges, not by how messy the code looks. The principal is paid once, to fix the problem; the interest is paid on every change that has to work around it. Two equally messy modules show why that matters.

- Take Martin Fowler's figures: a change that would take four days in clean code takes six in a messy module, because developers must first understand the mess and work around it. The two extra days are interest.
- Suppose tidying the module, the principal, would take ten developer-days and bring each change back to about four days. Whether that is worth doing depends on how often the module changes.
- If the team changes it about once a month, it pays about two days of interest a month, so ten days of tidying pays for itself in about five months. Left alone, the interest tends to grow, as new workarounds are built on old ones.
- A second module, just as messy, is changed only about twice a year. It charges about four days of interest a year, so the same ten days of tidying would take about two and a half years to pay back.
- A module that nobody changes, and that causes no failures, costs almost nothing in interest, however messy it looks. Tidying it may be wasted effort.
- **The lesson:** ask not 'how messy is this code?' but 'how much extra does each change cost, and how often do we change it?' Repay first where the interest is highest. The worked example at the end of this section makes the same calculation from ticket and version-control evidence.

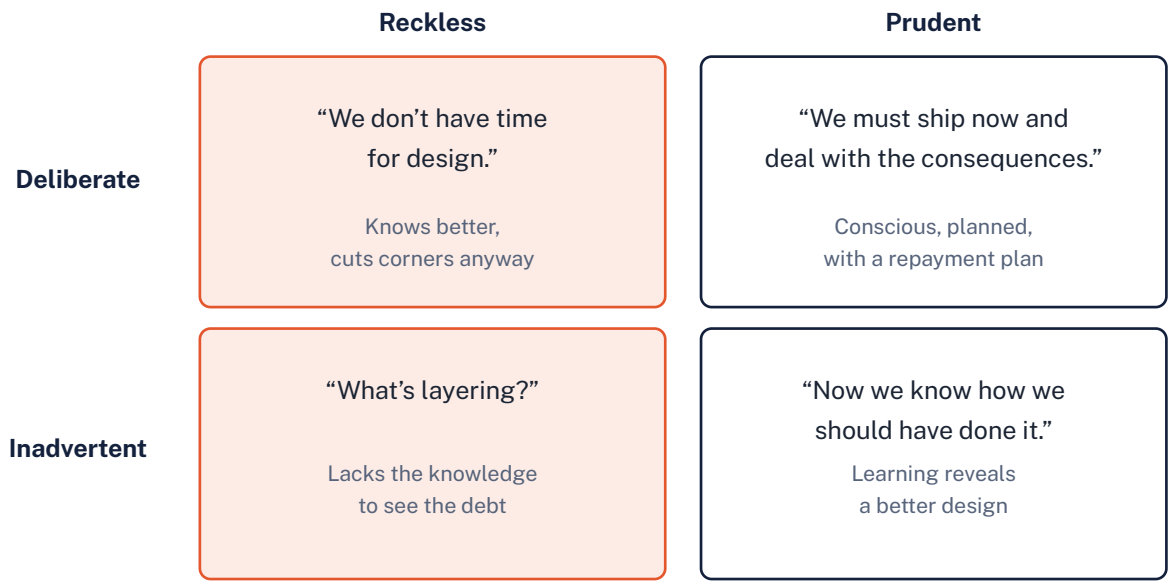
Fowler's technical debt quadrant

Not all debt is equal. In 2009 Martin Fowler proposed classifying debt along two axes: whether it was taken on **deliberately** or **inadvertently**, and whether the decision was **prudent** or **reckless**. The four combinations call for different responses.

Prudent and deliberate: the team knows the better design, judges that shipping now is worth the cost, and plans to repay. This is Cunningham's original sense, and it can be good business: a start-up proving demand before investing in scale. **Reckless and deliberate**: the team knows better but skips design because it believes it has no time, usually slowing itself down within weeks. **Reckless and inadvertent**: the team does not know enough to see that it is creating debt; the cure is skills, review and mentoring rather than refactoring alone. **Prudent and**

inadvertent: a skilled team does good work and only later understands how it should have been structured. Fowler argues this is unavoidable for any team that learns, which is why continuous refactoring is part of normal work rather than a sign of failure.

The technical debt quadrant



After Martin Fowler (2009): the two reckless quadrants are problems of discipline or skill; the two prudent ones are normal consequences of delivering under uncertainty.

Architectural debt versus code debt

Debt exists at every level, but its cost profile changes with scale. **Code debt** lives inside a module: duplicated logic, a function that does five things, missing tests, poor names. It is real, but it is local; one developer can usually repay it in hours or days, often while working on a nearby feature. **Architectural debt** lives in the structure: services that share a database so neither can change its schema alone, a domain split along the wrong lines so every feature touches four teams, a synchronous call chain that makes one slow service take down the checkout, a framework welded into every layer. Repaying it touches many modules, often many teams, and sometimes data.

Research supports the intuition that structural debt is the expensive kind. A 2015 survey of 1,831 engineers and architects by Neil Ernst and colleagues at the Software Engineering Institute found that architectural decisions were the most important source of technical debt reported, ahead of code-level issues. Tools that flag code smells are good at finding code debt and poor at finding architectural debt, because the problem is in relationships between parts rather than in any one file.

Architectural debt matters for the same reason architecture matters at all: as Why architecture matters showed, structural decisions are the ones with the highest cost of change.

Erosion and drift

Architecture debt often accumulates not through one bad decision but through thousands of small changes. Dewayne Perry and Alexander Wolf, in their 1992 paper "Foundations for the Study of Software Architecture", named two ways this happens. **Architecture erosion** is caused by *violations* of the architecture: a controller that queries the database directly, bypassing the service layer; a module that reaches into another's internal tables. Each violation makes the system more brittle, like removing a load-bearing wall. **Architecture drift** is caused by *insensitivity* to the architecture: changes that break no explicit rule but ignore the intended concepts, so the system gradually loses coherence and clarity of form. Drift is less immediately dangerous than erosion, but it obscures the architecture, which in turn makes violations easier.

Later researchers often use "erosion" for the broader gap between the intended architecture (what the documents and the architect's head say) and the implemented architecture (what the code actually does). Whatever the terms, the causes are familiar:

Deadline pressure that rewards the shortest path over the right path. **Knowledge loss** when the people who understood the design leave and new staff copy whatever they find. **Invisible architecture**: rules that live only in a wiki nobody reads, with nothing in the build to enforce them. **Changed requirements**, which leave a once-good structure fitting the problem badly. **Organisational change**: by Conway's law, a reorganisation creates pressure for the system to reshape itself along new team lines. **Ageing technology**: frameworks and platforms that fall out of support turn into debt without anyone touching the code.

Lehman's laws: why this keeps happening

M. M. Lehman began studying how large programs change through work on IBM's OS/360 process in the late 1960s, and refined his findings into eight "laws" by the 1990s. They apply to E-type systems: software embedded in the real world, which changes the world it serves. Three matter most here. **Continuing change**: such a system must be continually adapted, or it becomes progressively less satisfactory. **Increasing complexity**: as it evolves its complexity increases, unless work is done to maintain or reduce it. **Declining quality**: its perceived quality falls unless it is rigorously maintained. These are empirical observations, not physics; a 1997 study supported some and left others uncertain. The food-delivery platform will decay if left to itself, even with no bad decisions.

The Big Ball of Mud

In 1997, Brian Foote and Joseph Yoder wrote "Big Ball of Mud", naming what they called the most frequently deployed of software architectures: "a casually, even haphazardly, structured system", shaped more by expediency than by design. Their paper is unusual because it asks why mud persists instead of merely condemning it. Deadlines reward working software now, architecture is largely invisible, early understanding of the domain is thin, and throwaway code is rarely thrown away. Messy systems also work, and people who can navigate them become valuable. So they treat it as a pattern with forces behind it: piecemeal growth needs steady repair, a mess can be swept under the rug behind a clean interface, and Section 11.6 covers what follows when it cannot.

Detecting debt

Debt you cannot see cannot be managed, and gut feeling is unreliable: developers tend to complain loudest about the code they touch most, which is useful, but they rarely know how much each problem costs. Several techniques make debt visible.

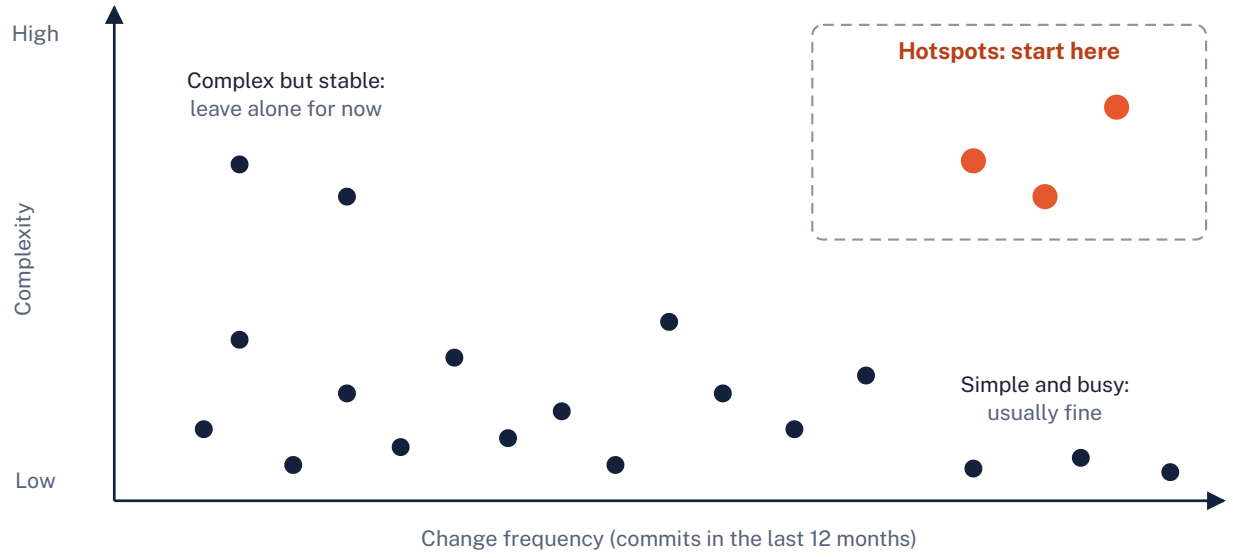
Hotspots. Adam Tornhill's *Your Code as a Crime Scene* (2015; second edition 2024) applies forensic ideas to version-control history. A hotspot is code that is both complex and changed often. Complexity alone is not enough: a complex file nobody touches costs no interest. Change frequency alone is not enough: a simple configuration file can change daily without harm. Where the two coincide, debt is being paid on every commit, and that is where repayment pays off first. In most codebases a small fraction of files accounts for most of the change activity.

Change coupling. Files or services that keep changing in the same commits are coupled, whether or not the code shows a dependency. If every change to the pricing service is accompanied by a change to the invoicing service, they are one thing pretending to be two. Change coupling across team or service boundaries is a strong sign of architectural debt.

Dependency cycles and rule violations. Static analysis can draw the real dependency graph and compare it with the intended one. Cycles (A uses B, B uses C, C uses A) mean none of the three can be understood, tested or released alone. Architecture rules ("the domain layer must not depend on the web layer") can be checked automatically on every build; these are fitness functions, covered in Chapter 11.

Delivery trends. Rising lead time for changes, more defects escaping per release, longer onboarding, more coordination meetings per feature and growing reluctance to touch certain areas are lagging indicators of accumulated interest. The DORA delivery metrics (such as change lead time and change fail rate) give a consistent way to track some of these over time.

Finding hotspots



Plotting each file by how often it changes and how complex it is shows where debt is being paid most often: the few files in the top-right corner.

Managing debt

No healthy codebase has zero debt, and trying to reach zero is itself a waste. The goal is to keep debt visible, deliberate and affordable. Teams that do this well combine several practices.

A debt register. Record significant debt items where the team plans its work: what the problem is, where it is, how it arose, the estimated principal, the estimated interest (with evidence such as hotspot data or recent tickets), and the trigger for repaying it. A register turns vague complaints into a list that can be prioritised alongside features.

A fixed share of capacity. Many teams reserve a steady proportion of each iteration, commonly somewhere between 10 and 20 per cent, for debt repayment and other improvement work. There is no evidence for a magic number; what matters is that the share is protected and does not disappear whenever a deadline looms.

Refactoring alongside features. The cheapest time to repay debt is when you are already working in the area, because you have the context loaded and the tests running. Fowler calls this opportunistic refactoring. Its everyday form is the boy scout rule, popularised by Robert C. Martin: leave the code a little cleaner than you found it. Because hotspots are by definition where features land, this naturally concentrates effort where interest is highest.

Planned structural repayment. Architectural debt rarely yields to small steps alone. It needs an explicit, funded piece of work with a goal, a plan for migrating in increments, and the support of stakeholders. Techniques such as the strangler fig pattern (Chapter 11, Modernising

legacy systems) let a team replace a structure piece by piece without a big-bang rewrite.

Prevention. Automated architecture checks in the build, code review that looks at structure as well as style, and up-to-date decision records slow erosion and drift at source.

AI-assisted development: faster code, faster erosion?

Coding assistants and agents (Section 12.4) can now produce code faster than a team can read it. Google's 2025 DORA report, from nearly 5,000 technology professionals, found 90 per cent using AI at work. Adoption was associated with higher delivery throughput but lower delivery stability, and DORA's main conclusion was that AI is an amplifier: it magnifies a team's existing strengths and weaknesses, and loosely coupled systems with fast feedback gain most. Those are survey correlations. Controlled evidence is thinner and moves quickly: METR's trial of early-2025 tools found experienced developers 19 per cent slower while they believed they were faster, and its 2026 follow-up suggested later tools probably help, while calling its own data very weak evidence of how much. Measure in your own setting, and check the current research.

The architectural risk is erosion at speed. A tool works on the task in front of it, with limited sight of the intended architecture, and it tends to imitate what is nearby. Ask an agent to add a refund reason to the food-delivery platform and it may query the Payments tables directly, because that compiles and the tests pass: a Perry and Wolf violation, produced in seconds. GitClear, a vendor of code analytics, reported cloned lines rising from 8.3 to 12.3 per cent of changed lines between 2021 and 2024, and refactoring falling from 25 to under 10 per cent. That shows a trend, not proof of cause. A quieter cost is understanding: code can reach production that nobody on the team could explain.

The architect's work therefore shifts from producing structure towards stating and defending it. What used to live in people's heads has to become explicit: clear module boundaries, decision records and repository instructions that tools and people can read. What used to rely on vigilance has to become automatic: dependency rules and fitness functions in the build (Section 11.3). Well-placed boundaries also limit the damage any one change can do.

Governance of generated code follows from this. **Accountability**: a named person owns every change and reviews it to the usual standard, and large generated changes are split until they can be reviewed. **Automatic gates**: architecture checks, static analysis, secret scanning and tests apply whoever or whatever wrote the code. **Dependencies**: a USENIX Security 2025 study of 576,000 samples found models inventing package names, on average at least 5.2 per cent for commercial models and 21.7 per cent for open-source ones, and an attacker can register such a name; restrict installs as in Section 8.6. **Security**: in a Stanford-led study using an early model, participants with an assistant wrote less secure code and were more likely to

believe it secure. **Permissions**: agents get least privilege and no production credentials. **Policy**: record which tools are approved and what code or data may be sent to them, and take advice on licensing and confidentiality.

Explaining debt to non-technical stakeholders

Sponsors and product owners do not need to hear about cyclic dependencies; they need to understand the effect on what they care about: speed, cost, risk and customer experience. The financial metaphor is a gift here, provided you use it with numbers.

Talk in **time and money**: "Every change to checkout costs us about three extra developer-days because of how payment and orders are tangled together." Talk in **risk**: "Two of our last four incidents started in this module." Talk in **options**: "Until we separate these, we cannot offer the new payment methods sales has asked for." And present repayment as an **investment with a payback period**, not as a plea for engineering time. Avoid two traps: presenting debt as a moral failing of the people who came before (it is usually prudent or inadvertent), and asking for an open-ended "refactoring sprint" with no measurable outcome.

WORKED EXAMPLE

Worked example: estimating the interest on a debt

An online shop's order and payment logic share one large module and the same database tables. The team suspects this costs them a lot. They gather evidence from the last six months of tickets and version-control history, then estimate.

Changes touching the module	48 changes in six months, about 8 per month (it is the top hotspot).
Extra effort per change	Comparing similar changes elsewhere, developers estimate about 1.5 extra days each for understanding, working around the shared tables and extra regression testing.
Effort interest	8 changes × 1.5 days = 12 developer-days per month.
Defect interest	3 production incidents in six months traced to this module, each costing about 4 developer-days to investigate and fix: 12 days ÷ 6 = 2 days per month.
Total interest	About 14 developer-days per month. At a loaded cost of £600 per developer-day, roughly £8,400 per month, or about £100,000 per year.
Principal	Separating orders from payments behind an interface and splitting the tables, done incrementally: estimated 45 developer-days, about £27,000.
Residual interest after repayment	Developers expect some overhead to remain: estimated 3 developer-days per month.
Payback	Monthly saving about 11 developer-days. 45 ÷ 11 is about 4 months to recover the principal.
Recommendation	Repay, in increments of about two weeks each, starting with the tables most often involved in incidents. Track lead time for changes to checkout to confirm the saving.

Every number is an estimate with wide error bars. Doubling the principal to 90 days still gives payback in about eight months, so the conclusion is robust; that check is worth showing stakeholders.

The same method can show that some debt is not worth repaying: a messy module changed twice a year may have a payback period of several years.

The case is made in the sponsor's terms (days, money, incidents, payback), not in terms of code quality.

WATCH OUT

Common misconceptions

- "Technical debt means bad code." Much debt comes from good code written with understanding that has since improved, or from requirements that have moved.
- "All debt must be repaid." Debt in code that never changes costs nothing; repay where the interest is.
- "We'll fix it in a big rewrite." Big-bang rewrites routinely overrun and lose hidden behaviour; incremental repayment is usually safer.
- "Only developers care about debt." Its effects (slower features, more incidents, blocked opportunities) are business effects, and sponsors should help decide what to repay.

CHECK YOUR UNDERSTANDING

▼ Why can a very messy module be a low priority for repayment?

Because interest is paid only when the code is changed or causes failures. If nobody touches it and it is stable, the debt costs almost nothing.

▼ Why is reckless, inadvertent debt not cured by refactoring alone?

Because the team did not know it was creating debt; unless its skills and review practices improve, it will keep creating the same kind of debt as fast as it repays it.

▼ What is the difference between erosion and drift in Perry and Wolf's terms?

Erosion results from violating the architecture's rules; drift results from changes made without regard to the architecture that break no explicit rule but blur its concepts.

▼ Why do hotspot analyses combine change frequency with complexity rather than use either alone?

Complex code that never changes accrues little interest, and simple code that changes often is cheap to change; only the combination shows where debt is costing effort repeatedly.

▼ Try it: A messy module is changed 6 times a month, each change costing 1 extra day at £600 a day; fixing it takes 60 days. Another, equally messy, has not been touched for two years. Which debt would you repay?

Repay the first. Its interest is 6 x £600 = £3,600 a month, and its principal is 60 x £600 = £36,000, so the fix pays for itself in about 10 months. The second carries principal but no interest: code that never changes costs nothing extra, so leave it and note it in the debt register. If it starts changing often, look again. Show the numbers to the business, not the mess.

SOURCES

- The WyCash Portfolio Management System, Ward Cunningham (OOPSLA 1992): https://c2.com/doc/oopsla92.html
- Technical Debt, Martin Fowler: https://martinfowler.com/bliki/TechnicalDebt.html
- Technical Debt Quadrant, Martin Fowler (2009): https://martinfowler.com/bliki/TechnicalDebtQuadrant.html
- Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt (SEI, 2015): https://sei.cmu.edu/library/measure-it-manage-it-ignore-it-software-practitioners-and-technical-debt
- Ward Explains Debt Metaphor (c2 wiki): https://wiki.c2.com/?WardExplainsDebtMetaphor
- Lehman: Laws of Software Evolution Revisited: https://www.cs.kent.edu/~jmaletic/cs63902/Papers/Lehman96.pdf
- Lehman et al.: Metrics and Laws of Software Evolution, The Nineties View: https://www.ifi.uzh.ch/dam/jcr:00000000-5852-3622-0000-00002e60ff8e/lehman-metrics-97.pdf
- Foote and Yoder: Big Ball of Mud: http://laputan.org/mud/
- Google Cloud: Announcing the 2025 DORA report: https://cloud.google.com/blog/products/ai-machine-learning/announcing-the-2025-dora-report
- METR: Measuring the impact of early-2025 AI on experienced open-source developer productivity: https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/
- METR: developer productivity experiment update (February 2026): https://metr.org/blog/2026-02-24-uplift-update/
- GitClear: AI Copilot Code Quality 2025: https://www.gitclear.com/ai_assistant_code_quality_2025_research
- Spracklen et al.: We Have a Package for You!: https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen

- Perry et al.: Do Users Write More Insecure Code with AI Assistants?: https://arxiv.org/abs/2211.03622

SECTION 1.8 OF 1.8 · INTERMEDIATE · 21 MIN READ · LAST REVIEWED NOVEMBER 2026

Architecture across the lifecycle: up-front, emergent and continuous architecture

BEFORE YOU READ THIS

This section builds on:

- 0.8 How software gets built
- 1.4 The architect's role
- 1.6 Decisions and trade-offs

Terms it uses from earlier: Agile, Continuous delivery, Last responsible moment, Fitness function, Quality attribute

THE IDEA IN BRIEF

When you plan a long trip, you book the flights and the first night's hotel well in advance, because those are expensive to change and might sell out. You do not plan every meal of every day before you leave; you decide those as you go, once you know how you feel and what is nearby. Good software architecture works the same way. The few big, hard-to-change decisions are made early and carefully. Everything else is decided along the way, as the team learns, and the plan is checked and adjusted throughout the journey rather than written once and filed.

WHAT YOU NEED TO KNOW

- Big design up front suits stable requirements and very costly change; purely emergent design suits small systems of two-way doors; most teams need something in between.
- The Agile Manifesto principle that the best architectures emerge from self-organising teams is about who designs and how, not a claim that architecture is unnecessary.
- Fairbanks' risk-driven model sets architecture effort in proportion to the risk of failure: identify and prioritise risks, apply techniques, evaluate the reduction, and stop when the remaining risk is acceptable.
- SAFe's architectural runway and Erder and Pureur's continuous architecture both treat architecture as ongoing work that must stay ahead of the features that depend on it.
- A walking skeleton proves the architecture end to end early; spikes turn uncertain decisions into evidence-based ones.

- In continuous delivery and DevOps, deployability and operability become key quality attributes, loosely coupled architecture enables delivery performance, and the running system becomes the main source of architectural feedback.

The old argument: up front or emergent?

For most of software's history, architecture was something you did at the start. In a sequential ("waterfall") process, requirements were gathered, then an architecture and detailed design were produced and approved, then the system was built to that design. This is big design up front (BDUF). Its logic is sound in some settings: when requirements are stable and well understood, when changes after delivery are extremely expensive (flight software, medical devices, hardware-bound systems), or when many contractors must build parts that fit together, a thorough design before construction reduces risk.

Its weaknesses are equally well known. Requirements change, and an elaborate design made early encodes assumptions that later prove wrong. Feedback from real use arrives only at the end, when change is most expensive. Teams design for imagined needs ("we might need to support five currencies") and pay for flexibility nobody uses. And documents that took months to approve tend to be followed after they have stopped being true.

Agile methods reacted against this. Extreme Programming in particular argued for emergent design: build the simplest thing that works for the features in hand, keep the code clean with continuous refactoring and automated tests, and let the design grow as the system does. Kent Beck's rules of simple design and the principle "you aren't gonna need it" (YAGNI) express the same idea. Emergent design works well inside a module, and for small systems where every decision is a two-way door. Its weakness appears at scale: some decisions (the data model of a system of record, the boundaries between teams, the security model, the choice between synchronous and asynchronous integration) are expensive to reverse and shape everything built on top. Leaving them to emerge often means they are made by accident by whoever writes the first code.

What the Agile Manifesto does and does not say

One of the twelve principles behind the Agile Manifesto (2001) reads: “The best architectures, requirements, and designs emerge from self-organizing teams.” It is frequently quoted as if it meant that agile teams do not need architecture or architects. It does not say that.

Read carefully, the principle makes a claim about *who* produces the best architectures (the people who build the system, organised by themselves rather than directed from outside) and *how* (they emerge through the work, rather than being handed down complete). It says nothing

about architecture being unimportant, and the same document contains another principle: continuous attention to technical excellence and good design enhances agility. A team that ignores structural decisions is not following the manifesto; it is ignoring half of it.

What the principle does rule out is an architecture imposed from an ivory tower by people who will not live with its consequences, and a phase in which the architecture is finished and frozen before anyone writes code. In practice, the most common reconciliation is that architecture becomes a continuous activity, shared within the team, with an architect (or architecturally minded engineers) acting as guide, facilitator and keeper of the big picture rather than as a gatekeeper.

A spectrum, not a choice



Most successful teams sit between the extremes: they decide the expensive-to-reverse questions early and let the rest emerge.

How much architecture? The risk-driven model

If neither extreme is right, how much architecture work is enough? George Fairbanks gave one of the clearest answers in *Just Enough Software Architecture: A Risk-Driven Approach* (2010). His risk-driven model says the effort you spend on architecture should be proportionate to the risk of failure. It has three steps, repeated as needed: **identify and prioritise risks** (ideally as concrete failure scenarios, such as "the booking page times out when 5,000 patients log in at 8 a.m."); **select and apply techniques** that reduce those risks (a model, a prototype, a load test, an analysis, a design pattern); and **evaluate the risk reduction**, stopping when the remaining risk is acceptable.

The model is powerful because it gives a stopping rule. A simple internal tool built on familiar technology with few users faces few architectural risks, and a sketch on a whiteboard may be enough. A payment platform handling millions of transactions, subject to regulation, faces many, and deserves careful modelling, prototyping and review. The same team may apply very

different amounts of architecture to different parts of one system. Fairbanks also stresses that time spent designing is time not spent building or testing, so over-designing is a real cost, not a safe default.

IF YOU ARE NEW TO THIS

Step by step: just enough architecture for the booking system

The step most people trip over here is Fairbanks' risk-driven model: there is no fixed right amount of architecture, only an amount that matches the risks. You keep designing while the risk of failure is too high, and stop when it is acceptable. Follow it through the hospital booking system.

- List the risks as concrete failures, such as 'the booking page times out when 5,000 patients log in at 8 a.m.' and 'the patient administration system, which supplies the clinics' free slots, cannot answer 50 requests a second'. Parts built on familiar technology for a few users raise few serious risks.
- Prioritise. The 8 a.m. rush would hit every patient booking then, so it goes to the top. The low-risk parts get a whiteboard sketch and no more.
- Apply a technique that fits each risk. For the morning rush, the team builds a small prototype and tests it with 5,000 simulated patients logging in together. For the patient administration system, a short experiment, agreed with its supplier, measures how many requests a second it can answer.
- Evaluate how far each risk has shrunk. Suppose the prototype copes with the rush, but the patient administration system slows badly before 50 requests a second. The first risk is now acceptable; the second is not, so the team goes round again, perhaps testing the short-lived slot cache from Section 1.5.
- Stop when every top risk is acceptably low. Further design is not a safe default, Fairbanks warns, because it takes time from building and testing.
- **The lesson:** let the risks set the effort. One part of a system may get a whiteboard sketch and another weeks of prototyping and testing: that is the model working.

Intentional architecture and the architectural runway

Large organisations face a coordination problem that a single team does not: dozens of teams must build parts that fit together. The Scaled Agile Framework (SAFe), a widely adopted and also widely debated framework for scaling agile practices, describes architecture as a balance between *emergent design* within teams and *intentional architecture*: deliberate, cross-team planning of the structures and guidelines that teams share. It calls the result the architectural runway: the existing code, components and technical infrastructure needed to implement near-

term features without excessive redesign and delay. Work to extend the runway (for example, introducing a messaging platform before the features that need it) is planned explicitly alongside feature work, and features consume the runway as they use it.

Described neutrally, the runway metaphor captures something true for any organisation: if the infrastructure a feature needs does not yet exist, that feature will be slow, and building the infrastructure takes lead time. Critics of SAFe argue that it can reintroduce heavyweight, top-down planning under agile vocabulary; supporters argue that some central coordination is unavoidable at scale. You do not need to adopt SAFe to use the idea: "what foundations will next quarter's features need, and are we building them in time?" is a good question for any architect.

Continuous architecture

Murat Erder and Pierre Pureur, in *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World* (2015) and *Continuous Architecture in Practice* (2021, with Eoin Woods), describe architecture as an ongoing flow of decisions rather than a phase or a document. Their six principles are: architect products, not just solutions for projects; focus on quality attributes, not on functional requirements; delay design decisions until they are absolutely necessary; architect for change by leveraging the power of small, loosely coupled parts; architect for build, test, deploy and operate; and model the organisation of your teams after the design of the system you are working on.

Two of those principles echo earlier sections of this chapter: delaying decisions is the last responsible moment, and modelling teams after the system is a deliberate use of Conway's law. The practical emphasis is on the unit of architectural work being the *decision*, recorded as it is made, and on feedback loops (automated tests, fitness functions, production monitoring) that tell the team whether the architecture still meets its quality goals.

Building evolutionary architectures

A related strand comes from Neal Ford, Rebecca Parsons and Patrick Kua's *Building Evolutionary Architectures* (2017; second edition 2022 with Pramod Sadalage). Long-lived systems must keep pace with their environment, so instead of predicting the future, the authors design for change: many small steps, across both technical and business dimensions, with what "fit" means made explicit and checked automatically wherever possible. If the food-delivery platform promises that checkout stays fast, an automated test can guard that promise on every release while the code beneath it changes. Section 11.3 covers these checks, called fitness functions, and the honest limits of the approach.

Walking skeletons and spikes

Two simple techniques bring architecture and delivery together from the first week. A walking skeleton, a term associated with Alistair Cockburn, is a tiny implementation of the system that performs one small end-to-end function through all the main architectural components. For a hospital booking system, it might let a test user log in, see one hard-coded clinic's free slots fetched from the patient administration system, book one, and receive a text message, deployed to a production-like environment through the real delivery pipeline. It does almost nothing useful, but it proves that the pieces connect, that the pipeline works, and that the architecture can be built. Every later feature adds flesh to a skeleton that already walks.

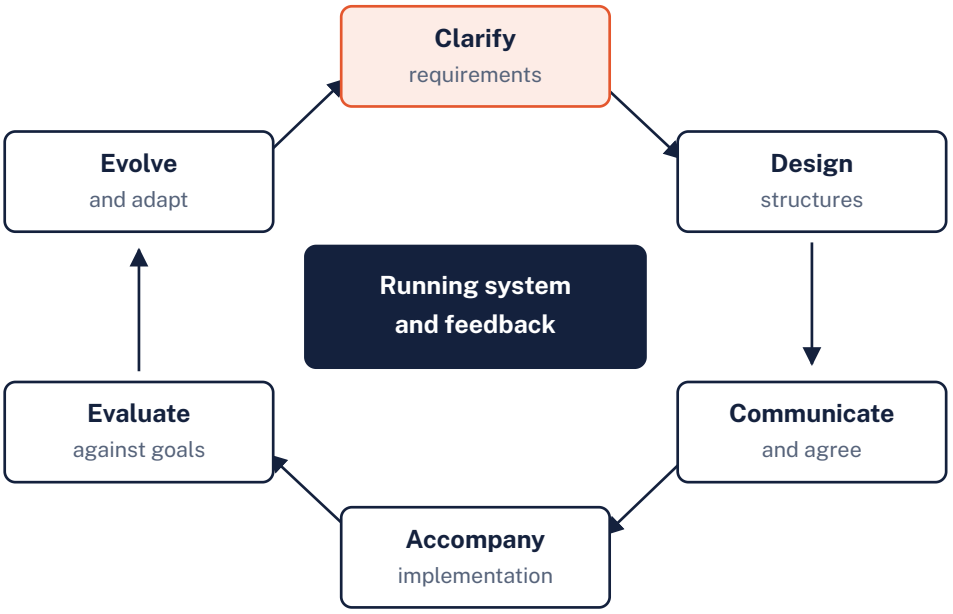
A spike (another term from Extreme Programming) is a short, time-boxed experiment to answer one question or reduce one risk: can the PAS interface handle 50 requests a second? How long does the identity provider take to issue a token? Can our chosen library generate accessible PDFs? Spikes are usually thrown away; their output is knowledge, recorded in a decision. They are the cheapest way to move a decision from guesswork to evidence before its last responsible moment.

Architecture activities in an iterative cycle

In an iterative process, architecture work does not happen once; it recurs in every cycle, at a depth that matches the risks in front of the team. The iSAQB curriculum describes the architect's core tasks in a way that maps naturally onto such a cycle, and The architect's role introduced them. Here they are as a loop:

Clarify requirements: find the stakeholders and concerns, sharpen the quality goals and constraints, and identify the risks (see Stakeholders and concerns). **Design structures**: make or revise the significant decisions, choosing styles and patterns and defining boundaries and interfaces. **Communicate and agree**: explain the decisions to the people who build, run and pay for the system, in views suited to each audience, and record them. **Accompany implementation**: work with the teams as they build, answer questions, review code and adjust the design when reality pushes back. **Evaluate**: check that the system meets its quality goals, through reviews, tests, measurements in production and automated fitness functions. **Evolve**: act on what evaluation found, repay debt, and adapt to changed requirements, which brings you back to clarifying requirements.

The architecture cycle



Each iteration revisits all six activities, with depth proportional to the risks in front of the team, around a running system that provides feedback.

Architecture in DevOps and continuous delivery

Continuous delivery, in which every change is built, tested and kept releasable, and DevOps, which joins development and operations responsibility, change both the goals of architecture and its feedback loops.

They change the goals because **deployability, testability and operability become first-class quality attributes**. A system cannot be released many times a day if components must be deployed together in a fixed order, if tests need a full copy of production, or if nobody can tell from the monitoring whether a release made things worse. The research summarised in *Accelerate* (Nicole Forsgren, Jez Humble and Gene Kim, 2018) found that a loosely coupled architecture, one that lets teams test and deploy their parts independently, was among the strongest predictors of continuous delivery performance. Architecture is therefore not just helped by good delivery practice; it is one of its preconditions.

They change the feedback loops because the running system becomes the most honest architecture review. Production metrics, traces and incident reviews show where the architecture is weak far more reliably than diagrams do. Architectural rules can be enforced as automated fitness functions in the pipeline (Chapter 11), infrastructure is defined as code and reviewed like code (Chapter 10), and techniques such as feature flags and canary releases

make many decisions cheaply reversible. The architect's role shifts accordingly: less approval of documents, more shaping of the platform, pipeline and guardrails that let teams make good decisions quickly.

Checklist: is your architecture work proportionate?

Run through this list at the start of a project and at each major milestone. Each item is a question about whether the effort matches the risk.

- [ ] **You can name the top five architectural risks**
  Written as concrete failure scenarios, prioritised, and each with an owner and a plan to reduce it.
- [ ] **Your design effort is concentrated on those risks**
  Areas with low risk and familiar technology get a sketch; areas with high risk get prototypes, measurements and review.
- [ ] **One-way doors are identified and decided deliberately**
  Expensive-to-reverse decisions are known, made at the last responsible moment and recorded; two-way doors are left to teams.
- [ ] **A walking skeleton exists early**
  An end-to-end path through the main components runs in a production-like environment within the first few iterations.
- [ ] **Decisions are recorded briefly as they are made**
  A short decision record exists for each significant decision; nobody has to reconstruct the reasoning from memory.
- [ ] **Quality goals are checked continuously**
  Key quality attributes have automated tests, fitness functions or production measurements, not just a one-off review.
- [ ] **Architecture is not blocking delivery**
  Teams rarely wait for an architecture decision or approval; if they do, the process is too heavy for the risk.
- [ ] **Delivery is not undermining architecture**
  Hotspots, change coupling and rule violations are tracked; erosion is noticed within weeks, not years.
- [ ] **The amount of documentation matches its readers**
  Each document or view answers a real stakeholder's concern and is kept current; unread documents are retired.

WATCH OUT

Two opposite failure modes

Architecture astronautics: months of diagrams, frameworks built for needs that never arrive, and a design that is obsolete before the first release. **Accidental architecture**: no one considers the structure until the system is large, slow to change and impossible to reason about, and a costly rescue becomes unavoidable. The cure for the first is feedback from running code; the cure for the second is deliberate attention to the few decisions that matter most. The risk-driven model is a way to steer between them.

CHECK YOUR UNDERSTANDING

▼ **When is big design up front a sensible choice?**

When requirements are stable and well understood, when change after delivery is extremely expensive, or when many separate parties must build parts that fit together exactly.

▼ **What stopping rule does the risk-driven model give for architecture work?**

Stop when the remaining risks are acceptably low; effort should be proportionate to the risk of failure.

▼ **What is the difference between a walking skeleton and a spike?**

A walking skeleton is a minimal end-to-end implementation through the real architecture that is kept and built upon. A spike is a short experiment to answer one question, usually thrown away once it has produced knowledge.

▼ **Why does continuous delivery make architecture more important rather than less?**

Frequent independent releases require a loosely coupled, testable and deployable structure; without it, the delivery pipeline cannot work at speed.

▼ **Try it: A new ticketing system has a known payment provider but nobody knows whether seat holds will survive 2,500 visitors a second. How much up-front design would you do, and what would you build first?**

Follow the risk. Use light design where things are well understood, such as the payment integration, and spend the effort on the biggest risk, seat holds under load. Build a walking skeleton through the whole purchase path early, and run a spike that load-tests the seat-hold approach, so the decision rests on evidence. Stop analysing when the remaining risk is acceptable. A full design for everything, or none at all, would both be poor fits.

SOURCES

- Principles behind the Agile Manifesto: https://agilemanifesto.org/principles.html
- A Risk-Driven Model for Agile Software Architecture, George Fairbanks (Methods and Tools): https://www.methodsandtools.com/archive/agilearchi.html
- Is Design Dead?, Martin Fowler: https://martinfowler.com/articles/designDead.html
- Architectural Runway (SAFe): https://framework.scaledagile.com/architectural-runway
- Continuous Architecture (Erder, Pureur and Woods): https://continuousarchitecture.com/
- Evolutionary Architecture (authors' site): https://evolutionaryarchitecture.com/
- O'Reilly: Building Evolutionary Architectures, 2nd edition: https://www.oreilly.com/library/view/building-evolutionary-architectures/9781492097532/

Further reading

- *Software Architecture in Practice, 4th edition*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. The standard textbook from the SEI: structures, quality attributes and tactics, and the definition used throughout this course.
  https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- *Fundamentals of Software Architecture, 2nd edition*, Mark Richards and Neal Ford (2025), O'Reilly Media. A practical, readable introduction to the architect's role, trade-off thinking and architectural styles.
- *ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description*, ISO/IEC JTC 1/SC 7 and IEEE (2022), ISO. The international standard for architecture descriptions, stakeholders, concerns and views.
  https://www.iso.org/standard/74393.html
- *Documenting Software Architectures: Views and Beyond, 2nd edition*, Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord and Judith Stafford (2010), Addison-Wesley. The source of the module, component-and-connector and allocation categories of structure.
  https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- *Who Needs an Architect?*, Martin Fowler (2003), IEEE Software. A short, influential essay on what architecture is and what kind of architect a team needs.
  https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- *Just Enough Software Architecture: A Risk-Driven Approach*, George Fairbanks (2010), Marshall & Brainerd. Explains how much architecture work to do, driven by the risks a project actually faces.
  https://www.georgefairbanks.com/book/
- *The Software Architect Elevator*, Gregor Hohpe (2020), O'Reilly Media. On the architect's role in large organisations, connecting business strategy and technical delivery.
- *The WyCash Portfolio Management System (OOPSLA '92 experience report)*, Ward Cunningham (1992), OOPSLA 1992. The short report in which the technical debt metaphor first appeared.
  https://c2.com/doc/oopsla92.html

Glossary

Architectural debt

Technical debt in the structure of a system, such as wrong boundaries, a shared database or cyclic dependencies, which is costly to repay because it touches many parts.

Architectural runway
In the Scaled Agile Framework, the existing code, components and infrastructure needed to implement near-term features without excessive redesign and delay.

Architecturally significant
Describes a decision or requirement that is expensive to change later, affects many parts of a system, or determines a key quality attribute.

Architecture Decision Record (ADR)
A short document recording one significant decision: its context, the options considered, the decision, its status and its consequences.

Architecture description
A work product (documents, diagrams, models, decision records) that expresses an architecture. The architecture exists whether or not it is described; the description is how people share it.

Architecture drift
Changes made without awareness of the intended architecture which, while not breaking explicit rules, move the system away from it and blur its concepts.

Architecture erosion
The gradual degradation of a system as changes violate its intended architecture, for example by bypassing layers or adding forbidden dependencies.

Big design up front (BDUF)
An approach in which the architecture and design are completed in detail before implementation starts.

Change coupling
Two or more files or components that tend to change in the same commits, revealing a dependency that may not be visible in the code.

Cognitive bias
A systematic error in thinking, such as anchoring or the sunk cost fallacy, that leads people to judge options unfairly.

Component
A unit that exists when the system runs, such as a service, a process or a database, which interacts with other components through connectors.

Concern
An interest a stakeholder has in a system, such as its cost, performance, security, ease of change or legal compliance.

Connector
The runtime mechanism by which components interact: an HTTP call, a message queue, a shared database, an event stream or a function call.

Conway's law
Melvin Conway's 1968 observation that organisations design systems whose structure mirrors the organisation's own communication structure.

Cost of change
The effort, money, time and risk needed to alter a decision once work has been built on top of it.

Cross-cutting concern
A concern that affects many parts of a system at once, such as logging, security, error handling or persistence, and so is usually decided once for the whole system.

Dead code
Code that is no longer meant to run but is still present in the deployed system, where it can be triggered by mistake.

Dependency cycle
A loop in which modules depend on each other directly or indirectly, so none can be understood, tested or released alone.

Detailed design
Decisions about the inside of one part of a system (its classes, functions, algorithms and local data structures) that can be changed without affecting the rest.

Element
A part of a system that an architect reasons about as a unit, such as a module, a service, a database, a queue or a server.

Emergent design
An approach in which the design grows incrementally, shaped by the needs of each new piece of functionality and continuous refactoring.

Enterprise architecture
Architecture at the level of a whole organisation: how its business capabilities, information, applications and technology fit together and change over time.

Feature flag
A configuration switch that turns a piece of behaviour on or off at run time without deploying new code.

Fitness function
An automated check that tells you whether a system still has an architectural property, such as a test that fails when one module depends on another it must not use.

Hotspot
A part of the code that is both complex and frequently changed, found by combining version-control history with a complexity measure; a likely place to repay debt.

Interest (of technical debt)
The extra effort, delay or risk paid on every change because the debt has not been repaid.

Knowledge pyramid
A model with three layers: what you know, what you know you do not know, and what you do not know you do not know. Architects deliberately widen the middle layer.

Last responsible moment
The point at which delaying a decision any further would remove an important option; deciding then uses the most information without causing harm.

Monolith
A system built and deployed as a single unit, even if its code is divided into well-separated modules inside.

Power/interest grid
A two-by-two map placing stakeholders by their power over a project and their interest in it, used to plan how closely to engage each group.

Principal (of technical debt)
The effort needed to fix a piece of technical debt, that is, to bring the code or structure to the state it should be in.

Proof of concept
A small, time-boxed piece of working code built to test whether an idea, technology or design choice works before committing to it.

Quality attribute
A measurable property of a system that says how well it does its job, such as performance, availability, security or modifiability. Chapter 2 covers them in depth.

Refactoring
Changing the internal structure of code without changing its external behaviour, to make it easier to understand and modify.

Relation
A connection between elements, such as one module using another, one service calling another, or a service being deployed on a server.

Reversibility
How cheaply and safely a decision can be undone. Highly reversible decisions can be made quickly; irreversible ones deserve more analysis.

Risk-driven model
George Fairbanks' approach in which the amount of architecture work is set by the risks a project faces: identify risks, apply techniques, evaluate the reduction.

Serverless
A cloud model in which the provider runs your functions or workflows on demand and bills per use, so you do not manage servers yourself.

Software architecture
The fundamental structures of a software system and the significant decisions that shaped them: its main parts, how they relate, their important properties and the principles that guide how it is built and evolves.

Solution architecture
Architecture of a solution to one business problem, often spanning several applications, vendors and teams, such as a new online claims process for an insurer.

Spike
A short, time-boxed investigation or prototype built to answer a technical question or reduce a risk, normally thrown away afterwards.

Staff engineer
A senior individual contributor role, above senior engineer, that carries technical leadership across teams without managing people; principal engineer is usually the next level up.

Stakeholder
A person, team or organisation with an interest in a system, such as users, sponsors, developers, operators, security officers or regulators.

Structure
A set of elements and the relations among them, seen from one angle: for example the code modules and their dependencies, or the running services and how they communicate.

Technical breadth
Knowing that many technologies and approaches exist, what they are good and bad at, and when to use them, even without deep hands-on expertise in each.

Technical debt
The extra future cost of change caused by shortcuts or by design that no longer fits, paid as slower and riskier work until it is repaid by improving the design.

TOGAF
The Open Group Architecture Framework, a widely used framework and method for enterprise architecture published by The Open Group; its current version is the TOGAF Standard, 10th Edition (2022).

Trade-off
A choice in which gaining one benefit means accepting a cost or giving up another benefit, such as faster delivery at the price of higher running cost.

View
A representation of one or more structures of a system, aimed at particular concerns of particular stakeholders, such as a deployment diagram for operations staff.

Viewpoint
A specification of how to build and read one kind of architecture view: which stakeholders and concerns it serves and which modelling conventions it uses.

Walking skeleton
A tiny implementation of the system that performs a small end-to-end function through all the main architectural components, usually built first.

Weighted decision matrix
A table that scores each option against weighted criteria so that the reasoning behind a choice is explicit and open to challenge.