BE SOFTWARE ARCHITECTURE WISE · CHAPTER 4

Documenting and Communicating Architecture

This chapter shows how to describe and communicate a software architecture so that the right people understand it and the description stays true. It covers who documentation is for and how much to write, views and viewpoints from ISO/IEC/IEEE 42010, 4+1 and Rozanski and Woods, the C4 model, the UML diagrams architects still use, Architecture Decision Records, arc42, diagrams as code, and presenting architecture to different audiences. It is practical throughout, using one food-delivery platform as a running example, and is written for developers, architects and technical leads.

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

AT A GLANCE

You will learn to

- Explain who reads architecture documentation, what each reader needs and how to decide what is worth writing down.
- Distinguish stakeholders, concerns, viewpoints, views and model kinds as defined by ISO/IEC/IEEE 42010:2022.
- Compare the 4+1 view model, Rozanski and Woods's viewpoints and perspectives, and the SEI's three view categories.
- Draw system context, container and component diagrams that follow the C4 notation rules.
- Read and draw the UML sequence, state machine, component, deployment, activity and class diagrams architects use.
- Write Architecture Decision Records and manage their lifecycle in a decision log.
- Apply arc42 and diagrams-as-code practices to keep documentation useful and current.
- Tailor an architecture explanation to executives, developers, operations, security staff and auditors.

Before you start

Chapters 1 to 3, especially stakeholders and concerns (Section 1.5) and decisions and trade-offs (Section 1.6).

Time

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

Key terms

Architecture description · View · Viewpoint · Concern · 4+1 view model · C4 model · Container (C4) · UML · Sequence diagram · State machine diagram · Architecture Decision Record · arc42 · Diagrams as code · Docs as code

Why documenting architecture is part of the job

An architecture that exists only in a few people's heads is fragile. Nobody outside the room can review it, the next engineer to join cannot learn it, an auditor cannot check it, and it leaves with the people who hold it. Chapter 1 separated an architecture (what the system actually is) from an architecture description (the documents, diagrams and records through which people share it). This chapter is about the second: how to produce descriptions that people actually read, that stay true as the system changes, and that help decisions get made.

This is a practical skill more than a theoretical one, and it is easy to get wrong in either direction. Teams that write too little lose the reasons behind their structure and keep having the same arguments. Teams that write too much produce wikis nobody trusts. The guiding

principle of the chapter comes from the SEI's *Documenting Software Architectures: Views and Beyond*: write for the reader. Decide who will use each piece of documentation and what they need from it, write that briefly and accurately, and write nothing else.

One running example appears throughout: a food-delivery platform with a customer app, a restaurant app and a courier app, services for ordering, dispatch and notifications, and card payments through an external provider. You will see it as a context diagram, a container diagram, a component diagram, UML sequence and state diagrams, a decision record and a one-page arc42 summary, so you can compare each technique on the same system.

Describing and drawing: Sections 4.1 to 4.4

The first four sections cover *what* to describe and *how to draw it*. Section 4.1 asks who documentation is for, what the Agile Manifesto does and does not say about it, how to decide what deserves writing down and which anti-patterns to avoid. The short answer is to write by hand what the code cannot tell you (context, quality goals, decisions and their reasons) and to generate what the code already states.

Section 4.2 introduces the central idea that one diagram is never enough. An architecture is described through several views, each answering the concerns of particular stakeholders and each built according to a viewpoint. The section covers the ISO/IEC/IEEE 42010:2022 conceptual model, Kruchten's 4+1 view model, Rozanski and Woods's viewpoints and perspectives, and the SEI's module, component-and-connector and allocation categories, and shows how to choose the minimum set of views a system needs.

Section 4.3 turns that theory into the most widely used practical approach, Simon Brown's C4 model: four zoom levels (system context, container, component, code), a few supplementary diagrams and a short set of notation rules that make diagrams explain themselves. Section 4.4 covers the parts of UML that architects still use, chiefly sequence, state machine, component, deployment, activity and small class diagrams, and when UML's precision is worth the effort.

Decisions, templates, tooling and people: Sections 4.5 to 4.8

Diagrams show structure; they rarely show *why*. Section 4.5 covers Architecture Decision Records: Michael Nygard's 2011 format, alternatives such as MADR and Y-statements, the lifecycle of a record from proposed to superseded, and a complete worked record for the delivery platform. Section 4.6 shows how a template such as arc42 gives the whole description a standard organisation, so writers know what to cover and readers know where to look, and how to use it lightly for a small system.

Section 4.7 deals with the problem that defeats most documentation: staleness. It covers writing diagrams as text with PlantUML, Mermaid, Structurizr DSL and D2, the difference between drawing and modelling tools, publishing documentation from the build pipeline and the ownership and review habits that keep documentation true. Section 4.8 widens the lens from documents to people: tailoring the message to executives, developers, operations staff, security teams and auditors, explaining trade-offs, following one request through a system, design documents and RFC processes, presenting at a review and building consensus when people disagree.

The one idea to take away

Every technique in this chapter serves one goal: a shared, accurate understanding of the system among the people who build, run, pay for and depend on it. A diagram, a decision record or a template is worth producing only if a real reader needs it, and only for as long as someone keeps it true. Small, current, owned documentation that lives next to the code beats a comprehensive document that nobody trusts. If you remember one question from this chapter, make it this: *who will read this, and what will they do with it?*

SECTION 4.1 OF 4.8 · INTERMEDIATE · 28 MIN READ · LAST REVIEWED NOVEMBER 2026

Why and how much to document

BEFORE YOU READ THIS

This section builds on:

- 1.1 Defining architecture
- 1.5 Stakeholders and concerns

Terms it uses from earlier: Architecture description, Stakeholder, Version control, Pull request

THE IDEA IN BRIEF

Think of the paperwork for a house. A buyer wants a floor plan, an electrician wants the wiring diagram, an insurer wants the survey, and the person who renovates it in twenty years wants to know why that wall was never moved. Nobody wants a list of every brick, and a plan that shows a wall that has since been knocked down is worse than no plan at all. Software architecture documentation is the same: a small set of accurate documents, each written for someone who actually needs it, explaining the things you cannot see just by looking at the code.

WHAT YOU NEED TO KNOW

- An architecture description is the set of documents, diagrams and records through which people share an architecture; it serves education, communication among stakeholders and analysis.
- Different readers (newcomers, developers, operations, security, auditors, future maintainers and decision makers) bring different questions, so documentation should be organised around readers and their tasks.
- The Agile Manifesto values working software over comprehensive documentation, but explicitly says the documentation still has value; it rejects documentation for its own sake, not documentation.
- Write by hand what cannot be recovered from the code (context, quality goals, decisions and their reasons, principles); generate what the code already states and changes often.
- Views and Beyond's seven rules: write for the reader, avoid unnecessary repetition, avoid ambiguity, use a standard organisation, record rationale, keep documentation current but not too current, and review it for fitness of purpose.

- Docs-as-code and living documentation keep documentation in version control, reviewed with code changes and derived from the system where possible, which is the most reliable defence against staleness.
- Common anti-patterns are the stale wiki, the big-bang document nobody reads, diagrams without keys, mixed abstraction levels and descriptions that omit the why.

Why an architecture needs writing down

Every system has an architecture whether or not anyone describes it (see Section 1.1). The architecture description is a separate thing: the documents, diagrams, models and decision records through which people share an understanding of that architecture. In a team of three who sit together and wrote every line, much of the description can live in their heads. As soon as the team grows, people leave, a second team joins, an auditor arrives or the system outlives its authors, knowledge that exists only in heads becomes a risk. Ralph Johnson's definition of architecture as the shared understanding of the expert developers carries a warning: an understanding that is not written down can be shared only by talking, and talking does not scale.

Documenting Software Architectures: Views and Beyond (Paul Clements and colleagues at the Software Engineering Institute, 2nd edition, 2010) names three uses for architecture documentation, and they remain the clearest way to think about why it is worth the effort. It **educates**: it introduces new team members, external reviewers and new architects to the system. It is the main vehicle for **communication among stakeholders**: developers, operators, security staff and managers argue about the system through it, and decisions reached in one meeting reach people who were not there. And it is the **basis for analysis and construction**: it tells implementers what they may and may not do, and it is what an evaluator examines when asking whether the system will meet its quality goals (the evaluation methods of Chapter 11 need a description to work from).

Two consequences follow. First, documentation is a means, not an end: a document that serves none of these uses is waste, however well formatted. Second, the reader decides whether it worked. A description that its author finds complete but that a newcomer cannot use has failed at education, and a diagram the operations team cannot map to real servers has failed at analysis.

Who reads architecture documentation, and what they need

Reader	Questions they bring	What serves them best
Newcomers (new developers, contractors, a new architect)	What does this system do, for whom, and where do I start? What are the main parts and the words people use?	A one-page overview, a system context diagram, a container diagram, a glossary and a short reading order
Current developers	Where does this new feature belong? What may I depend on? Why is it built this way, and may I change it?	Module or component structure, conventions, cross-cutting concepts (error handling, security, logging) and decision records
Operations and SRE staff	Where does it run, what does it depend on, how does it fail and how do we recover?	Deployment view per environment, external dependencies, runbooks, monitoring and recovery objectives
Security and privacy staff	Where are the trust boundaries? Where does personal data flow and rest? Who can call what?	Context and data-flow views, identity and access concept, list of third-party services
Auditors and regulators	Can you show that the controls exist, that decisions were made deliberately and that changes are traceable?	Decision records with dates and owners, traceability from requirements to design and tests, version history
Future maintainers	Why on earth was this done? Which constraints still apply? What is safe to remove?	Rationale above all: decisions with their context, the options rejected and the assumptions made
Decision makers (sponsors, product and engineering managers)	What are the risks, costs and options? What do I need to decide, and by when?	A one-page summary, quality goals, top risks and technical debt, decisions awaiting an owner

Working software over comprehensive documentation

The Agile Manifesto (2001) lists four value pairs, one of which is “Working software over comprehensive documentation”. It is quoted, often, as permission not to document at all. Read in full, the manifesto says something narrower. Its authors state that while there is value in the items on the right, they value the items on the left more. The value is a priority for when the two compete, not a rejection of documentation.

What the value rejects is the documentation-driven project of the 1990s: hundreds of pages of specifications signed off before code is written, treated as progress in themselves, and abandoned once the code diverges. The word that matters is *comprehensive*: documentation that tries to describe everything, for its own sake, and is valued more than the running system.

Nothing in the manifesto argues against a context diagram, a short record of why the team chose asynchronous messaging, or a runbook that lets the on-call engineer restore service at three in the morning.

Practitioners do disagree about how far to go. Some agile coaches argue that conversation and tests are almost always enough, and that written architecture documentation goes stale faster than it pays back. Others, especially those who maintain long-lived systems or work in regulated sectors, point out that the people who most need the documentation (successors, auditors, the operations team) are exactly those absent from the conversations. Both positions agree on the underlying economics: documentation has a cost to write, a cost to keep current and a cost to read, and it is worth producing only where its value to readers exceeds those costs.

Just enough documentation

Scott Ambler's phrase "just barely good enough" describes the target: a model or document is sufficient when it serves its purpose for its audience, and effort beyond that point costs more than it returns. His point that the audience, not the author, decides whether an artefact is sufficient is the same principle as writing for the reader. George Fairbanks makes the equivalent argument for architecture work as a whole in *Just Enough Software Architecture* (2010): do as much as the risks justify (see Section 1.8).

Just-enough documentation is therefore not a fixed quantity. It grows with the size and lifetime of the system, the number of teams, the turnover of people, the cost of failure and the demands of regulators. A two-person prototype needs a README and perhaps one diagram. A payments platform run by eight teams under financial regulation needs context, container and deployment views, a decision log going back years, documented security concepts and evidence that the documents are reviewed.

A practical signal is the repeated question. If newcomers ask the same thing in their first week, or the same design argument recurs every few months, the answer belongs in writing. Another is the expensive mistake: if a team broke an unwritten rule ("never call the payment provider from the restaurant app") because nobody told them, the rule was architecture and should have been written down.

What is worth documenting, and what is not

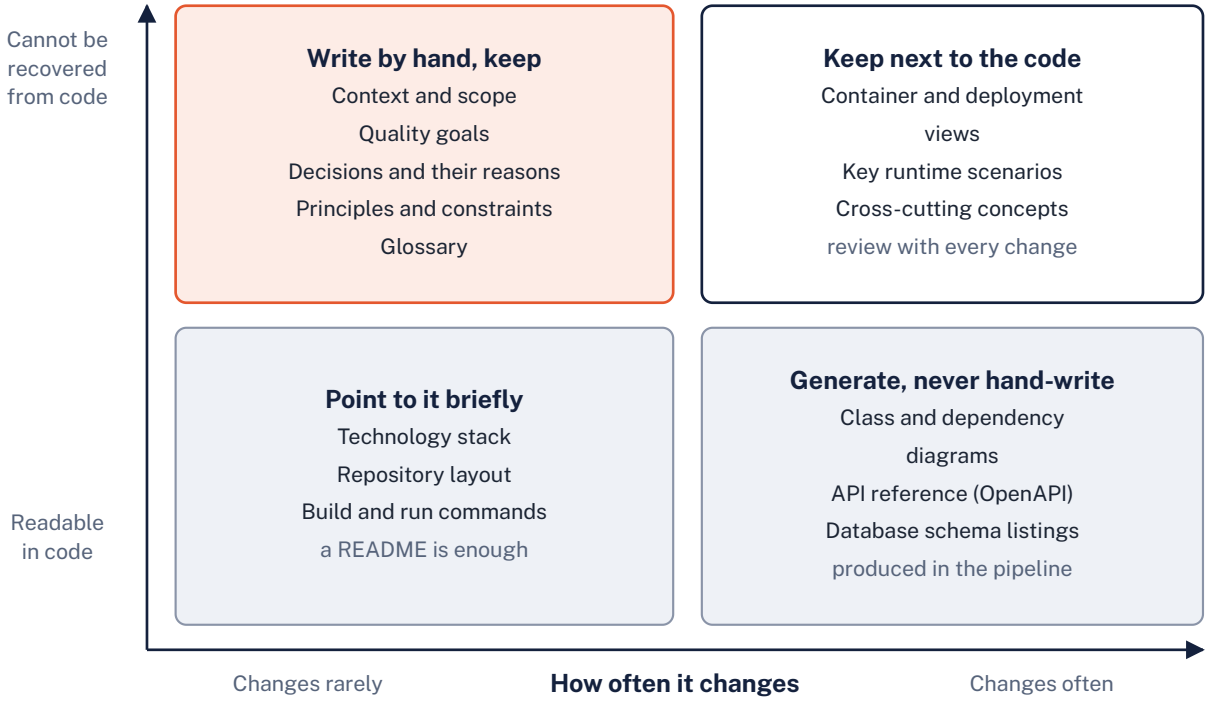
Information differs in two ways that decide how it should be documented. The first is whether it can be **recovered from the code**. Class names, method signatures and database tables can be read from the source at any time; the reasons a decision was made, the quality goals the

system serves and the options that were rejected cannot, because they were never in the code. The second is **how often it changes**. The system's purpose and its main external partners change rarely; the list of classes in a module changes weekly.

Hand-written architecture documentation should concentrate on what is both important and not recoverable from code: the system's **context and scope** (who uses it and which other systems it depends on), its **quality goals** and constraints, the **key decisions and their reasons**, the coarse-grained **structure** (the main deployable parts and their responsibilities), the **deployment** onto infrastructure, the important **runtime scenarios**, and the **cross-cutting concepts** that every part must follow, such as how errors are handled, how users are authenticated and how personal data is protected. This list is close to the structure of the arc42 template, covered in Section 4.6.

What is not worth writing by hand is anything the code or the tools already state precisely and that changes often: class diagrams of a whole codebase, lists of endpoints that an OpenAPI file already holds, or descriptions of each function. Where such views are useful, generate them from the source in the build pipeline so they cannot drift. Also omit aspirational content (diagrams of how the system ought to look, presented as if it already did), minutes of meetings whose outcome is captured elsewhere, and duplicated text, which will eventually contradict its copy.

Deciding how to document a piece of information



Write by hand what the code cannot tell you; generate what the code already says and changes often.

IF YOU ARE NEW TO THIS

Step by step: write it, generate it or point to it

The idea that needs the most care here is deciding, for each piece of knowledge about a system, whether a person should write it down, a tool should generate it or a short pointer is enough. Two questions settle it: can it be read from the code, and how often does it change? Try five facts about the food-delivery platform.

- Why order events are sent asynchronously instead of by direct calls. The code does not say why, or what was rejected, and the reasons rarely change. A person writes them down once, as an Architecture Decision Record.
- The rule 'never call the payment provider from the restaurant app'. The code shows that the app does not call the provider today, not that it must never do so. A team that never heard the rule could break it, so it is written down too.
- The list of the ordering service's endpoints. Its OpenAPI file already states them precisely, and they change often, so the pipeline generates the reference from that file.
- The diagram of the platform's main apps, services and databases and how they connect. No single codebase shows that whole picture, yet it changes whenever a part is added or removed. So it is kept in the repository, ideally drawn from a model there, and the pull request that adds or removes a part changes it too.
- The commands to build and run each service. They are in the build files and rarely change, so a short README pointing to them is enough.
- **The lesson:** spend hand-written effort where the code is silent: reasons, rules, goals and the big picture. Generate or point to whatever the code already states: hand-made copies drift.

Seven rules for sound documentation

Views and Beyond sets out seven rules for sound documentation. They are worth knowing by heart because almost every documentation failure breaks one of them.

1

1. Write for the reader

Documentation is written once and read many times, so optimise for the reader. Decide who the readers are and what they will try to do with the document, organise it for lookup rather than for linear reading, and avoid private jargon. A sign that this rule is broken: the document is organised in the order the author thought of things.

2 **2. Avoid unnecessary repetition**
Each piece of information should have one home, with links from elsewhere. Repetition makes documents longer and, worse, lets copies drift apart until readers cannot tell which is true. Deliberate repetition for the reader's convenience, such as a summary, is fine if it is clearly marked.

3 **3. Avoid ambiguity**
A box-and-line diagram with no key can be read in several ways: is a line a call, a data flow or a deployment dependency? Define the notation in a key, use one name for one thing, and state what each element and relation means.

4 **4. Use a standard organisation**
A known structure (a template such as arc42 or the Views and Beyond template) tells readers where to look and shows authors what is missing. An empty section marked "not applicable, because..." is more informative than a silent gap.

5 **5. Record rationale**
Write down why, including the options rejected. Future maintainers can usually work out what was built; they cannot work out why, and without the why they either fear changing anything or undo a decision whose reasons still hold. Decision records exist for this.

6 **6. Keep it current, but not too current**
Out-of-date documentation misleads. But updating every document for every half-formed idea creates churn; record decisions once they are made and update the description on a predictable rhythm, such as with each release or each significant change.

7 **7. Review it for fitness of purpose**
Only the intended readers can say whether a document works. Ask a newcomer to set up and explain the system from the documents, or ask an on-call engineer to use the deployment view in a drill, and fix what they stumble over.

Living documentation and docs-as-code

The traditional failure of architecture documentation is staleness: the description is written at the start, the system changes, and nobody updates the document because it lives somewhere separate from the work. Two related ideas attack this problem at its root.

Docs-as-code, a practice promoted by the Write the Docs community, means treating documentation like source code: written in plain-text formats such as Markdown or AsciiDoc, kept in version control (usually in the same repository as the code it describes), changed

through the same pull requests, reviewed by the same people and built and published by the same pipeline. A pull request that changes how orders are dispatched can then be required to update the runtime view in the same commit, and the reviewer sees both.

Living documentation, a term popularised by Cyrille Martraire's book of the same name (2019), goes further: wherever possible, documentation should be derived from the system itself so that it cannot drift. Dependency diagrams generated from the code, API references generated from OpenAPI files, architecture rules checked by automated tests, and diagrams generated from a model kept next to the code are all examples. Hand-written text is reserved for what cannot be derived, chiefly intent and rationale.

Neither idea removes the need for judgement about what to write. Section 4.7, Diagrams as code, covers the tools and habits in detail.

DECISION GUIDE

Decision guide: should we write this down?

Ask these questions in order about a piece of architectural knowledge.

Q1 **Would someone make a costly mistake, or ask the same question repeatedly, if this were not written down?**

- **Yes** Continue: it is a candidate for documentation.
- **No** Leave it to conversation and the code for now; revisit if the question starts recurring.

Q2 **Can it be read reliably from the code, configuration or a tool?**

- **Yes** Generate it, or link to where it lives; do not copy it by hand.
- **No** Continue: it needs to be written by a person.

Q3 **Is it a decision with reasons and rejected options?**

- **Yes** Write a decision record (see Section 4.5).
- **No** Continue.

Q4 **Will it change with most releases?**

- **Yes** Keep it in the repository next to the code it describes and make updating it part of the pull request.
- **No** Put it in the architecture overview or the appropriate template section and review it on a fixed rhythm.

WORKED EXAMPLE

Worked example: a just-enough documentation set for the food-delivery platform

A food-delivery company runs a platform with a customer app, a restaurant app, a courier app, ordering, dispatch and notification services, and an external payment provider. It has 30 engineers in five teams and is preparing for an audit by a card-payment assessor. The architect proposes this documentation set, all kept in a single documentation repository or alongside the code, and none of it longer than it needs to be.

Architecture overview (2 to 3 pages)	Purpose, main users, quality goals in priority order (for example: never lose or double-charge an order; order confirmation p99 under 1.5 seconds at peak; restaurant app usable on poor connections), key constraints, and a reading order for newcomers. Owner: the architect. Reviewed each quarter.
System context and container diagrams	Two C4 diagrams (Section 4.3) with keys, generated from a model in the repository. Updated in the pull request that adds or removes a container or an external dependency.
Decision log	About 40 short decision records, numbered, in the repository: asynchronous order events, the choice of payment provider, one database per service, and so on. Never rewritten; superseded when a decision changes.
Deployment view	One diagram and a table per environment: regions, clusters, managed databases, the message broker and the network boundary around card data. Owned by the platform team.
Runtime scenarios	Sequence diagrams for the four flows that matter most: place order, restaurant rejects, courier assignment, refund. Each fits on one screen.
Cross-cutting concepts	One page each for authentication, idempotency and retries, logging and personal data. These are the rules every team must follow.
Runbooks	One per alert that can wake someone up, linked from the alert itself.
Not written by hand	Class diagrams, endpoint lists and database schemas (generated in the pipeline); meeting notes; the history of how the system grew (the decision log and version history already tell it).

Every item has a named reader, an owner and an update trigger. Documentation without an owner decays first.

The audit is served by the decision log, the deployment view and the version history, not by a separate audit document that would have to be kept consistent with them.

The whole set can be read in about two hours, which makes it plausible that newcomers will actually read it.

WATCH OUT

Documentation anti-patterns

These are the failures seen most often in practice, and the rule each one breaks.

- **The stale wiki.** Hundreds of pages, last edited years ago, partly true and impossible to tell which part. Readers learn to distrust all of it. Remedy: fewer pages, each with an owner and a "last reviewed" date, and delete what nobody maintains.
- **The big-bang document.** A 120-page architecture specification written up front, approved once and never read again. It breaks the rule about writing for the reader and is out of date within months.
- **Diagrams without keys.** Boxes of different colours and lines of different styles with no legend. Every reader invents a meaning, and two readers invent different ones.
- **Mixed levels of abstraction.** A single diagram showing a mobile app, a Java class, a Kubernetes cluster and a business process. Use separate views (Section 4.2).
- **Documentation only in slides.** Decks are made for one meeting and one audience, are rarely versioned and cannot be searched. Keep the source of truth in the repository and build slides from it.
- **The what without the why.** A complete description of the structure with no rationale, so future teams cannot tell which constraints are essential and which were accidents.
- **Write-only documentation.** Produced to tick a governance box, never reviewed by a real reader, and therefore never tested for fitness of purpose.

Questions people ask

Our code is clean and well tested. Is that not documentation enough?
Clean code and tests are excellent documentation of *what* the system does at a fine-grained level. They do not show the big picture across services and repositories, the quality goals, the constraints from outside the code, or why the structure is the way it is. Those are exactly what architecture documentation is for.

Who should write the architecture documentation?
The people who make and live with the decisions, which in most modern teams means engineers as well as architects. The architect usually owns the overall structure and consistency, and teams own the parts that describe their services. What matters most is that every document has a named owner.

How do we stop documentation going stale?
Keep it small, keep it next to the code, make updating it part of the definition of done for changes that affect it, generate what can be generated, and review it on a fixed rhythm. Section 4.7 covers these habits in detail.

Developer portals: making documentation findable

Documentation nobody can find may as well not exist. A developer portal gives engineers one place to look. Backstage, the best-known (Section 10.7), centres on a software catalogue built from metadata files stored with the code, so ownership is recorded and updated through normal reviews. Its TechDocs feature takes Markdown kept beside the code and publishes it as a documentation site, reachable from the service's catalogue page. On the food-delivery platform, the Ordering page could show the owning team, its decision records and its runbooks together. A portal is an index, not a cure: stale pages stay stale, and running one has a cost. A small team may need only a well-kept repository.

AsyncAPI and schema registries as contract documentation

Event flows are easy to lose sight of, because no single service shows them. An AsyncAPI document (Section 6.5) records them in machine-readable JSON or YAML: the channels, which applications send or receive on them, and the message formats. It works across protocols such as Kafka, MQTT and AMQP; check the current version. A schema registry adds enforcement. Confluent's, for example, checks each new version of a schema against a compatibility rule, backward by default, and refuses incompatible ones, and open-source Apicurio Registry can also store OpenAPI and AsyncAPI documents (Section 7.7). So `OrderPlaced`'s documentation cannot silently drift from what is published. Neither explains what a field *means*; that still needs words.

CHECK YOUR UNDERSTANDING

▼ **Name the three uses of architecture documentation identified in Views and Beyond.**

Education (introducing people to the system), communication among stakeholders, and serving as the basis for analysis and construction.

▼ **Does "working software over comprehensive documentation" mean agile teams should not document their architecture?**

No. The manifesto states that the items on the right still have value; it sets a priority when the two compete and rejects documentation produced for its own sake, not useful documentation.

▼ **Why should rationale be documented by hand while class diagrams generally should not?**

Rationale cannot be recovered from the code and changes rarely, so it must be written by a person. Class structure can be read or generated from the code and changes often, so a hand-drawn copy quickly drifts.

▼ **What does "keep documentation current, but not too current" warn against?**

Both stale documentation that misleads readers and constant churn from updating documents for every unsettled idea; update on a predictable rhythm once decisions are made.

▼ **Try it: A ticketing team has an 80-page wiki, out of date, that nobody reads. Developers ask what to replace it with and fear wasting time. What would you write by hand, generate, or delete, and how would it stay current?**

Hand-write what code cannot tell you: context, quality goals, principles, and decisions with their reasons, as short ADRs. Generate what the code already states and changes often, such as API references. Delete stale pages rather than patching them. Keep everything in the repository, reviewed in the same pull request as the code change, so it stays current. Organise it around readers' questions, a newcomer's or an operator's for example, not around the old structure.

SOURCES

- Manifesto for Agile Software Development: https://agilemanifesto.org/
- Documenting Software Architectures: Views and Beyond, 2nd edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- Just Barely Good Enough models and documents (Scott Ambler, Agile Modeling): https://agilemodeling.com/essays/barelygoodenough.htm
- Docs as Code (Write the Docs): https://www.writethedocs.org/guide/docs-as-code/
- Principles of technical documentation (Gernot Starke, INNOQ): https://www.innoq.com/en/blog/2022/01/principles-of-technical-documentation/

- Backstage: TechDocs: https://backstage.io/docs/features/techdocs/
- Backstage: Software Catalog: https://backstage.io/docs/features/software-catalog/
- AsyncAPI specification: https://www.asyncapi.com/docs/reference/specification/latest
- Confluent: Schema evolution and compatibility: https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html
- Apicurio Registry: https://www.apicur.io/registry/

SECTION 4.2 OF 4.8 · INTERMEDIATE · 22 MIN READ · LAST REVIEWED NOVEMBER 2026

Views and viewpoints: ISO/IEC/IEEE 42010, 4+1 and Rozanski and Woods

BEFORE YOU READ THIS

This section builds on:

- 1.1 Defining architecture
- 1.5 Stakeholders and concerns

Terms it uses from earlier: Stakeholder, Concern, Architecture description

THE IDEA IN BRIEF

A house is never described by one drawing. The buyer looks at the floor plan, the electrician at the wiring plan, the plumber at the pipe layout and the planning officer at the drawing of the front of the house. Each drawing leaves most things out on purpose so that it can answer one person's questions clearly. Software architecture works the same way. Instead of one huge diagram, architects draw several views of the same system, each for a particular group of people, and choose only the views those people actually need.

WHAT YOU NEED TO KNOW

- A view represents a system from the perspective of particular concerns; a viewpoint is the reusable set of conventions (concerns framed, stakeholders, model kinds) for constructing one kind of view.
- In ISO/IEC/IEEE 42010:2022, stakeholders have concerns, viewpoints frame concerns and govern views, views address concerns and consist of view components governed by model kinds, and correspondences relate elements across views.
- Kruchten's 4+1 model (IEEE Software, 1995) uses logical, process, development and physical views, tied together and validated by a small set of scenarios.
- Rozanski and Woods define seven viewpoints (context, functional, information, concurrency, development, deployment, operational) and perspectives, such as security and performance, that are applied across all views rather than drawn as views.
- The SEI groups views into module, component-and-connector and allocation categories; a complete documentation set includes at least one of each.
- Choose the minimum set of views by mapping stakeholders and concerns to views, combining views where possible and documenting risky areas in more depth than routine ones.

One system, many views

No single diagram can describe a real system. A drawing that shows how the food-delivery platform's code is divided into modules says nothing about which servers they run on; a drawing of servers and networks says nothing about what happens, step by step, when a courier accepts a job. If you try to put all of this on one page you get the familiar architecture poster: dozens of boxes, lines meaning six different things, and nobody able to answer a precise question from it.

The idea that resolves this is old and simple. Describe the system through several views, each showing one set of structures for one set of concerns, just as a building is described by a floor plan, an elevation, a wiring plan and a plumbing plan. Each is a partial description; together they describe the whole. This idea underpins the international standard, Kruchten's 4+1 model, Rozanski and Woods's viewpoint catalogue, the SEI's Views and Beyond approach and, in a lightweight form, the C4 model in the next section. This section explains the vocabulary they share and how to choose the views a particular system needs.

Recall from Section 1.5 that a stakeholder is anyone with an interest in the system and a concern is any matter of importance to a stakeholder. Views exist to answer concerns. A view that answers nobody's concern is decoration.

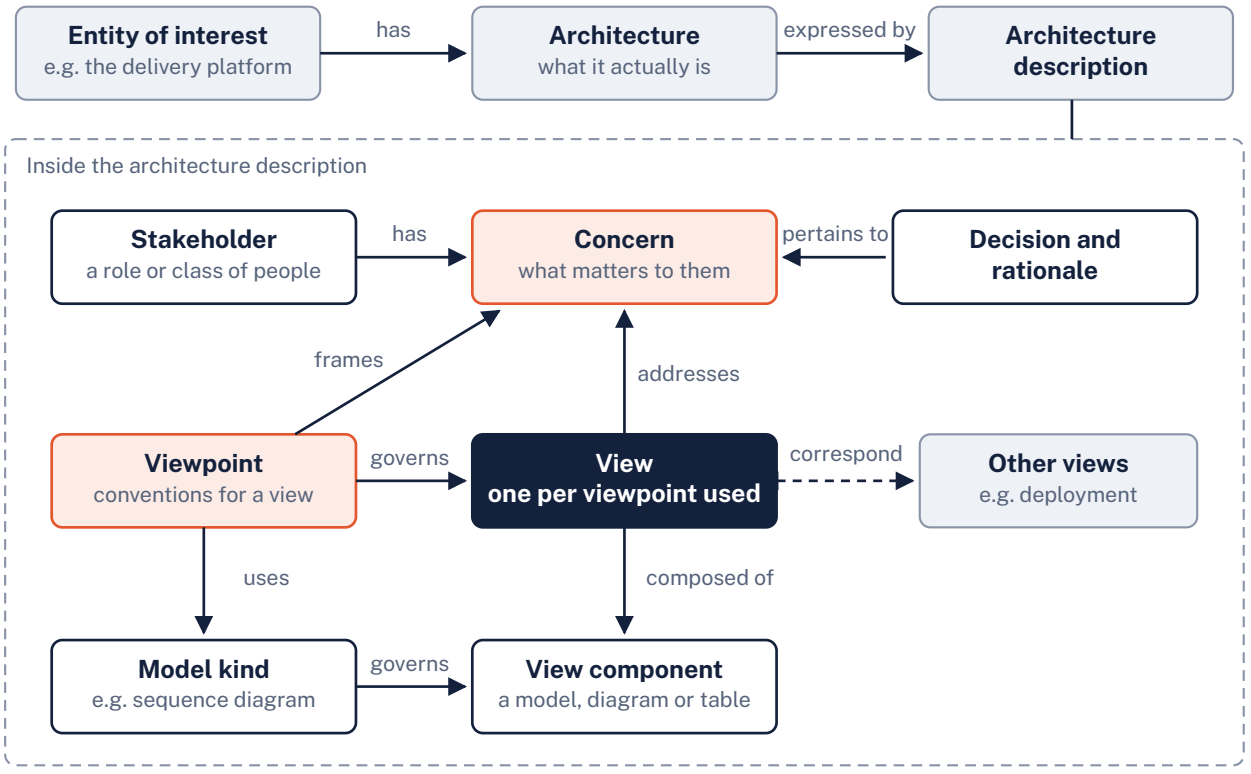
The ISO/IEC/IEEE 42010:2022 conceptual model

ISO/IEC/IEEE 42010 is the international standard for architecture description. Its current edition was published in 2022 and replaced the 2011 edition. It does not tell you how to design, which notation to use or which views to draw; it defines the concepts an architecture description should contain and how they relate, so that descriptions produced by different methods can be compared and checked. Its conceptual model is the common skeleton of almost every documentation approach in this chapter.

The model starts with the entity of interest: the thing whose architecture is described, which can be a software system, a system of systems, a product line or an enterprise (the 2011 edition called it the system of interest). The entity has an architecture, and an architecture description expresses it. Inside the description, **stakeholders** have **concerns**. A viewpoint is a reusable specification of conventions for constructing and using one kind of view: it names the concerns it frames, the typical stakeholders, and the model kinds (notations and modelling conventions) to use. A **view** applies a viewpoint to a particular entity: it addresses those concerns for this system, and is made of view components, such as diagrams, tables and models, each governed by a model kind (the 2011 edition called view components architecture models).

Two further elements matter in practice. Correspondences record relations between elements of the description, most usefully between views: this container in the container view runs on that cluster in the deployment view; this requirement is addressed by that decision. And **architecture decisions and their rationale** are part of the description, linked to the concerns they address. The 2022 edition also defines more fully two concepts the 2011 edition already mentioned, stakeholder perspectives (ways of grouping concerns) and architecture aspects (characteristics of the entity that cut across views), renames "architecture framework" as architecture description framework, and defines what conforming architecture description frameworks, such as a company's own documentation template, must contain.

The 42010 conceptual model, simplified



Concerns are the hinge: viewpoints frame them, views address them and decisions pertain to them.

IF YOU ARE NEW TO THIS

If you are new to this: viewpoint versus view

A viewpoint is like a type of map: "a road map" is defined by its conventions (roads in red, motorways thicker, a scale bar) and its users (drivers). A view is a particular map drawn to those conventions: the road map of Scotland. You define the viewpoint once, perhaps for the whole company, and draw a view of it for each system.

A model kind is the drawing technique a viewpoint prescribes, such as "UML sequence diagram" or "C4 container diagram". The distinction lets an organisation say: every system's deployment view must show regions, network zones and managed services, drawn as a C4 deployment diagram with a table of environments.

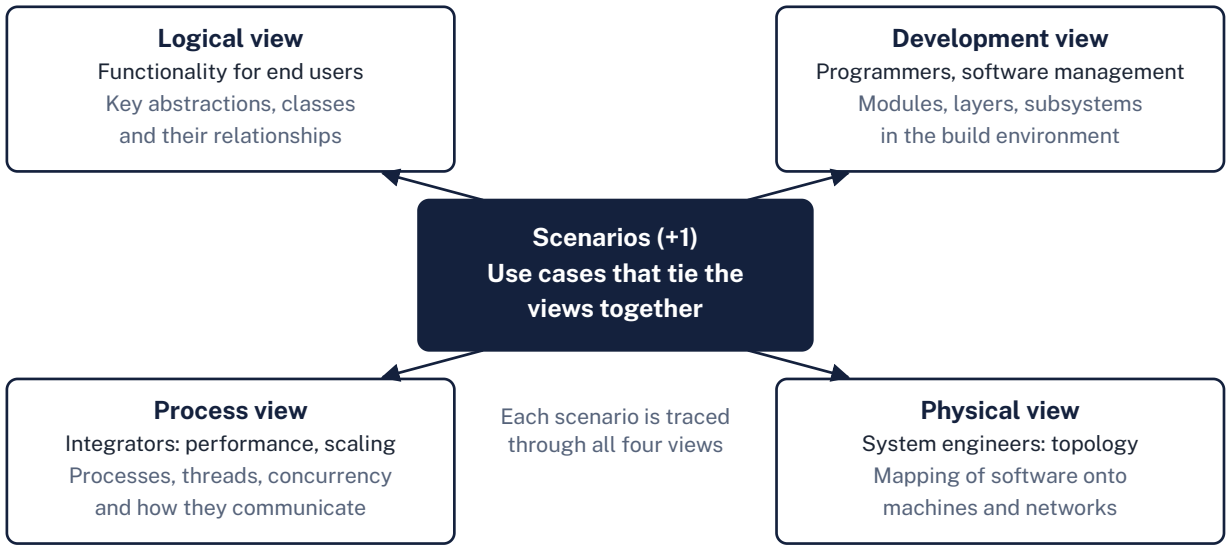
Kruchten's 4+1 view model

Philippe Kruchten, then at Rational Software, published *Architectural Blueprints: The "4+1" View Model of Software Architecture* in *IEEE Software* in November 1995. The 4+1 view model was the first widely adopted multi-view approach and became part of the Rational Unified Process. Kruchten's argument was that the single diagrams of the day tried to capture too much at once, and that different stakeholders (end users, developers, system engineers, project managers) need different views.

The four views are these. The **logical view** describes the functionality the system provides to end users, typically as the key abstractions and their relationships (in UML, class and state diagrams). The **process view** addresses runtime concerns: processes and threads, concurrency, distribution, performance, scalability and fault tolerance. The **development view** (also called the implementation view) describes the organisation of the software in its development environment: modules, subsystems, layers and their dependencies, which matter to programmers and to managers planning the work. The **physical view** (or deployment view) maps the software onto hardware and networks. The "+1" is the **scenarios**: a small set of important use cases, each traced through all four views, which both illustrate the architecture and validate that the views fit together.

The scenarios are the part most often forgotten and arguably the most valuable. A view in isolation can look tidy while hiding a gap; walking one real scenario such as "customer places an order" through the logical, process, development and physical views exposes missing elements and inconsistent assumptions. The same idea reappears in the runtime view of arc42, in C4 dynamic diagrams and, in Section 4.8, in the advice to explain a system by following one request.

The 4+1 view model



Four views for four groups of stakeholders, held together by a small set of scenarios.

Rozanski and Woods: viewpoints and perspectives

Nick Rozanski and Eóin Woods's *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* (2nd edition, Addison-Wesley, 2012) offers a catalogue of seven viewpoints aimed at large information systems, together with a second, complementary idea: perspectives.

Their seven viewpoints are: **context** (the system's relationships with people, systems and external entities around it), **functional** (the runtime functional elements, their responsibilities, interfaces and primary interactions), **information** (how data is stored, managed, owned and distributed), **concurrency** (which parts run concurrently and how they are coordinated), **development** (what developers, testers and builders need: module structure, common processing, standards and codeline organisation), **deployment** (the runtime environment and how software maps onto it) and **operational** (how the system is installed, migrated, administered, monitored and supported in production).

A perspective is their name for a quality concern that cuts across all views, such as security, performance and scalability, availability and resilience, or evolution (their catalogue also includes accessibility, development resource, internationalisation, location, regulation and usability). A perspective is not another view to draw. It is a set of questions, tactics and checks applied to every view: for security, you examine the context view for external actors and trust boundaries, the information view for sensitive data, the deployment view for network zones, and so on. This captures something single views cannot: qualities emerge from the whole system, so they have to be checked everywhere (compare Chapter 2's quality models).

The operational viewpoint deserves particular attention because many approaches underplay it. Who installs the system, how data is migrated from the old one, how configuration is managed, what operators monitor and how they are alerted: these are architectural questions whose answers shape the system, and leaving them until just before go-live is a classic cause of failed launches.

The SEI's three categories: module, component-and-connector, allocation

Chapter 1 introduced the SEI's grouping of structures into three kinds (see Section 1.1). *Documenting Software Architectures: Views and Beyond* uses the same grouping to organise views, and it is the most useful way to check that a set of views is complete.

Module views show units of implementation (packages, libraries, namespaces) and relations such as *is part of*, *depends on* and *is a*. Typical styles are decomposition, uses, generalisation, layered and data model views. Component-and-connector (C&C) views show the elements present at runtime (services, clients, databases, queues) and the connectors through which they interact (HTTP calls, messages, shared data); typical styles include client-server, publish-subscribe, pipe-and-filter, shared-data and multi-tier. Allocation views map software elements onto non-software things: onto hardware and cloud infrastructure (deployment), onto the file system and build artefacts (implementation and install) and onto teams (work assignment).

The three categories map naturally onto the other approaches. Kruchten's development view is a module view, his process view a C&C view and his physical view an allocation view. Rozanski and Woods's functional and concurrency viewpoints are mostly C&C, their development viewpoint is module, and their deployment and operational viewpoints are allocation. The C4 container and component diagrams of the next section are C&C views with some module flavour, and C4 deployment diagrams are allocation views. When you assemble a documentation set, check that it contains at least one view from each category, because each answers questions the others cannot.

Approaches compared

Concern area	4+1 (Kruchten)	Rozanski and Woods	SEI category	C4 diagram
System boundary, users, external systems	(implicit in scenarios)	Context	C&C (context)	System context
Runtime elements and interactions	Process (and logical)	Functional, concurrency	Component-and-connector	Container, component, dynamic
Data ownership and flow	(not separate)	Information	Module (data model) and C&C	(not separate)
Code organisation	Development	Development	Module	Component, code
Mapping to infrastructure	Physical	Deployment	Allocation	Deployment
Running the system in production	(not covered)	Operational	Allocation (partly)	(not covered)
Validation through use cases	Scenarios (+1)	(applied to every view)	Behaviour documentation	Dynamic

Mapping stakeholders and concerns to views

The most practical tool from the standard is a simple table: list the stakeholders and their main concerns, then name the view or views that address each concern. Gaps (a concern no view addresses) show where the description is incomplete; views no one needs are candidates for deletion. The table below is a first pass for the food-delivery platform.

Stakeholders, concerns and views for the food-delivery platform

Stakeholder	Main concerns	Views that address them
Product managers and sponsors	Scope, external partners, what is bought versus built, cost	Context view; one-page overview with quality goals and top risks
Developers in each team	Where code belongs, interfaces between services, conventions	Container view; component view for complex services; cross-cutting concepts; decisions
Operations and SRE	Where things run, failover, capacity, dependencies, recovery	Deployment view per environment; operational view (monitoring, runbooks, backup and restore)
Security team	Trust boundaries, authentication, card data, secrets	Context and deployment views annotated with trust zones; security concept (a perspective applied to every view)
Data protection officer	Personal data of customers and couriers: where it flows, how long it is kept	Information view with a data-flow diagram and retention table
Payment provider's integration team	Which calls you make, webhooks, idempotency, versions	Interface specification and a sequence diagram of the payment flow
Card-payment assessor (auditor)	Scope of the card-data environment, controls, change history	Deployment view showing the card-data boundary; decision log; version history

Choosing the minimum set of views

The temptation, once you know that many viewpoints exist, is to produce all of them. Resist it. Every view has to be kept consistent with the system and with the other views, so each one adds a maintenance cost. The aim is the smallest set of views that answers the important concerns of the important stakeholders, following the rule from Section 4.1 to write for the reader. Views and Beyond describes a similar procedure: build a stakeholder-and-view table, combine views where one diagram can serve several needs, then prioritise.

A procedure for choosing views

1. **1. List stakeholders and concerns**
   Use the stakeholder analysis from Chapter 1. For each concern, note how important it is and how much is at stake if it is misunderstood.

2 **2. Map concerns to candidate views**
Use a catalogue (4+1, Rozanski and Woods, or the SEI categories) as a checklist so that whole areas, typically information and operations, are not forgotten.

3 **3. Combine where you can**
A container diagram annotated with technologies and protocols often serves developers and operations at once; a deployment diagram with trust zones serves operations and security. Combining reduces the number of artefacts to maintain.

4 **4. Check coverage of the three categories**
Make sure the set includes at least one module, one component-and-connector and one allocation view, plus a few scenarios that walk through them.

5 **5. Decide the depth of each view**
Not every view needs the same detail. Document the risky or complex parts deeply and the routine parts briefly.

6 **6. Record correspondences**
State how elements in one view relate to elements in another (container to deployment node, module to container) so readers can move between views. Mismatches between views are where many design errors hide.

WORKED EXAMPLE

Worked example: the minimum view set for the food-delivery platform

Applying the procedure to the platform described in Section 4.1 (five teams, card payments through an external provider, an upcoming payment-security assessment) gives the following set.

Context view	One C4 system context diagram: customers, restaurant staff, couriers, support agents, the payment provider, the mapping service, the push-notification services. Serves everyone; the first thing a newcomer reads.
C&C view	One C4 container diagram for the whole platform, plus component diagrams only for the ordering and dispatch services, which are the most complex and change most.
Information view	A table of the main data entities, which service owns each, where personal data flows, and retention periods; one data-flow diagram for customer data. Added because the data protection officer's concerns were otherwise unaddressed.
Allocation view	A C4 deployment diagram for production, with the card-data boundary and network zones marked, and a short table of differences in the staging environment.
Module view	Not drawn by hand. Each service's internal module structure is generated from the code and checked by architecture tests.
Scenarios	Four sequence diagrams: place order, restaurant rejects order, courier assignment, refund. Each is checked against the container and deployment views.
Perspectives	Security and availability are applied as checklists to every view during review rather than drawn as separate views.

Seven artefacts, most of them single diagrams, answer the concerns of all seven stakeholder groups in the table above.

The information view was added only because the stakeholder-to-view table exposed a gap. Without the table it would have been missed, which is common.

The module view is generated, not drawn, because it is recoverable from code and changes often (Section 4.1).

GOING DEEPER

Going deeper: documenting a view properly

Views and Beyond recommends a standard structure for each view, which also makes a good review checklist: a **primary presentation** (usually the diagram, showing the elements and relations), an **element catalogue** (what each element is, its responsibilities, interfaces and properties), a **context diagram** showing what lies outside the part being shown, a **variability guide** (what can vary, such as optional components or configuration), and the **rationale** for the design shown.

It also stresses documentation that lies *beyond* the views, which gives the book its name: a roadmap explaining how the documentation is organised, a system overview, the mapping between views, the rationale for decisions that cut across views, and a directory or glossary. Most of the anti-patterns in Section 4.1 come from skipping these parts.

WATCH OUT

Common mistakes with views

- Mixing categories on one diagram, for example drawing code packages, running services and servers as boxes on the same canvas with lines that mean different things.
- Drawing views nobody asked for because a framework lists them, while missing the one view the operations team needs.
- Letting views contradict each other: the container view shows six services but the deployment view shows five. Correspondences and reviews catch this.
- Treating quality attributes as one more view instead of a perspective that has to be checked in every view.

CHECK YOUR UNDERSTANDING

▼ **What is the difference between a viewpoint and a view?**

A viewpoint is a reusable specification of conventions for one kind of view (concerns framed, stakeholders, model kinds). A view applies a viewpoint to a specific system and addresses those concerns for it.

▼ **What role do scenarios play in the 4+1 model?**

They are a small set of important use cases traced through all four views, which illustrate the architecture and validate that the views are consistent and complete.

▼ **How does a Rozanski and Woods perspective differ from a viewpoint?**

A viewpoint produces a view. A perspective is a cross-cutting quality concern, such as security, applied as questions and tactics to every view rather than drawn as a view of its own.

▼ **Which SEI category does a deployment diagram belong to, and why?**

Allocation, because it maps software elements onto non-software things, in this case hardware and cloud infrastructure.

▼ **Try it: For a ticketing platform, three readers matter: new developers, operators on sale day, and a payment-security assessor. Which minimum set of views would you produce, and how does each reader's concern justify it?**

A context view for everyone. A container view shared by developers and operators, so one diagram serves two concerns. A deployment or operational view for operators, showing zones, scaling and failure handling. A security perspective applied across those views, showing the trust boundaries and card data flows for the assessor. Add a runtime view of the purchase journey, because it is the risky area. Skip detailed views of routine parts. Fewer views stay current.

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
- Philippe Kruchten, Architectural Blueprints: The 4+1 View Model of Software Architecture (arXiv copy of the 1995 IEEE Software paper): https://arxiv.org/abs/2006.04975
- Viewpoints catalogue, Rozanski and Woods: https://www.viewpoints-and-perspectives.info/home/viewpoints/
- Documenting Software Architectures: Views and Beyond, 2nd edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/

SECTION 4.3 OF 4.8 · INTERMEDIATE · 24 MIN READ · LAST REVIEWED NOVEMBER 2026

The C4 model: context, containers, components and code

BEFORE YOU READ THIS

This section builds on:

- 0.4 How computers talk
- 0.7 Servers and the cloud
- 4.2 Views and viewpoints

Terms it uses from earlier: API (application programming interface), HTTP, Viewpoint

THE IDEA IN BRIEF

Think of an online map. Zoomed out, you see a whole country and the motorways between cities. Zoom in and you see one city's districts, then one district's streets, then a single building. Each zoom level is useful to different people, and you always know which level you are looking at. The C4 model does the same for software: one picture shows the whole system and who uses it, the next shows the apps and databases inside it, the next shows the parts inside one app, and the last, rarely needed, shows the code itself.

WHAT YOU NEED TO KNOW

- The C4 model, created by Simon Brown, describes software with a hierarchy of four diagrams (system context, container, component and code) built on the abstractions person, software system, container and component.
- A C4 container is a separately runnable or deployable unit that executes code or stores data, such as a web app, mobile app, service, database schema or message topic; it is not the same as a Docker container.
- System context and container diagrams are recommended for every team; component diagrams only where they add value, and code diagrams rarely, preferably generated from the code.
- Supplementary diagrams are the system landscape (all systems in an organisation), dynamic (runtime collaboration for one scenario) and deployment (containers mapped to infrastructure, one per environment).
- Every diagram needs a title with type and scope and a key; every element a name, type, description and (for containers and components) technology; every relationship a single direction, a specific label and, between containers, a protocol.

> - Modelling tools such as Structurizr generate many consistent diagrams from one model, while drawing tools produce independent pictures that drift apart.

Maps of your code

The C4 model is an approach to diagramming software architecture created by Simon Brown. It grew out of the software architecture training he gave from around 2006: according to the C4 site, its roots go back to somewhere between 2006 and 2009, the diagram types were named in early 2010 and the name "C4" was first used in early 2011. Brown had noticed that teams asked to draw their architecture produced diagrams that were hard to understand: unexplained shapes and colours, boxes that could be anything from a class to a data centre, unlabelled lines and several levels of detail mixed on one page. UML, which might have solved the problem, had largely fallen out of everyday use.

His answer was not a new notation but a small set of *abstractions* and a hierarchy of diagrams built on them, which he compares to the zoom levels of an online map. You start with the whole country, zoom into a city, then a district, then a street. Each level shows a different amount of detail for a different purpose, and you always know which level you are looking at. C4 is deliberately notation-independent and tool-independent: you can draw it on a whiteboard, in a drawing tool or with a modelling tool, provided you follow a few rules.

C4 has become one of the most widely used approaches to architecture diagrams because it is easy to learn, fits naturally with the multi-view thinking of Section 4.2, and produces diagrams that both developers and non-technical readers can follow. In 42010 terms, each C4 diagram type is a viewpoint with its own model kind; C4 just keeps the set small.

The abstractions: people, software systems, containers, components

C4 is built on four abstractions, plus code. A **person** is a human user of the system: a customer, a courier, a support agent. A software system is the highest level of abstraction, something that delivers value to its users and is typically owned and built by one team or organisation; the food-delivery platform is one, and so are the payment provider and the maps service it depends on.

A container is something that needs to be running for the system to work: in Brown's words, essentially a runtime boundary around some code that is executed or some data that is stored. Examples are a server-side web application, a single-page app running in a browser, a mobile app, a desktop application, a serverless function, a database schema, a file store or a message topic. Each is separately deployable or runnable, and containers communicate with each other across process boundaries, usually over a network.

A component is a grouping of related functionality behind a well-defined interface, living *inside* a container and running in the same process as the other components there. In a Java service it might be a set of classes behind an interface; in a Node.js service, a module. Components are not separately deployable; the container is the unit of deployment. Finally, **code** is the classes, functions and tables that implement a component.

WATCH OUT

A container is not a Docker container

The single most common misunderstanding of C4 is to read "container" as "Docker container". The C4 term predates Docker's popularity and means any separately runnable or deployable unit: a mobile app on a phone, a JavaScript application in a browser and a database schema are all containers even though none of them runs in Docker. Conversely, one C4 container may be deployed as twenty Docker containers behind a load balancer; how many copies run, and where, belongs on a deployment diagram, not a container diagram.

If the word causes confusion in your organisation, most teams keep it and explain it in the key or the glossary. The C4 site acknowledges the clash, notes that the two sometimes line up usefully, and says teams may change the terminology if they need to; if you do, change it everywhere.

The four core diagrams

Level	Scope	Shows	Main audience	Recommended?
1. System context	One software system	The system as a single box, the people who use it and the other software systems it interacts with	Everyone, technical and non-technical, inside and outside the team	Yes, for every team
2. Container	One software system	The containers (apps, services, data stores, topics) inside it, their responsibilities, technologies and how they communicate	Architects, developers, operations and support staff	Yes, for every team
3. Component	One container	The components inside it, their responsibilities and technologies, and their relationships	Architects and developers	Only where it adds value; generate it if it must be kept long-term
4. Code	One component	Classes, interfaces, functions or tables, typically as a UML class or entity-relationship diagram	Developers	Rarely; generate on demand from the code for the most complex parts

IF YOU ARE NEW TO THIS

Step by step: system, container or component?

The step most people trip over here is deciding which C4 abstraction a part of a system is. The names sound alike, so walk through the food-delivery platform from the outside in, asking of each part whether it can be deployed and run on its own.

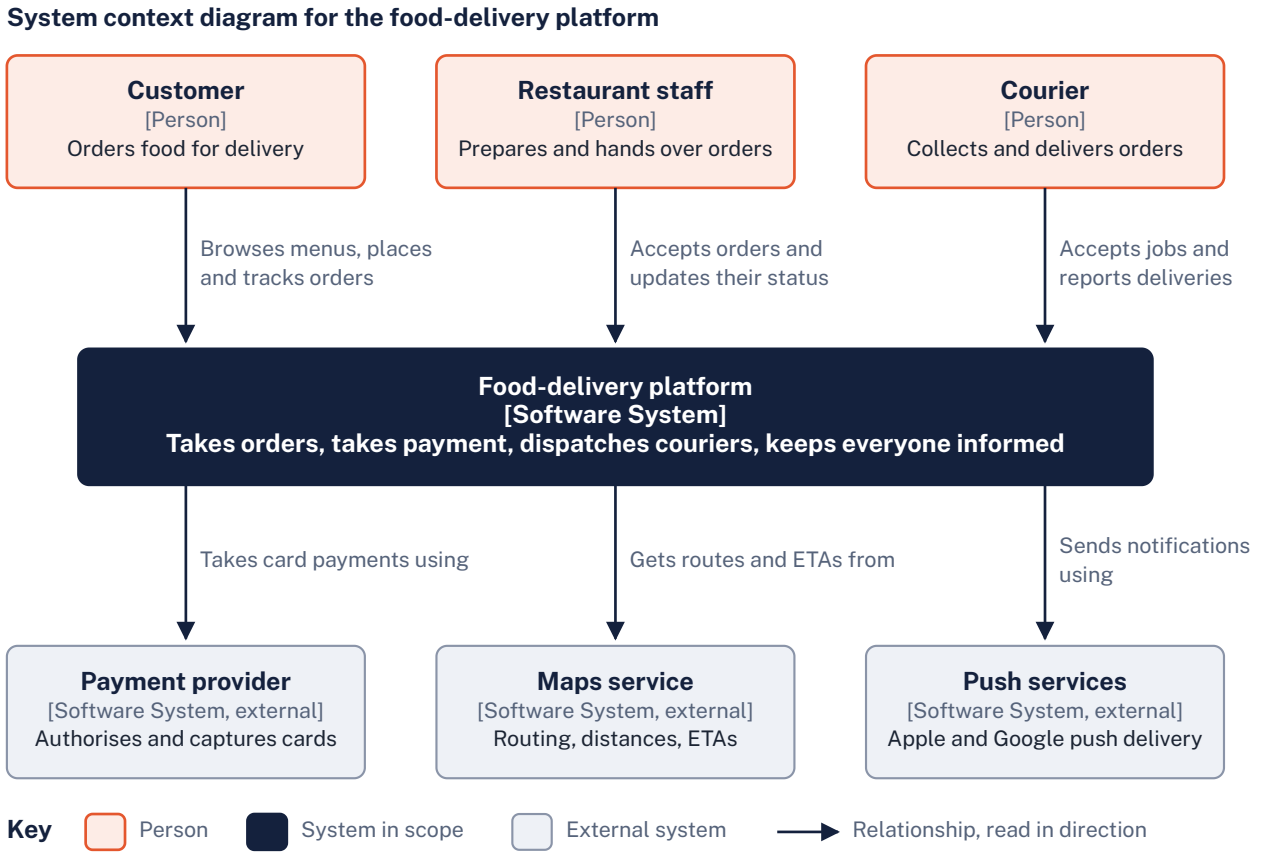
- Start at the top. The food-delivery platform as a whole is a software system: it delivers value to customers, restaurant staff and couriers. The payment provider is another software system, built and owned by a different organisation.
- Now look inside the platform. The ordering service runs as its own program and talks to the other parts over the network, so it is a container. So are the customer app on a phone and the restaurant app running in a tablet's browser.
- A container does not have to run your own code. The order database stores data, and the platform cannot work unless it is running, so it is a container too. The section's diagrams treat the order events topic, which holds messages, the same way.
- Zoom into the ordering service. Its order workflow, which applies the order rules, and its payment client, which calls the payment provider, are components. They run in the same process as the rest of the service and are deployed with it; the payment client cannot be deployed on its own.
- **The lesson:** a part that can be deployed and run by itself is a container; a part inside one, sharing its process, is a component; the whole that delivers value to its users, usually owned by one team or organisation, is a software system. Settle this before drawing, and every box lands at the right zoom level.

Level 1: the system context diagram

The system context diagram is the starting point. It puts the system you are describing in the middle as a single box and surrounds it with the people who use it and the software systems it depends on or that depend on it. It deliberately hides all internal detail, including technologies, protocols and internal services. It answers the questions every newcomer, product manager and security reviewer asks first: what is this system, who uses it, and what does it connect to?

Because it hides internals, the context diagram is also the right diagram for discussions with non-technical stakeholders and for scoping: where does our responsibility end? In the food-delivery example below it immediately shows that card payments, routing and push delivery are bought in, which is architecturally significant (each is an external dependency with its own availability, cost and contract) and invisible in most code-level views.

Food-delivery platform: system context



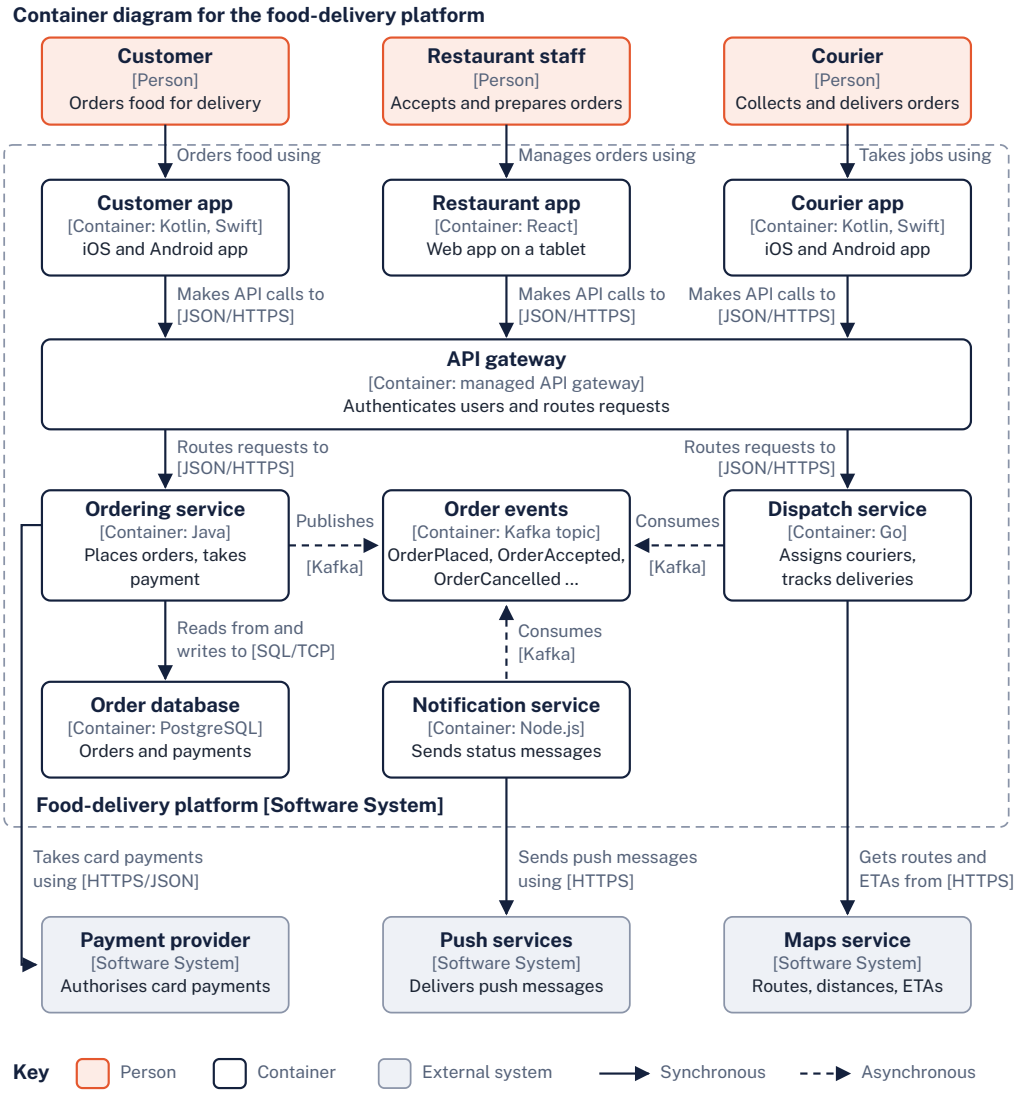
Level 1: the platform as one box, with its three kinds of users and three external systems; every element has a name, a type and a description, and every arrow is labelled.

Level 2: the container diagram

The container diagram zooms into the system box and shows the containers inside it: the applications and data stores that make up the system, what each is responsible for, which technology each uses, and how they communicate. For most teams this is the most valuable diagram they own. It shows the major structural decisions (how many deployable units, which communicate synchronously and which through messaging, which own data) and is concrete enough for developers and operations staff while still fitting on one page.

Notice two choices in the diagram below. The message topic for order events is drawn as a container in its own right, because it is a store that producers write to and consumers read from; the C4 guidance offers this explicit style or, alternatively, omitting the topic and writing "via order events topic" on the arrows. Drawing it shows the coupling through the topic clearly. And the asynchronous relationships use a dashed line, which the key explains. The container diagram says nothing about how many copies of each service run, in which region, or behind which load balancer; that belongs in a deployment diagram.

Food-delivery platform: containers



Level 2: the apps, services and data stores inside the platform boundary, with technologies on every element and protocols on every arrow.

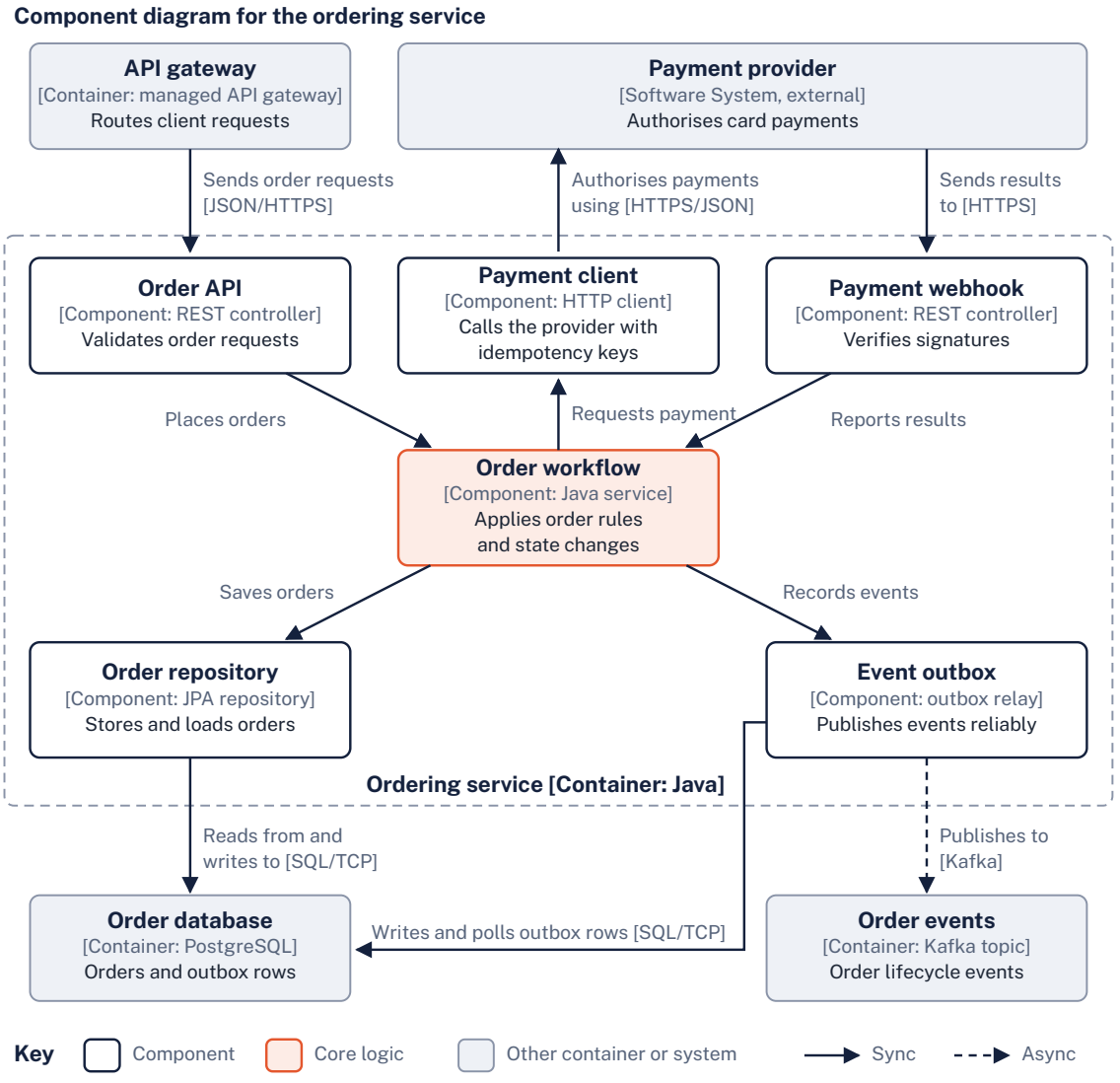
Level 3: the component diagram

The component diagram zooms into one container and shows the components inside it. It is aimed at architects and developers working on that container, and it is where the internal structure of a service becomes visible: which part handles incoming requests, where the business rules live, which part talks to the database, how events leave the service.

C4's guidance is to draw component diagrams only where they add value, typically for containers that are complex, risky or changed by many people, and to generate them automatically if they are to be kept for a long time, because component structure changes as often as the code does. For the food-delivery platform, the ordering service qualifies: it handles money, talks to an external provider and must publish events reliably. The diagram below

shows a typical structure, including an outbox component that writes events to the database in the same transaction as the order and relays them to Kafka afterwards (the outbox pattern is covered in Chapter 6).

Ordering service: components



Level 3: inside one container, with neighbouring containers and systems shown outside the dashed boundary so each relationship has both ends.

Level 4: code, and the supplementary diagrams

The code level zooms into a component and shows its classes, interfaces or tables, usually as a UML class diagram or an entity-relationship diagram. The C4 guidance treats it as optional and generally not worth drawing by hand: most development environments can generate it on demand, and a hand-drawn copy is out of date as soon as someone refactors. Draw it only for the most important or intricate parts, such as a domain model at the heart of the business (Section 4.4 shows such a class diagram).

Three **supplementary diagrams** fill the gaps the hierarchy leaves. A **system landscape diagram** is a context diagram without a single system in focus: it shows all the software systems and people in an organisation or domain, which is useful for enterprise and platform architects. A dynamic diagram shows how elements collaborate at runtime for one scenario, as numbered interactions on a container or component diagram (or as a UML sequence diagram); this is C4's equivalent of the scenarios in 4+1. A deployment diagram shows how containers are mapped onto infrastructure in one environment: deployment nodes such as cloud regions, Kubernetes clusters, virtual machines, managed database services and devices, with the number of instances. C4 recommends one deployment diagram per environment that matters, typically production and perhaps staging.

Notation rules that make diagrams readable

C4 does not prescribe shapes or colours, but it does prescribe what every diagram must say. These rules, condensed from the C4 notation guidance, are what make the diagrams above readable without a meeting to explain them.

Checklist: a C4 diagram that explains itself

- [ ] **A title stating type and scope**
  For example "Container diagram for the food-delivery platform", so readers know the zoom level at once.
- [ ] **A key (legend)**
  Explain every shape, colour, border style, line style and arrowhead used, even if the notation seems obvious. Readers from other teams will not share your conventions.
- [ ] **Every element has a name, a type and a description**
  The type in brackets (Person, Software System, Container, Component) removes ambiguity; one short sentence of responsibility says what the element is for.
- [ ] **Technology on every container and component**
  "Container: Java", "Container: PostgreSQL". Technology choices are architecture and readers need them.
- [ ] **Every relationship is one-directional and labelled**
  Use one arrow per relationship, labelled with what it does in the direction of the arrow ("Publishes order events to"), never just "uses" or a double-headed arrow.
- [ ] **Protocols on relationships between containers**
  JSON/HTTPS, SQL/TCP, Kafka. Synchronous versus asynchronous is one of the most important facts on the diagram.
- [ ] **Acronyms explained**
  Write them out or define them in the key or glossary.
- [ ] **One level of abstraction per diagram**
  No classes on a container diagram, no servers on a component diagram.
- [ ] **Colour as a supplement, not the only signal**
  Diagrams are printed in black and white and read by colour-blind people; use line style and labels as well, and keep colour meanings consistent across diagrams.

WORKED EXAMPLE

Worked example: improving a typical team diagram

A team in the food-delivery company shares this description of their whiteboard diagram: five boxes labelled "App", "Backend", "Orders", "DB" and "Stripe"; lines between them with no arrowheads; the Orders box is green and the others blue; there is no title or key. Applying the C4 rules step by step:

Title and level	Decide what the diagram is. It mixes a mobile app, a service and a database, so it is a container diagram. Title it "Container diagram for the food-delivery platform".
Name, type, description	"App" becomes "Customer app [Container: Kotlin, Swift] iOS and Android app for ordering food". "Backend" turns out to be the API gateway; "DB" becomes "Order database [Container: PostgreSQL]".
External systems	"Stripe" is a product name for an external system. Show it as "Payment provider [Software System]" outside the system boundary, with the vendor named in the description if that matters to readers.
Relationships	Replace each plain line with one labelled arrow: "Calls [JSON/HTTPS]", "Reads from and writes to [SQL/TCP]", "Authorises card payments using [HTTPS]". The team discovers that one line meant two different things (a call in one direction and a webhook in the other), so it becomes two arrows.
Missing elements	Labelling the arrows exposes that order events reach the dispatch service through a Kafka topic that was not drawn. Add the topic as a container, with dashed asynchronous arrows.
Key and colour	The green box turns out to mean "owned by our team". Add that to a key, or replace colour with a dashed boundary labelled with the team name.
Result	The diagram has gained a title, a key, three elements and twice as many arrows, yet takes less explanation, because every question a reader would ask is answered on the page.

Most of the value came from labelling the arrows: unlabelled lines hide the most important architectural facts (direction, protocol, synchronous or asynchronous).

The improved diagram exposed a dependency (the topic) that the team itself had forgotten, which is a typical side effect of applying the rules.

WATCH OUT

Common C4 mistakes

- Drawing every microservice as a separate software system when one team owns them all. C4's guidance is that services owned by one team are containers inside that team's system; services owned by separate teams can each be modelled as a software system.
- Showing a shared message bus or service mesh as one big container through which everything flows, which hides who actually depends on whom. Model the individual topics or queues, or write them on the arrows.
- Drawing component diagrams for every container by hand. They go out of date quickly; draw them selectively or generate them.
- Putting deployment details (replicas, load balancers, availability zones) on the container diagram instead of a deployment diagram.
- Arrows labelled only "uses" or "data", or arrows pointing both ways.
- Omitting people from the context diagram, which removes the reason the system exists.

Tools: drawing versus modelling

C4 diagrams can be produced with any drawing tool, and many teams start with a whiteboard or a general diagramming application. The C4 site distinguishes two families of tools. **Diagramming tools** draw boxes and lines; they are flexible and familiar, but the diagrams are independent pictures, so renaming an element means editing every diagram that shows it, and nothing checks that the diagrams agree. **Modelling tools** hold a single model of people, systems, containers, components and relationships, and generate each diagram as a view of that model; renaming happens once, diagrams stay consistent and the model can be queried and validated.

Structurizr, created by Simon Brown, is the reference modelling tool for C4. Its text-based Structurizr DSL describes the model and the views in a file that lives in version control next to the code. Other options include C4-PlantUML (C4 shapes for PlantUML), Mermaid's C4 diagram support and various commercial modelling tools; Section 4.7, Diagrams as code, shows examples. The choice matters less than the habit: keep the source of the diagrams where the team works, and update it in the same pull request as the change it describes.

Questions people ask

Is C4 a replacement for UML?
Not exactly. C4 defines abstractions and diagram types, not a notation, and it says nothing about many things UML can express precisely, such as state machines or detailed message sequences. Many teams use C4 for structure and borrow UML sequence, state and deployment diagrams where precision is needed (Section 4.4).

How do serverless functions fit into C4?
Each function, or each group of functions deployed together, is a container: a separately deployable unit that runs code. Shared data stores and queues they use are containers too.

Our system is a single deployable monolith. Is the container diagram pointless?
No. A monolith typically still has a web or mobile client, the server application, a database and perhaps a file store and a scheduled job, each a container. The component diagram of the monolith then becomes the most interesting view of its internal modules.

CHECK YOUR UNDERSTANDING

▼ **What is a container in the C4 model, and give two examples that do not involve Docker.**

A separately runnable or deployable unit that executes code or stores data. Examples: a mobile app on a phone, a single-page app in a browser, a database schema, a serverless function.

▼ **Which two C4 diagrams does the C4 guidance recommend for every team?**

The system context diagram and the container diagram.

▼ **Where should replica counts, regions and load balancers be shown?**

On a deployment diagram, one per environment, not on the container diagram.

▼ **Why does C4 insist on one-directional, labelled arrows?**

Unlabelled or two-way lines hide what the relationship is (call, data flow, event), which way it goes and which protocol it uses, which are among the most important architectural facts on the diagram.

▼ Try it: A diagram shows boxes "Web", "API", "Kafka", "Postgres" and "Payment" joined by arrowless lines, with no title or key. One box is red. Apply the C4 rules: what is wrong, and what would you change?

It has no title saying type and scope, and no key, so nobody knows what red means. Boxes need name, type, description and, for containers, technology. Payment is an external software system and should look different. Lines need a single direction, a specific label and a protocol, such as "publishes OrderPlaced [Kafka]". This is a container diagram, so call it one. Keep red only if the key explains it, and remove it otherwise.

SOURCES

- The C4 model for visualising software architecture (Simon Brown): https://c4model.com/
- C4 model: Container (not Docker!): https://c4model.com/abstractions/container
- C4 model: Notation: https://c4model.com/diagrams/notation
- C4 model: System context diagram: https://c4model.com/diagrams/system-context
- Structurizr DSL documentation: https://docs.structurizr.com/dsl

SECTION 4.4 OF 4.8 · INTERMEDIATE · 23 MIN READ · LAST REVIEWED NOVEMBER 2026

The UML diagrams architects still use

BEFORE YOU READ THIS

This section builds on:

- 0.6 APIs and integration
- 4.2 Views and viewpoints
- 4.3 The C4 model

Terms it uses from earlier: Interface, Synchronous call, Allocation view, Dependency inversion

THE IDEA IN BRIEF

Musicians all over the world can read the same sheet music, because the notes and symbols mean the same thing everywhere. UML is a similar shared set of symbols for drawing software: a particular arrow means "waits for an answer", another means "sends and carries on", a filled diamond means "is made of". You do not need to know all of it. Architects mostly use a handful of its drawings: one showing who talks to whom in what order, one showing the stages something goes through, one showing where everything runs, and a few others, each used only when its precision really helps.

WHAT YOU NEED TO KNOW

- UML is a general-purpose modelling language standardised by the OMG; the current version is UML 2.5.1 (December 2017), which defines fourteen diagram types in structure and behaviour families.
- Fowler distinguishes UML as sketch (selective, informal communication), blueprint (complete, precise models) and programming language (executable models); architects mostly sketch.
- Component diagrams show provided (ball) and required (socket) interfaces; deployment diagrams show nodes, artifacts and communication paths.
- Sequence diagrams show lifelines and activations; a filled arrowhead is a synchronous call, an open arrowhead on a solid line an asynchronous message, and a dashed line a reply; fragments such as alt, opt and loop add control flow.
- State machine diagrams describe lifecycles with states and transitions labelled event [guard] / action; activity diagrams describe workflows with decisions, forks, joins and swimlanes.

- Class diagrams are best kept for small domain models with associations, multiplicities and composition; full class diagrams should be generated, not hand-drawn.
- Precision is worth paying for at interfaces between organisations, in regulated and safety-critical work, for lifecycles shared by several teams and for distributed development.

What UML is, and where it stands today

The Unified Modeling Language (UML) is a general-purpose graphical language for describing the structure and behaviour of software. It grew out of the merger, at Rational Software in the mid-1990s, of the methods of Grady Booch, James Rumbaugh and Ivar Jacobson, and was adopted as a standard by the Object Management Group (OMG), an industry standards consortium, in 1997. UML 2.0 (2005) was a major revision. The current version is UML 2.5.1, published by the OMG in December 2017; the earlier UML 2.4.1 was also published by ISO as ISO/IEC 19505 (2012). The specification defines fourteen diagram types, divided into structure diagrams (such as class, component and deployment diagrams) and behaviour diagrams (such as activity, state machine and sequence diagrams).

UML's reputation has changed considerably. In the late 1990s and early 2000s it was promoted as the basis of model-driven development, in which detailed models would be transformed into code. That ambition largely failed outside specialist domains, and many teams associate UML with heavyweight processes and diagrams nobody read. When Marian Petre interviewed 50 professional software developers for a study presented at ICSE 2013, 35 said they did not use UML at all and most of the others used it selectively and informally; most teams draw informal boxes and lines rather than strict UML. Yet a handful of UML diagram types remain the best tools available for particular questions, especially questions about behaviour over time, and their notation is widely understood. An architect does not need all fourteen diagram types; they need perhaps six, used deliberately.

This section covers those six, each with a small example from the food-delivery platform used throughout this chapter. It assumes the C4 vocabulary from Section 4.3; the two combine well, with C4 for the structural overview and UML where precision about interfaces, interactions or states pays for itself.

Sketch, blueprint or programming language

Martin Fowler's *UML Distilled* (3rd edition, Addison-Wesley, 2003) and his essays of the same year distinguish three ways of using UML, a classification Steve Mellor arrived at independently. As a **sketch**, UML is used informally and selectively to communicate ideas: on a whiteboard before writing code (forward sketching) or to explain part of an existing system

(reverse sketching). Only the important elements appear, the notation is followed loosely, and the diagram is often thrown away. Fowler's own summary is “comprehensiveness is the enemy of comprehensibility”.

As a **blueprint**, UML is used to produce complete, precise models that someone else implements, or that are generated from existing code to document it in full; the point is completeness, and tools are usually needed. As a **programming language**, the UML model is detailed enough to be compiled into executable code, the goal of model-driven architecture (MDA). This mode survives in some embedded and safety-critical niches, often via specialised profiles and tools, but has not become mainstream.

For architects, sketch mode is the default and blueprint mode the exception. The useful question for each diagram is: who will read it, and what decision or understanding does it support? A sequence diagram sketched to agree how payment failures are handled is valuable even if it ignores half the notation. A full class model of a 200-class service, maintained by hand, is usually not.

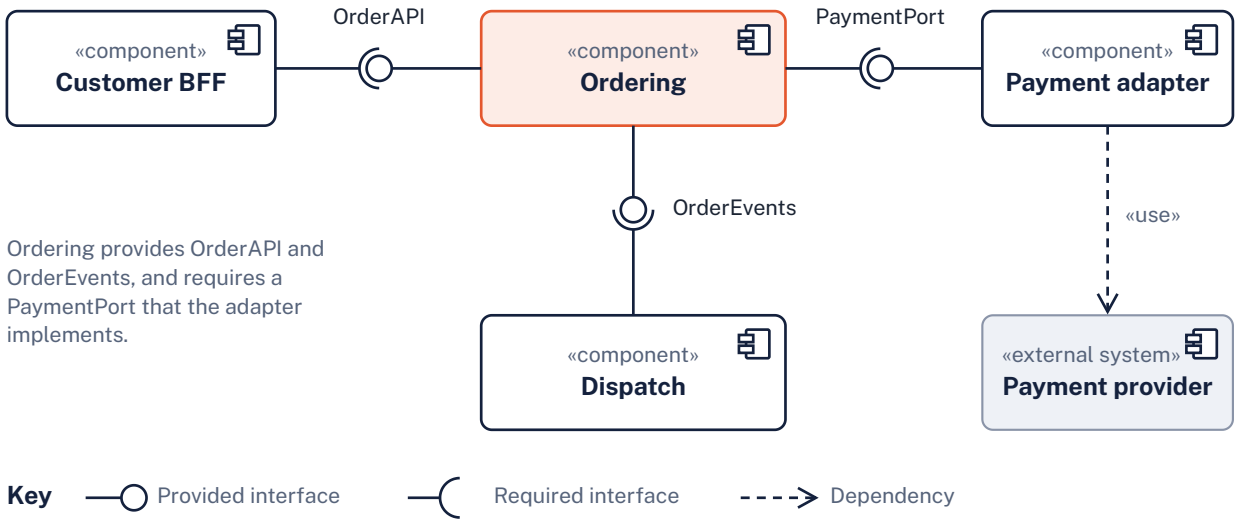
Component diagrams: parts and their interfaces

A UML component diagram shows modular parts of a system and, above all, their **interfaces**. Its distinctive notation is the "ball and socket": a small circle on a stick (a provided interface, which the component offers) and a half-circle (a required interface, which it needs someone else to provide). When a ball sits in a socket, the two components are wired together through that interface. Dependencies are dashed arrows, optionally labelled with a stereotype such as «use».

This is exactly what architects care about when they want to show dependency direction and replaceability. In the example, the ordering component does not depend on the payment provider directly; it requires a `PaymentPort` interface, which a separate payment adapter implements. That is the dependency inversion and ports-and-adapters thinking of Chapter 3 made visible: swapping payment providers means writing a new adapter, not changing ordering. A C4 component diagram can show the same parts, but it does not distinguish provided from required interfaces.

Component diagram

Component diagram: ordering and its interfaces



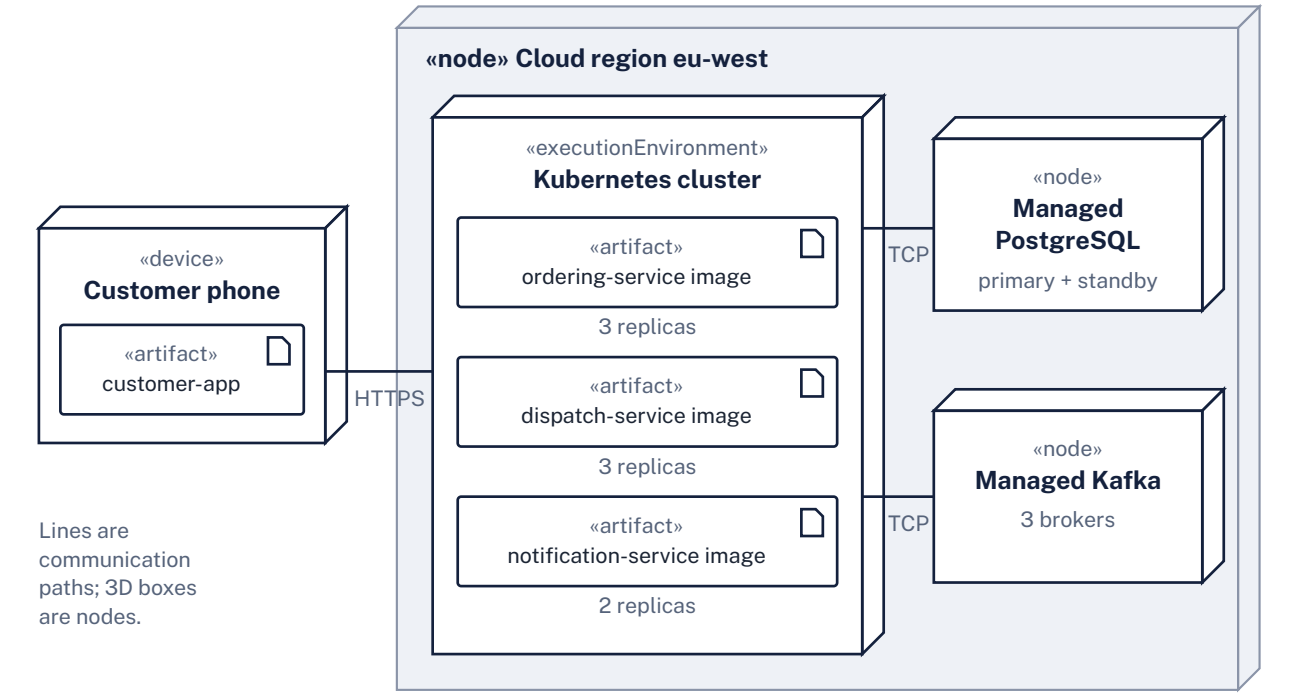
Balls are interfaces a component provides; sockets are interfaces it requires. Ordering depends on a port, not on the provider. (BFF: backend for frontend, a server-side component dedicated to one client app.)

Deployment diagrams: nodes and artifacts

A UML deployment diagram shows the physical or virtual infrastructure and what runs on it. Its elements are **nodes**, drawn as three-dimensional boxes, which are computational resources: a «device» such as a phone or server, or an «executionEnvironment» such as an operating system, a container runtime or a Kubernetes cluster, and nodes can be nested. **Artifacts**, drawn as rectangles with a document icon and the «artifact» stereotype, are the physical files deployed: a JAR, a container image, a mobile app bundle, a configuration file. Lines between nodes are **communication paths**, labelled with protocols.

Deployment diagrams are allocation views in the SEI's terms (Section 4.2). They answer the operations team's questions: where does each part run, how many copies, in which region, which network paths exist. C4's deployment diagram covers the same ground with less formal notation; either is fine as long as there is one per important environment and it shows replicas, managed services and network boundaries.

Deployment diagram



Nodes are where things run; artifacts are what is deployed; lines are communication paths.

Sequence diagrams: who calls whom, in what order

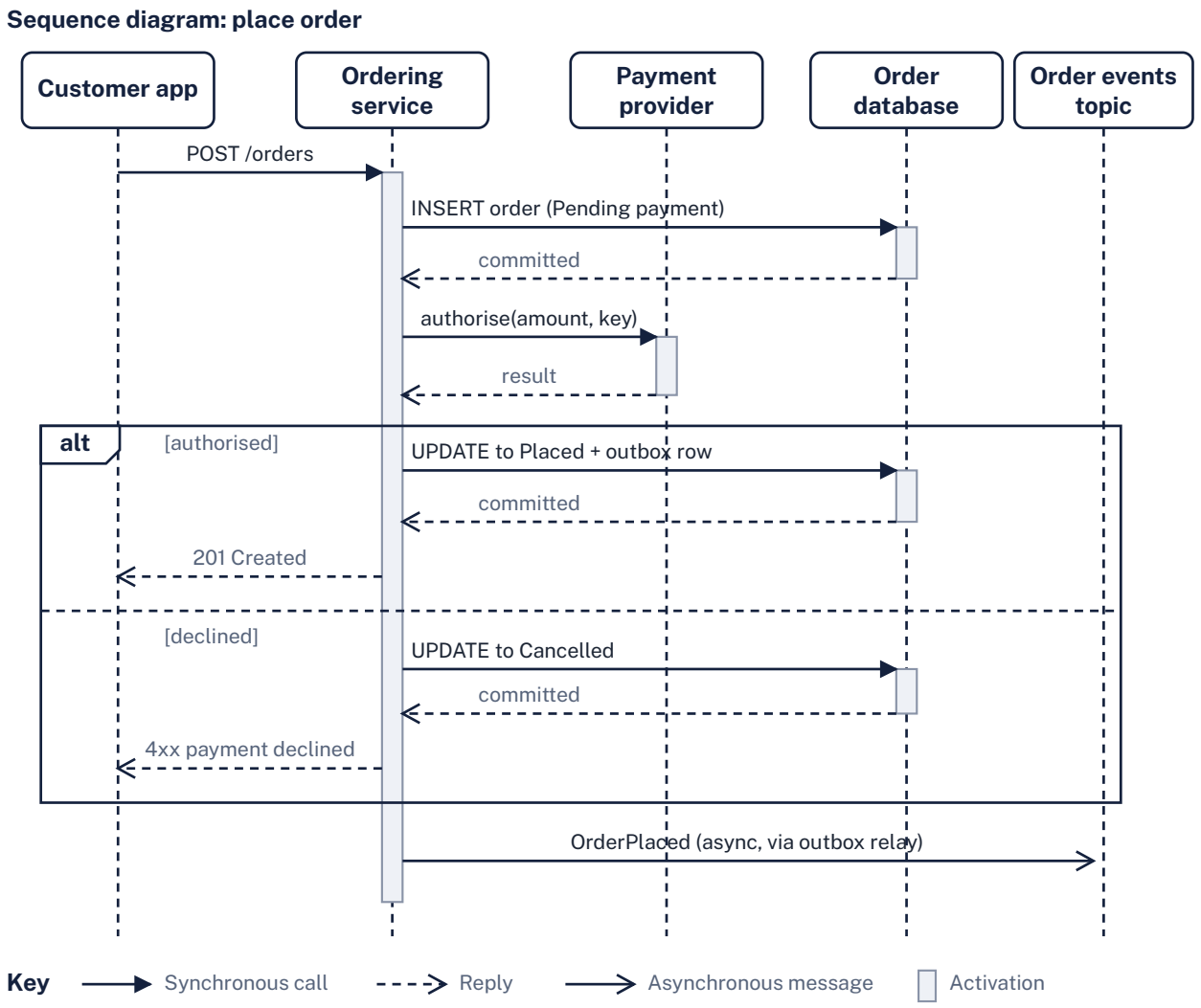
If architects keep only one UML diagram, it is usually the sequence diagram. It shows how participants interact over time for one scenario. Participants appear across the top, each with a dashed vertical lifeline; time runs downwards. A thin rectangle on a lifeline, the **activation** (execution specification), shows when that participant is busy handling a message.

The arrowheads carry meaning. A **synchronous message**, where the sender waits for an answer, has a solid line and a filled arrowhead. A **reply** has a dashed line and an open arrowhead. An **asynchronous message**, where the sender carries on without waiting, has a solid line and an open arrowhead. Getting these right matters architecturally: whether the ordering service waits for the payment provider or publishes and moves on decides its latency, its failure modes and its coupling (see Section 0.6). The order of the steps matters too: the diagram below saves the order before it calls the provider, so a crash between the two can never leave a customer charged with no record of the order, and the idempotency key makes a retried authorisation safe.

Combined fragments add control flow: a box labelled `alt` with guarded sections separated by a dashed line shows alternatives (if/else), `opt` shows an optional part, `loop` a repetition and `par` parallel sections. Use them sparingly; a sequence diagram with nested loops and

alternatives is code drawn badly. Draw one diagram per important scenario, including the failure scenarios, which are where most of the architectural thinking happens.

Sequence diagram



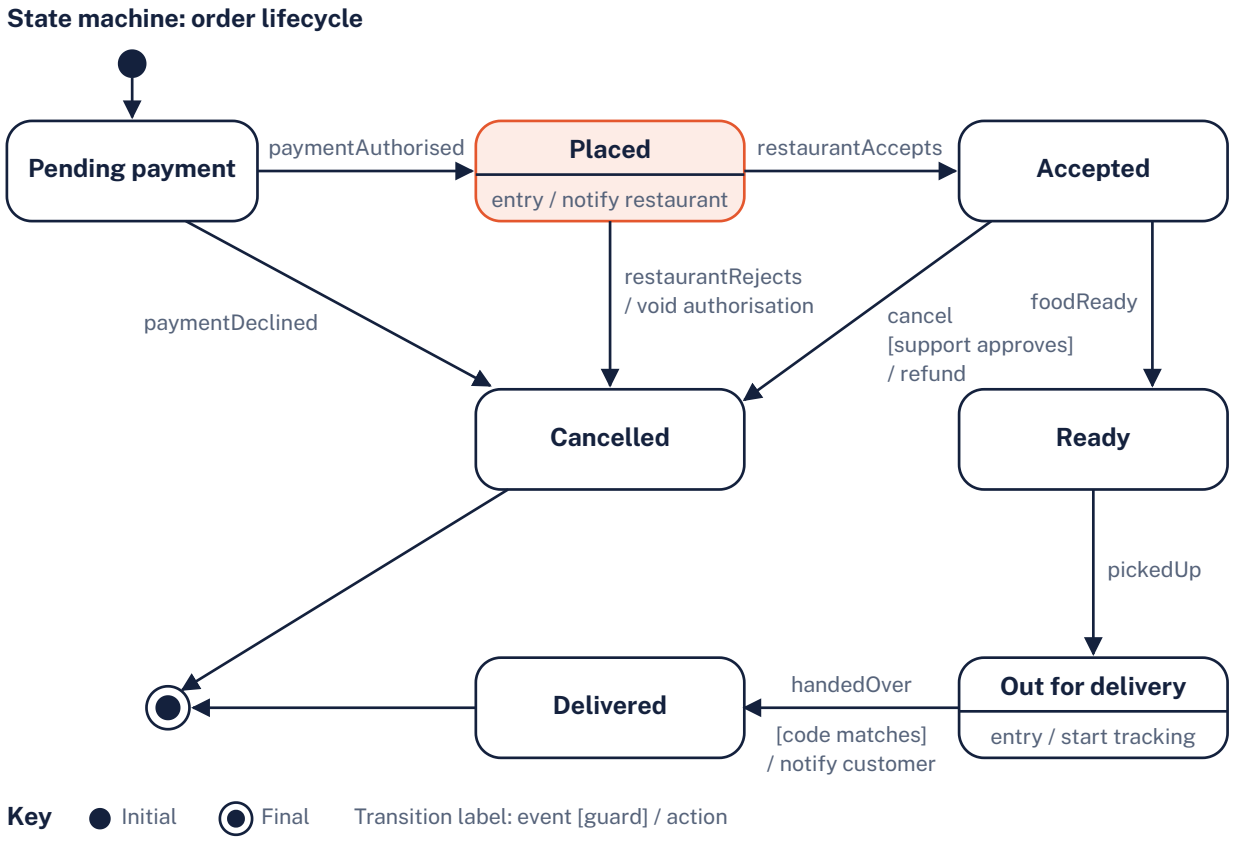
Placing an order safely: save the order first, authorise the card with an idempotency key, then update the order. An alt fragment shows the two outcomes and an asynchronous event ends the flow.

State machine diagrams: lifecycles and rules

Many business objects have a lifecycle: an order, a payment, a subscription, an insurance claim. A state machine diagram describes it precisely. **States** are rounded rectangles; a filled circle marks the **initial** pseudo-state and a bullseye the **final** state. **Transitions** are arrows labelled `event [guard] / action`: the event that triggers the transition, an optional guard condition that must be true, and an optional action performed on the way. States can have internal behaviours such as `entry / notify restaurant`.

State machines are architectural more often than people expect. The order lifecycle below is shared by several services: ordering owns it, dispatch and notification react to its transitions, and support staff can trigger some of them. Agreeing on the states and the legal transitions, in one diagram, prevents a whole class of bugs in which one service thinks an order is cancelled while another dispatches a courier. The diagram also exposes questions that text hides: can an order be cancelled once it is out for delivery? (Here, no: there is no transition from that state to Cancelled.) Is a refund an action on a transition or a separate lifecycle? Money follows the states too: the card is only authorised when the order is placed and is captured when the restaurant accepts, so a rejection voids the authorisation, and only a later cancellation needs a refund.

State machine diagram



The order lifecycle: every legal transition, with events, guards and actions, on one page.

IF YOU ARE NEW TO THIS

Step by step: reading a transition label

This section's hardest idea is the small language written on each arrow of a state machine diagram: event [guard] / action. Follow one order through the section's order lifecycle and read each label in turn.

- A new order starts in Pending payment, where the filled circle points. It stays there until an event arrives that one of its outgoing arrows names.
- The event paymentAuthorised arrives, and its arrow leads to Placed. That label is only an event: no condition and no action. On arrival, Placed's entry behaviour runs, notify restaurant, so the restaurant hears about the order.
- The event restaurantAccepts moves the order to Accepted. Now a cancel request comes in. The arrow from Accepted reads cancel [support approves] / refund: cancel is the event, the guard in square brackets is the condition that must be true, and refund, after the slash, is the action carried out on the way.
- If support has not approved, the guard is false: the transition does not fire, and the order simply stays Accepted.
- If support has approved, the transition fires: the refund is made on the way, and the order ends up in Cancelled.
- Had the order already been Out for delivery, no arrow labelled cancel leaves that state, so the request could not cancel it. A missing arrow is a rule too.
- **The lesson:** read every arrow as 'when this event happens, and if this condition is true, do this and move there'. Agreeing these labels in one diagram helps several services keep to the same rules for an order.

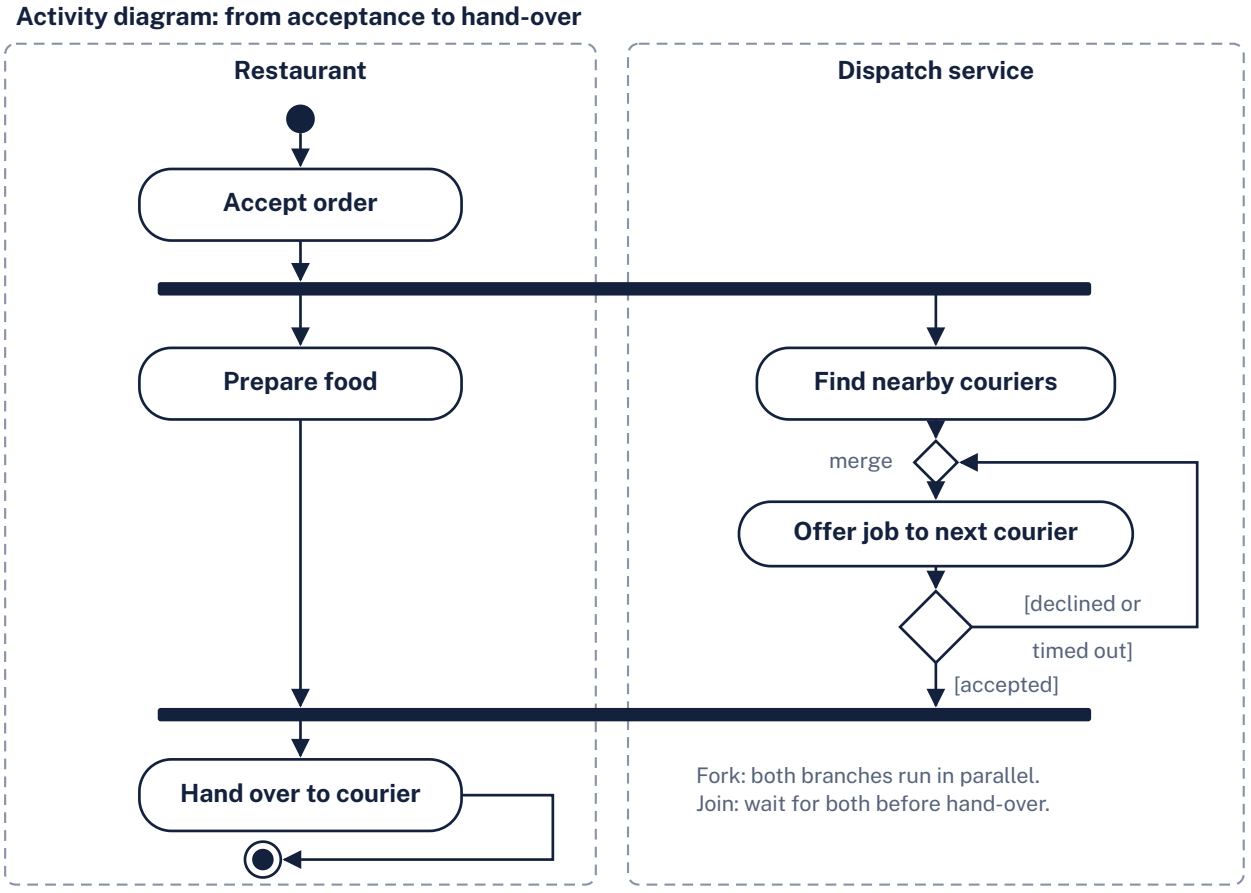
Activity diagrams: workflows and parallelism

An activity diagram is a flowchart with precise semantics. Rounded rectangles are **actions**; diamonds are **decisions** (with guards on the outgoing flows) and merges; thick bars are **forks**, which split one flow into parallel flows, and **joins**, which wait for all incoming flows before continuing. **Partitions**, often called swimlanes, show who or what performs each action. A loop must flow back through a merge diamond: two flows drawn straight into one action would mean that the action waits for both, like a join.

Architects use activity diagrams for business processes and workflows that span several parts of a system, especially where parallelism and responsibility matter. The example shows that food preparation and courier assignment run in parallel, that assignment loops until a courier accepts, and that hand-over waits for both. That raises an architectural question immediately:

what if no courier accepts within twenty minutes while the food is ready? The diagram makes the gap visible; the answer might be a timeout transition, a compensation step, or a change to the order lifecycle. Long-running workflows of this kind are often implemented as sagas or orchestrated workflows, covered in Chapter 6.

Activity diagram



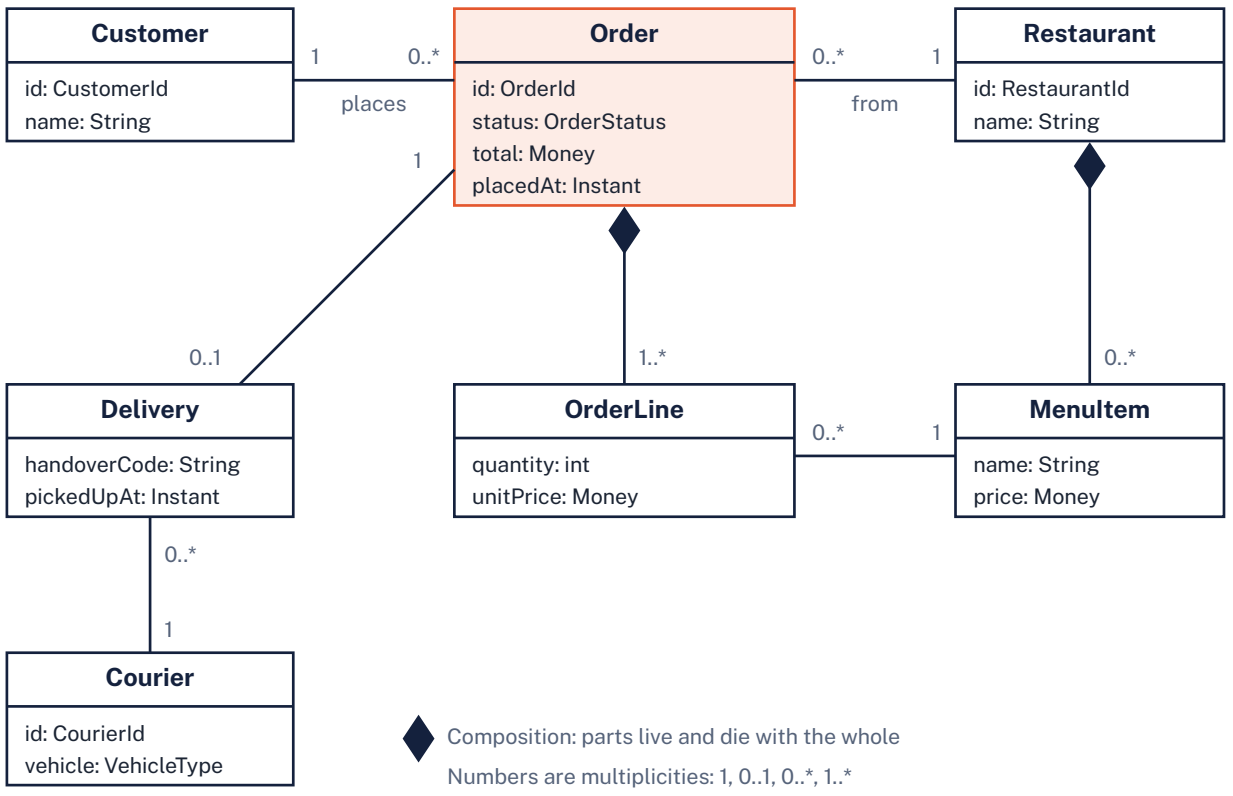
Food preparation and courier assignment run in parallel between a fork and a join; the decision loops until a courier accepts.

Class diagrams, used sparingly: the domain model

The class diagram is the best-known UML diagram and the one architects should use most sparingly. Drawing every class of a service by hand is the blueprint mode at its least useful: it is out of date within weeks and the code says the same thing more precisely. Where class diagrams earn their place is the domain model: a small diagram of the key business concepts, their attributes and their relationships, which developers, product managers and domain experts can discuss together. In domain-driven design terms (Chapter 5) it captures the ubiquitous language of one bounded context.

The essentials are few. A class box has a name, attributes and optionally operations. A plain line is an **association**, optionally named, with multiplicities at each end (`1`, `0..1`, `0..*`, `1..*`). A filled diamond marks **composition**: the parts belong to exactly one whole and do not outlive it, as order lines belong to an order. A hollow triangle marks generalisation (inheritance) and a dashed line with an open arrowhead a dependency. Multiplicities carry real design decisions: `Order 1 to 0..1 Delivery` says an order has at most one delivery, which rules out splitting a large order between two couriers. If the business might want that, the diagram is where the question surfaces.

Class diagram of the domain model



A domain model small enough to discuss with the business: key concepts, attributes, associations and multiplicities.

The six diagrams at a glance

Diagram	Answers	Key notation	Typical use by architects
Component	What are the parts and through which interfaces do they depend on each other?	Component boxes, provided (ball) and required (socket) interfaces, dependencies	Showing dependency direction and replaceable parts; ports and adapters
Deployment	Where does each part run, how many copies, connected how?	Nodes (3D boxes), artifacts, communication paths	Production and staging infrastructure, network zones
Sequence	In what order do participants interact for this scenario?	Lifelines, activations, sync and async messages, replies, fragments	Key runtime scenarios, especially failures, retries and integrations
State machine	Which states can this object be in, and what moves it between them?	States, initial and final states, transitions with event [guard] / action	Lifecycles shared between services: orders, payments, claims
Activity	What is the workflow, who does each step, what runs in parallel?	Actions, decisions, forks and joins, partitions	Business processes, long-running workflows, sagas
Class	What are the key concepts and how are they related?	Classes, associations, multiplicities, composition, generalisation	Small domain models; generated diagrams of code

When UML is overkill, and when precision matters

UML is overkill when the audience does not know it and the question does not need it: a C4 context diagram communicates better to executives than a UML use case diagram, and a whiteboard sketch is the right tool for a design discussion that will be over in an hour. It is overkill when diagrams are maintained by hand that could be generated, and when strict notation is enforced in a sketch whose purpose is to explore. Fowler's observation that the precision added in UML 2 mostly serves blueprint and programming-language users, and burdens sketchers, still holds.

Precision pays when misunderstanding is expensive. **Interfaces between companies** are the clearest case: when the food-delivery company integrates with a payment provider or a large restaurant chain's ordering system, each side builds against the other's description and discovers mismatches only in testing or production. A precise sequence diagram of the payment flow, including timeouts, retries, idempotency keys and webhooks, attached to the

interface specification, is cheap insurance. **Regulated and safety-critical industries** (medical devices, rail, aviation, automotive, some financial systems) often require design documentation that auditors can trace from requirements to implementation; formal UML or its systems-engineering relative SysML is common there. **Concurrency and lifecycle rules** shared by several teams, like the order states above, benefit from a notation that forces every transition to be stated. And **outsourced or distributed development**, where the people implementing a design cannot easily ask its author, needs descriptions that leave less to interpretation.

WATCH OUT

Common mistakes with UML

- Using the wrong arrowheads in sequence diagrams, so that an asynchronous event looks like a blocking call, or the reverse. Readers draw architectural conclusions from them.
- Drawing whole-system class diagrams by hand. Generate them if needed, and draw only the domain model.
- Omitting the key because "it is standard UML". Many readers do not know the notation; a two-line key costs nothing.
- One giant sequence diagram with every branch. Draw one diagram per scenario, and make the failure scenarios separate diagrams.
- Insisting on full UML in a sketch, which slows discussion and puts people off diagramming altogether.

Questions people ask

Should our team learn UML at all?
Learn the notation essentials of the six diagrams in this section; it takes a few hours and gives the team a shared vocabulary for interfaces, interactions and lifecycles. There is little need to learn the other diagram types or the full specification unless you work in a domain that requires them.

How does UML relate to C4?
They are complementary. C4 provides a small set of abstractions and a hierarchy of structural diagrams; UML provides precise notations, especially for behaviour. Many teams use C4 for context, container and component views and UML for sequence, state and occasionally deployment diagrams.

Can I draw UML with text instead of a drawing tool?
Yes. PlantUML and Mermaid both generate sequence, state, class and activity diagrams from short text files that can live in version control. Section 4.7 shows examples.

CHECK YOUR UNDERSTANDING

▼ In a sequence diagram, how do you tell a synchronous call from an asynchronous message?

A synchronous call has a solid line with a filled arrowhead (the sender waits); an asynchronous message has a solid line with an open arrowhead (the sender continues). Replies are dashed lines with open arrowheads.

▼ What does the label `handedOver [code matches] / notify customer` on a transition mean?

When the handedOver event occurs and the guard "code matches" is true, the transition fires and the action "notify customer" is performed.

▼ Why should architects rarely hand-draw class diagrams of whole services?

They duplicate what the code states more precisely and go out of date quickly; small domain models are worth drawing, and full diagrams can be generated when needed.

▼ Give two situations in which precise UML is worth the effort.

For example, interfaces between companies, regulated or safety-critical development, lifecycles shared by several teams, or outsourced development where implementers cannot easily ask the designers.

▼ Try it: Three food-delivery teams keep disagreeing about when an order can be cancelled. Which UML diagram would you draw? Give one example transition in the proper notation, and say why more precision is worth paying for here.

A state machine diagram, because the argument is about one object's lifecycle. Example: Confirmed to Cancelled on cancelRequested [kitchen has not started] / refundPayment, following event [guard] / action. A sequence diagram would show one path, but not all the allowed states. Precision is worthwhile because several teams share this lifecycle and mistakes cost real money. A careful sketch of a dozen states settles arguments without needing a full UML model of the system.

SOURCES

- UML specification versions (Object Management Group): https://www.omg.org/spec/UML/
- About the Unified Modeling Language 2.5.1 (OMG): https://www.omg.org/spec/UML/2.5.1/About-UML/
- UmlMode (Martin Fowler): https://martinfowler.com/bliki/UmlMode.html

- UmlAsSketch (Martin Fowler): https://martinfowler.com/bliki/UmlAsSketch.html

Architecture Decision Records

BEFORE YOU READ THIS

This section builds on:

- 0.8 How software gets built
- 1.2 Architecture versus design
- 1.6 Decisions and trade-offs

Terms it uses from earlier: Architecturally significant, Quality attribute, Trade-off, Pull request

THE IDEA IN BRIEF

Imagine moving into a house and finding a wall that seems to serve no purpose. Do you knock it down? If the builder had left a note saying "this wall holds up the bathroom upstairs", you would know. An Architecture Decision Record is that note for software: a short, dated page explaining one important choice, the alternatives that were considered and what the choice costs. Notes are never rewritten; if the owners change their minds, they add a new note saying what changed and why.

WHAT YOU NEED TO KNOW

- An Architecture Decision Record (ADR) captures one significant decision with its context, the options considered, the decision and its consequences; Michael Nygard popularised the lightweight form in a 2011 blog post.
- Nygard's format has a title, context, decision, status and consequences; context is written neutrally, the decision in the active voice ("We will...") and the consequences include costs as well as benefits.
- MADR (version 4.0.0, 2024) adds decision drivers, considered options with pros and cons, and confirmation; a Y-statement compresses a decision into one sentence: in the context of, facing, we decided for, and neglected, to achieve, accepting that.
- Records move from proposed to accepted or rejected, and later to deprecated or superseded; an accepted record's content is never rewritten, because a new superseding record preserves the history.
- ADRs live in the code repository as numbered Markdown files (for example docs/decisions/0012-title.md), are reviewed in pull requests and are indexed in a decision log.

- Write an ADR for decisions that are costly to reverse, affect key qualities, constrain others or were contested; avoid records written after the fact, with no real alternatives or with only positive consequences.

What an ADR is and the problem it solves

An Architecture Decision Record (ADR) is a short text document that captures one architecturally significant decision: the situation that forced it, the options that were weighed, what was chosen and what follows from the choice. A collection of them, kept in order, forms the decision log of a system. The idea of recording decisions is old (design rationale research goes back decades, and Jeff Tyree and Art Akerman published a detailed decision template in *IEEE Software* in 2005), but the lightweight form most teams use today comes from Michael Nygard's blog post *Documenting Architecture Decisions*, published on 15 November 2011.

Nygard's starting point was a problem every long-lived team recognises. A new developer finds a decision in the code that looks strange: an extra queue, a duplicated table, a library nobody would choose today. Without knowing why it was made, the newcomer has two bad choices. They can accept it blindly, preserving something whose reasons may no longer apply, or change it blindly, breaking something whose reasons still do. Nygard's remedy was to write down the reasons at the time the decision is made, in a small file kept with the code, so that the motivation travels with the system. Big architecture documents rarely achieve this because nobody updates them; a short record of one decision, written once and never rewritten, stays true because it only claims to describe what was known and decided on a particular date.

This builds directly on the anatomy of a decision in Chapter 1: decisions and trade-offs: context, options, criteria, decision with rationale, consequences and status. An ADR is simply that anatomy written down in a predictable format. It also answers the question architecture versus design leaves open in practice: if architecture is the set of significant decisions, the ADR log is the closest thing a team has to a list of its architecture.

Nygard's format

Nygard proposed five parts. Most teams still use them, often moving the status to the top so that readers see at once whether the record is in force. His guidance on how to write each part matters as much as the headings.

The five parts of a Nygard-style ADR

Part	What it contains	How to write it well
Title	A short noun phrase naming the decision, with a sequence number, such as “ADR 12: Order events through a message broker”.	Name the decision, not the problem: a reader scanning the log should learn what was decided from the title alone.
Context	The forces at play: the problem, the relevant quality requirements and constraints, technical and organisational facts, and the tension between them.	Nygard asks for value-neutral language: describe facts and forces, not the preferred answer. Include numbers where they exist (peak load, targets, incident dates).
Decision	The response to those forces.	Full sentences in the active voice, beginning “We will…”, so that there is no doubt about what was decided and by whom.
Status	Proposed, accepted, and later possibly deprecated or superseded, with a reference to the replacing record.	Keep it current; it is the one field that changes after acceptance.
Consequences	The resulting context after the decision is applied.	List all consequences, not only the positive ones: what becomes easier, what becomes harder, new risks, new work and options that are now closed.

Nygard also suggested that a record should fit on one or two pages, that records are numbered sequentially and numbers are never reused, and that they live in the project repository as plain text. Each of those small rules exists for a reason: short records get read, stable numbers can be cited in code comments and commit messages, and plain text in the repository can be reviewed and versioned like code.

Other templates: MADR, Y-statements and richer forms

Nygard's format says nothing explicit about the options that were rejected, and many teams found that the alternatives are exactly what a later reader most wants to know. Several templates add them.

MADR (Markdown Architectural Decision Records, maintained at adr.github.io) is the most widely used structured template. Its current release at the time of writing is version 4.0.0, published in September 2024. The full template has a title, then *Context and Problem Statement*, optional *Decision Drivers*, *Considered Options*, *Decision Outcome* with optional

Consequences and *Confirmation* subsections, optional *Pros and Cons of the Options* and optional *More Information*. Metadata such as status, date, decision-makers, and who was consulted and informed can go in a YAML header. Two ideas in MADR are worth borrowing even if you use another format. Decision drivers make the criteria explicit before the options are judged, which is the weighted-matrix discipline from Chapter 1 in compact form. *Confirmation* says how the team will check that the decision is actually implemented and still holds, for example a review, a test or an automated architecture check.

The Y-statement, proposed by Olaf Zimmermann and colleagues and first presented at the SATURN conference in 2012, compresses a decision into one structured sentence: *In the context of* (a use case or component), *facing* (a concern or quality requirement), *we decided for* (the option) *and neglected* (the alternatives), *to achieve* (the benefits), *accepting that* (the drawbacks). The name is a pun on "why". A Y-statement is ideal for a decision log summary, a slide, a code comment or a pull request description; for a contested one-way door it is usually too compressed, and teams expand it into a full record.

Heavier templates exist for regulated or very large settings. Tyree and Akerman's 2005 template adds fields such as assumptions, constraints, related decisions, related requirements and implications. These are useful when decisions must be traced to requirements for an auditor; for most product teams they add effort that nobody reads.

Choosing a template

Template	Size	Records rejected options?	Best for
Nygard (2011)	Half a page to two pages	Only if you put them in the context or consequences	Teams starting out; the smallest habit that works.
MADR 4.0.0 (2024)	One to three pages	Yes: considered options, with optional pros and cons of each	Most product teams; contested decisions where alternatives must be visible.
Y-statement (2012)	One sentence	Yes, in the *neglected* clause	Decision log summaries, small decisions, pull request descriptions, slides.
Tyree and Akerman (2005)	Several pages	Yes, plus assumptions, constraints and traceability	Regulated environments needing traceability to requirements.

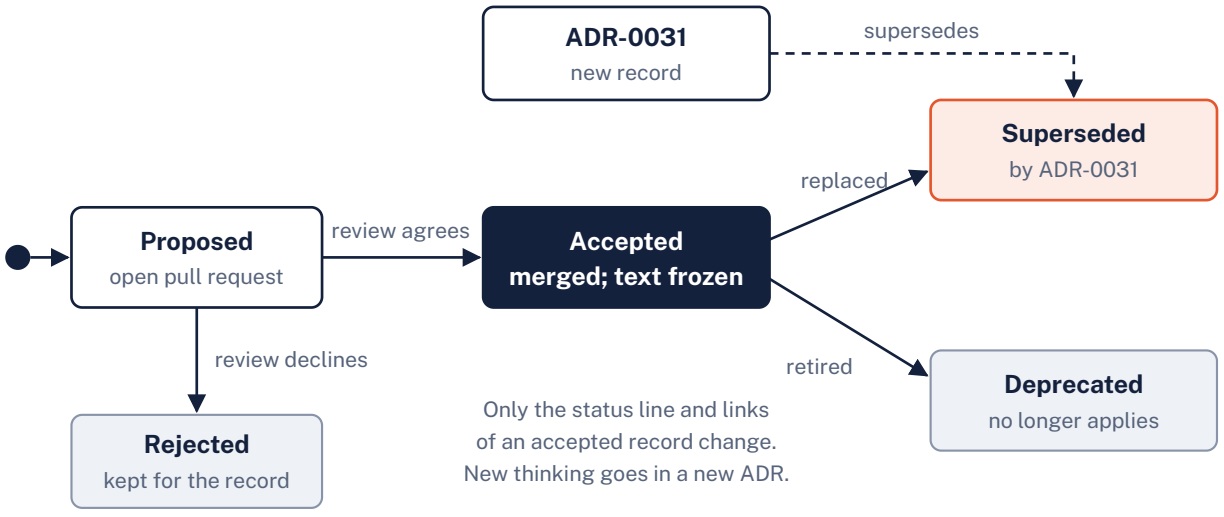
Status, lifecycle and why accepted records are not edited

An ADR moves through a small number of states. It starts as **proposed**, usually as a pull request that adds the file, so that discussion happens in review comments. When the people accountable agree, it is merged as **accepted**. A proposal that is turned down can be kept as

rejected, which is valuable: the next person who suggests the same idea finds the earlier reasoning. Later, an accepted record may become **deprecated** (the decision no longer applies, perhaps because the component was retired) or **superseded** by a newer record that replaces it, with links in both directions. MADR lists the same set: proposed, rejected, accepted, deprecated and superseded.

The central rule is that an accepted record's content is not rewritten. If the team changes its mind, it writes a new ADR that supersedes the old one, and changes only the old record's status line to point forward. The reasons are practical. First, the record is evidence of what was known and decided at a point in time; editing it after the fact turns history into fiction and destroys the trust that makes the log useful. Second, the old record explains why the system looked the way it did for a period, which matters when debugging old data or reading old code. Third, a new record forces the team to state what changed in the context, which is often the most important insight. Small corrections (a typo, a broken link) are harmless; changing the decision, the reasons or the consequences is not. Martin Fowler's description of the practice says the same: once accepted, a record is not reopened; a superseding record is written instead.

The ADR lifecycle



Records are proposed and reviewed, then accepted or rejected; accepted records are later deprecated or superseded, never rewritten.

IF YOU ARE NEW TO THIS

Step by step: changing your mind without rewriting history

The idea that needs the most care here is why an accepted record is never rewritten, even when the team changes its mind. Chapter 1 stated the rule; follow one real change in the food-delivery platform's decision log to see what it means in practice.

- In 2025 the team accepts ADR-0009: the notification service sends push notifications through a third-party SDK. Its context, decision and consequences describe what the team knows and chooses at that time.
- In 2026, after ADR-0012 has brought in a message broker, the team decides to replace the SDK with its own push delivery, running on the broker.
- The tempting shortcut is to open ADR-0009 and rewrite it to describe the new design. But then the log claims the team chose its own push delivery back in 2025, and anyone reading old code or old notification data finds no record that the SDK was ever used, or why.
- Instead the team writes a new record, ADR-0015, and proposes it in a pull request like any other. Its context says what has changed since ADR-0009, which is often the most useful part of the record.
- When ADR-0015 is accepted, the only change to ADR-0009 is its status line, which now reads 'Superseded by ADR-0015'. ADR-0015 links back to it, and the decision log lists both.
- A newcomer reading the log now sees the whole story: what was decided in 2025, what replaced it in 2026, and why.
- **The lesson:** an accepted record describes what was known and decided on its date, which is why it stays true. New thinking goes into a new record; apart from small fixes such as a typo, the old one changes only its status line.

Where to keep ADRs

Keep ADRs in the repository of the code they govern, as plain-text files (usually Markdown), one decision per file, with a sequence number and a short slug in the file name. Two directory conventions are common: `doc/adr`, the default of Nat Pryce's `adr-tools` command-line tool, and `docs/decisions`, recommended by MADR with names such as `0012-publish-order-events-through-broker.md`. Thoughtworks placed lightweight ADRs in the *Adopt* ring of its Technology Radar (most recently in 2018) and specifically recommended source control over a wiki, so that records stay in step with the code they describe.

Keeping records next to the code has concrete advantages. They are reviewed in the same pull requests as the code that implements them, they are versioned and branched with it, a developer can find them with the same search tools, and code can point at them: a comment such as “see ADR-0012” above an outbox relay tells the next reader where the reasons are. Decisions that span many repositories, such as an organisation-wide choice of identity provider, need a home of their own; a dedicated architecture repository with the same conventions works well, with links from each affected codebase.

```
docs/
  decisions/
    README.md                                  ← the decision log (index)
    0001-record-architecture-decisions.md
    0004-use-hosted-card-fields.md
    0007-use-managed-postgresql-for-orders.md
    0012-publish-order-events-through-broker.md
    template.md
src/
  ordering/
    outbox_relay.py      # comment: see docs/decisions/0012-...
```

A typical layout: numbered ADR files beside the code, an index, a template and a code comment that points back to the record

Which decisions deserve an ADR

Writing an ADR for every choice buries the important ones; writing none loses the reasons. The test is significance, as defined in Chapter 1: a decision deserves a record when it is expensive to reverse, affects a key quality attribute, has wide scope, constrains other teams, or was contested. In practice that means choices such as a primary datastore, a communication style between services, an identity provider, an API versioning policy, a hosting platform, a build-versus-buy decision, and conventions whose violation is expensive (how money is represented, how time zones are stored). It also includes decisions *not* to do something that people keep suggesting, such as not introducing a cache yet.

Choices that are local, cheap to reverse and invisible outside one module usually do not need a record: the choice of a test assertion library, the name of a class, the internal layout of a component. A useful signal is the question a newcomer would ask: if someone is likely to ask “why on earth did they do this?” within a year, write the record now, while the reasons are fresh.

DECISION GUIDE

Decision guide: does this decision need an ADR?

Ask these questions in order when a technical choice comes up in design discussion or code review.

Q1 **Would reversing this decision in a year take more than a few days of work, a data migration or changes by other teams?**

- **Yes** Write an ADR. It is a one-way door or close to one.
- **No** Go to the next question.

Q2 **Does it noticeably affect a quality attribute that matters to stakeholders (for example availability, latency, security or cost) or a constraint such as regulation?**

- **Yes** Write an ADR, and link it to the quality scenario it serves.
- **No** Go to the next question.

Q3 **Was it contested, or did the team reject an option that people are likely to propose again?**

- **Yes** Write a short ADR (a Y-statement may be enough) so the reasoning is not re-argued from scratch.
- **No** Go to the next question.

Q4 **Will a newcomer probably look at the result and ask why it was done this way?**

- **Yes** Write a short ADR or at least a code comment and a commit message that explain the reason.
- **No** No ADR needed. A clear commit message and the code itself are enough.

A complete worked ADR

The food-delivery platform used throughout this chapter has customer, restaurant and courier apps, an ordering service, payments through an external payment provider, courier dispatch and notifications. The record below follows the MADR structure, slightly shortened. Notice how the context gives numbers and dates rather than adjectives, how the drivers are stated before the options, how the rejected options are given a fair hearing, and how the consequences include real costs. The decision itself (asynchronous messaging for order events) is explored properly in Chapter 6: event-driven architecture; here the point is how it is recorded.

```
# ADR-0012: Publish order events through a managed message broker

- Status: Accepted
- Date: 2026-09-14
- Deciders: ordering team lead, platform lead, principal architect
- Consulted: restaurant, courier and notifications teams; SRE
- Related: ADR-0007 (managed PostgreSQL for the order service)

## Context and problem statement

When a customer's payment succeeds, the ordering service calls the restaurant,
courier-dispatch and notification services synchronously over HTTP before it
confirms the order. On Friday evenings (peak about 40 orders/s) p99 order
confirmation time reaches 4.2 s against our target of 1.5 s (scenario QS-3).
On Friday 21 August a notification-service outage blocked all new orders for
23 minutes. Two new consumers (loyalty points, analytics) want order data.

## Decision drivers

- Order confirmation must not depend on notification or dispatch availability.
- p99 confirmation under 1.5 s at three times today's peak.
- No order event may be lost, even if a service crashes mid-request.
- New consumers should not require changes to the ordering service.
- The platform team has three people; operational load must stay small.

## Considered options

1. Keep synchronous calls; add timeouts, retries and circuit breakers.
2. Publish events to a managed message broker using a transactional outbox.
3. Let consumers poll a shared "order_events" table in the order database.

## Decision outcome

Chosen option: 2. We will publish OrderPlaced, OrderAccepted, OrderPickedUp,
OrderDelivered and OrderCancelled events to our cloud provider's managed
broker. The ordering service writes each event to an outbox table in the
same database transaction as the order; a relay publishes it. Payment
authorisation stays synchronous, because the customer must see the result.

### Consequences

- Good: confirmation no longer waits for downstream services.
- Good: loyalty and analytics can subscribe without ordering-team work.
- Bad: services become eventually consistent; the restaurant tablet may
  show a new order a second or two after the customer's confirmation.
- Bad: a broker to pay for and monitor; consumers must be idempotent
  because delivery is at-least-once.
- Neutral: event schemas become public contracts and need versioning.

### Confirmation
```

```
Load test at 120 orders/s in staging before go-live; a build check fails
if the ordering service gains an HTTP client for notification or dispatch.

## Pros and cons of the other options

- Option 1: least new infrastructure, but a slow dependency still delays
  every order, and each new consumer means another call to add.
- Option 3: no new infrastructure, but couples every consumer to our schema
  and adds polling load to the order database (see ADR-0007).
```

Markdown source of ADR-0012 in docs/decisions/0012-publish-order-events-through-broker.md

WORKED EXAMPLE

Worked example: the same decision as a Y-statement

For the decision log, a pull request description or a slide, the team also writes the decision as one Y-statement.

Y-statement	In the context of **order confirmation in the food-delivery platform**, facing **slow or unavailable downstream services that block orders at peak times**, we decided for **publishing order events through a managed message broker with a transactional outbox** and neglected **synchronous calls with circuit breakers, and consumers polling a shared table**, to achieve **order confirmation that does not depend on notification, dispatch or restaurant services, and new consumers that need no change to ordering**, accepting that **services become eventually consistent, the broker must be run and paid for, and every consumer must handle duplicate events**.
What it keeps	The context, the driving concern, the choice, the rejected options, the benefits and the accepted costs: every element of the full record, at a glance.
What it loses	The numbers (p99 of 4.2 s, 23-minute outage), the decision-makers, the confirmation plan and the reasoning against each rejected option. For a decision this significant, the full record is still needed.

A good test of any ADR is whether you can compress it into a sensible Y-statement. If you cannot fill the *neglected* or *accepting that* clauses, the record probably considered no real alternatives or hid its costs.

Reviewing ADRs

Because an ADR is a file in the repository, the natural review mechanism is the pull request. The proposer opens it with status *proposed*; the people affected comment; the record is revised; when the accountable people agree, the status changes to *accepted* and it is merged.

This keeps the discussion attached to the record and visible to anyone who later reads the history.

Who must review depends on how decisions are governed. In a centralised model an architecture board approves significant records. In the advice process described by Andrew Harmel-Law (martinfowler.com, 2021), anyone may make an architectural decision, provided they first seek advice from everyone meaningfully affected and from people with relevant expertise, and record that advice; the decision-maker need not follow the advice but must hear it. ADRs are the backbone of that process: the advice appears in the record, and a regular open advisory forum reviews proposed records. Governance models are covered in Chapter 11: governance in agile organisations.

Olaf Zimmermann's *ecADR* checklist (2020) gives reviewers a compact definition of done for a decision. **E**vidence: there is reason to believe the option will work, from a spike, a proof of concept or trusted experience. **C**riteria: at least two options were compared against the stakeholder concerns. **A**greement: the decision has been challenged by peers and the team accepts it. **D**ocumentation: it is recorded and shared with those affected. **R**ealisation and review: implementation is planned and a date or trigger for revisiting it is set. A reviewer who checks those five points, plus the honesty of the consequences, catches most weak records.

Anti-patterns

ADRs fail in recognisable ways. Zimmermann catalogued several in 2023 with memorable names, and the most damaging are worth knowing.

WATCH OUT

How ADRs go wrong

- **Written after the fact to justify.** The decision was made in a corridor and the record is reverse-engineered to look rational. Its options are invented and its context is selective. Write the record while deciding, as a proposal, not as a press release afterwards.
- **No options considered** (Zimmermann's *Sprint*), or a *Dummy Alternative*: a straw-man option that could never work, added so the favourite looks vetted. If no credible alternative exists, say so and why.
- **No real consequences** (the *Free Lunch Coupon*) or only benefits (the *Fairy Tale*). Every significant decision has costs; a record that hides them will be found out, and readers stop trusting the log.
- **The Mega-ADR.** Diagrams, component specifications and code stuffed into a multi-page record. Link to the architecture documentation instead; the ADR records the decision and its reasons.
- **Editing accepted records** to match the current design, which destroys the history the log exists to keep.
- **ADRs for everything,** including trivial two-way doors, until nobody reads them; or ADRs that nobody can find, scattered across wikis, chat threads and slide decks.

The decision log as an index

Individual records are useful; the log as a whole is more useful. A short index file (often the `README.md` of the decisions folder) lists every record with its number, title, status and date. A newcomer reads it in five minutes and gets a compressed history of the architecture: what was decided, what was rejected, what was replaced and when. Tools such as `adr-tools` can generate the index and create superseding records that update the old record's status automatically, and documentation tools including the Structurizr DSL can import a folder of ADRs into the published documentation.

Treat the log as part of the architecture description. The arc42 template has a section for architecture decisions that can simply link to it, and a docs-as-code pipeline can publish it with the rest of the documentation.

```
# Decision log

| ADR  | Title                                                | Status                | Date       |
|------|------------------------------------------------------|-----------------------|------------|
| 0001 | Record architecture decisions                        | Accepted              | 2025-03-02 |
| 0004 | Use the payment provider's hosted card fields        | Accepted              | 2025-04-18 |
| 0007 | Use managed PostgreSQL for the order service         | Accepted              | 2025-06-11 |
| 0009 | Send push notifications through a third-party SDK    | Superseded by 0015    | 2025-09-30 |
| 0012 | Publish order events through a managed broker        | Accepted              | 2026-09-14 |
| 0013 | Store courier locations in a time-series database    | Rejected              | 2026-09-20 |
| 0015 | Replace the SDK with our own push delivery on broker | Accepted              | 2026-10-08 |
```

docs/decisions/README.md: the decision log for the food-delivery platform

CHECK YOUR UNDERSTANDING

▼ **Why should the Context section of an ADR avoid arguing for the chosen option?**

Because its job is to record the forces and facts as they were, so that a later reader can judge whether they still hold. If the context is written as advocacy, the reader cannot separate the situation from the conclusion.

▼ **Your team changes its mind about an accepted decision. What do you do with the old ADR?**

Leave its content unchanged, set its status to superseded with a link to the new record, and write a new ADR that explains what changed in the context and what is now decided.

▼ **What does MADR's Confirmation section add that Nygard's format lacks?**

A statement of how the team will check that the decision is implemented and still holds, such as a load test, a review or an automated dependency check.

▼ **What is a quick test of whether an ADR genuinely considered alternatives and costs?**

Try to write it as a Y-statement. If the "and neglected" or "accepting that" clauses cannot be filled honestly, the record is missing real options or real consequences.

▼ **Try it: Critique this ADR: "We chose Kafka because it is the best broker. Consequences: fast and scalable." Six months later someone edits it to say RabbitMQ. What is wrong, and what should have been done each time?**

It gives no context, no alternatives and no criteria, and "best" is not a reason. The consequences list only benefits, with no costs such as running the cluster. It should state the problem with numbers, compare at least two options fairly, then say "We will..." and name what is given up. Editing later destroys the history: write a new ADR that supersedes it, and mark the old one superseded.

SOURCES

- Documenting Architecture Decisions, Michael Nygard (2011): https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
- MADR: Markdown Architectural Decision Records: https://adr.github.io/madr/
- Architecture Decision Record, Martin Fowler (bliki): https://martinfowler.com/bliki/ArchitectureDecisionRecord.html
- Scaling the Practice of Architecture, Conversationally, Andrew Harmel-Law: https://martinfowler.com/articles/scaling-architecture-conversationally.html
- A Definition of Done for Architectural Decision Making, Olaf Zimmermann: https://ozimmer.ch/practices/2020/05/22/ADDefinitionOfDone.html
- Jeff Tyree and Art Akerman: Architecture Decisions: Demystifying Architecture (IEEE Software, 2005): https://doi.org/10.1109/MS.2005.27

SECTION 4.6 OF 4.8 · INTERMEDIATE · 22 MIN READ · LAST REVIEWED NOVEMBER 2026

arc42 and other documentation templates

BEFORE YOU READ THIS

This section builds on:

- 4.2 Views and viewpoints
- 4.3 The C4 model
- 4.5 Decision records

Terms it uses from earlier: Architecture description, Viewpoint, Correspondence, Quality attribute scenario, Architecture Decision Record (ADR)

THE IDEA IN BRIEF

A template is like the standard layout of a car's handbook: every manual has a section on tyres, one on warning lights, one on servicing, so you know where to look in any car. arc42 does the same for software architecture. It gives twelve standard sections, such as goals, context, structure and decisions, so that writers know what to cover and readers know where to find it. You do not have to fill every section; a small system might need only four or five short ones.

WHAT YOU NEED TO KNOW

- arc42, created by Gernot Starke and Peter Hruschka and free under a Creative Commons licence, is a template of twelve sections; it prescribes where information goes, not how to design or which notation to use.
- The sections are introduction and goals, constraints, context and scope, solution strategy, building block view, runtime view, deployment view, cross-cutting concepts, architecture decisions, quality requirements, risks and technical debt, and glossary.
- Every section is optional; a minimal description covers three to five quality scenarios, the context, a brief solution strategy, the level-1 building block view and the key cross-cutting concepts.
- Use arc42 lightly: link to requirements, ADRs and infrastructure code rather than copying them, and state deliberately when a section is not needed.
- The SEI Views and Beyond template is more thorough, with a roadmap, per-view packages and a mapping between views; IEEE 1016-2009 specifies software design descriptions and has been inactive-reserved since 2020.

> - ISO/IEC/IEEE 42010 prescribes the content of an architecture description, not a template; an arc42 description conforms once stakeholders, concerns, viewpoints, views, correspondences and rationale are made explicit.

What arc42 is

arc42 is a free template for describing and communicating software architecture, created by Gernot Starke and Peter Hruschka and used in practice since 2005. It is published under a Creative Commons Attribution-ShareAlike licence, in more than a dozen languages and in formats including AsciiDoc, Markdown and Word, with a “plain” version and a version with built-in help text. It is especially widespread in German-speaking countries and in iSAQB training, but it is used worldwide.

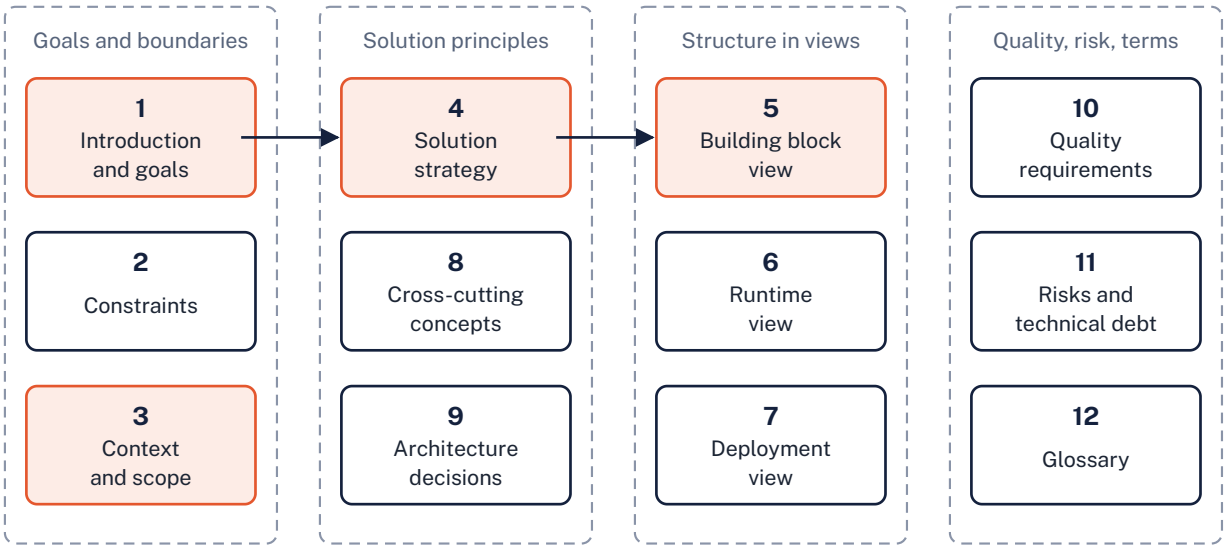
arc42 is deliberately not a method and not a notation. It does not tell you how to design, and it does not prescribe UML, C4 or any other diagram style. It is a set of twelve labelled drawers: for each kind of architectural information, it says where to put it, so that every reader knows where to look and every writer knows what is missing. Its authors are explicit that every section is optional; a small system might fill four sections in a few pages, a large one might fill all twelve across many.

The template builds on ideas from earlier in this chapter. Its three views (building block, runtime and deployment) correspond closely to the SEI's module, component-and-connector and allocation categories and to Kruchten's development, process and physical views from views and viewpoints. Its decision section is where Architecture Decision Records go, and its quality section is where the quality attribute scenarios of Chapter 2 go.

The twelve sections

The diagram groups the sections by the question they answer. Sections 1 to 3 set the goals and boundaries: what the system is for, what limits the solution and where the system ends. Section 4 states the solution's fundamental ideas, and sections 8 and 9 hold the principles and decisions that apply across it. Sections 5 to 7 show the structure from three angles. Sections 10 to 12 make quality, risk and vocabulary explicit.

The structure of arc42



Shaded: the core of even a minimal description (goals, context, strategy and level-1 building blocks); the arc42 FAQ also asks for 3 to 5 quality scenarios and key concepts (8).

The twelve arc42 sections grouped by purpose; the highlighted sections are the core of even a minimal description.

What goes in each section, and how much

Section	What goes in it	Small system (one team)	Large system (many teams)
1. Introduction and goals	A short requirements overview, the top three to five quality goals in priority order, and a stakeholder table with each group's expectations.	Half a page: purpose, three quality goals, five or six stakeholders.	Two or three pages, linking to the requirements documents rather than copying them.
2. Constraints	Anything that limits design freedom: regulations, mandated technologies, organisational rules, deadlines, budget, conventions.	A bullet list.	A table grouped into technical, organisational and legal constraints, each with its source.
3. Context and scope	The system as a black box: users and neighbouring systems, what flows between them (business context) and the channels and protocols used (technical context).	One context diagram (often a C4 system context diagram) and a short table of external interfaces.	Business and technical context diagrams plus an interface table with owners and contracts.
4. Solution strategy	The few fundamental decisions and approaches that shape everything else: decomposition style, key technologies, how the top quality goals are achieved.	A handful of bullets, each tied to a quality goal.	One or two pages; a table mapping each quality goal to the approach that serves it.
5. Building block view	The static decomposition into building blocks (subsystems, services, components, modules), refined level by level, each with responsibilities and interfaces. Usually the largest section.	Level 1 only: one diagram (a C4 container diagram works) and a sentence per block.	Level 1 for the whole system, levels 2 and 3 only for complex or risky parts.
6. Runtime view	How building blocks cooperate in important scenarios: key use cases, start-up, error handling, failover.	One or two sequence diagrams for the most important flows.	The critical and the surprising scenarios; never every use case.
7. Deployment view	The technical infrastructure: environments, nodes, networks, and which building blocks run where.	One diagram or a short list of hosted services.	Diagrams per environment and region, linked to the infrastructure-as-code that defines them.

Section	What goes in it	Small system (one team)	Large system (many teams)
8. Cross-cutting concepts	Concepts that apply across many building blocks: domain model, security, error handling, logging and observability, persistence, internationalisation, testing.	A short paragraph each for the two or three concepts that matter.	A page or more per concept, often owned by a platform or enabling team.
9. Architecture decisions	Important decisions with their reasons, unless recorded elsewhere.	A link to the ADR folder.	A link to the decision log, perhaps with the five or six decisions every newcomer must know.
10. Quality requirements	A quality tree (the utility tree from Chapter 2) and concrete quality scenarios that refine the quality goals of section 1.	Three to five scenarios.	A utility tree with prioritised scenarios, linked to fitness functions.
11. Risks and technical debt	Known technical risks and debt, ideally prioritised, with mitigation plans.	A short list, reviewed each quarter.	A link to the debt register, plus the top risks with owners.
12. Glossary	The important domain and technical terms, with definitions everyone agrees on.	Ten or so terms.	A shared glossary, sometimes per bounded context.

Using arc42 lightly

The most common way to fail with arc42 is to treat it as a form that must be completed: twelve headings are created, each is filled with something, and the result is long, generic and never read. The template's own guidance points the other way. The arc42 FAQ says there is no general rule for the minimum, because it depends on criticality, size, complexity and risk, but suggests that even a minimal description should contain three to five central quality requirements as scenarios, the context and external interfaces, a brief solution strategy, the level-1 building block view and the most important cross-cutting concepts.

Four habits keep arc42 light. **Delete empty sections or mark them deliberately** (“Deployment: single managed platform; see the infrastructure repository”), so that a reader knows the gap is intentional. **Link rather than copy**: the requirements live in the backlog, the decisions in the ADR folder, the deployment in infrastructure-as-code; arc42 points to them. **Prefer diagrams with short element tables** to pages of prose. **Write for the next reader's questions**, not for completeness: if nobody will ask about the runtime behaviour of the admin screens, do not describe it.

arc42 combines well with the C4 model. A common pairing is a C4 system context diagram in section 3, a container diagram as level 1 of the building block view in section 5 with component diagrams as level 2, and a C4 deployment diagram in section 7. Kept as diagrams as code in the same repository as the arc42 text, the whole description can be versioned and reviewed like code.

IF YOU ARE NEW TO THIS

Step by step: a pointer, not a copy

The step most people trip over here is that arc42 is a set of labelled drawers, not a form to fill in. Many sections should hold a short pointer to where the real information already lives, not a copy of it.

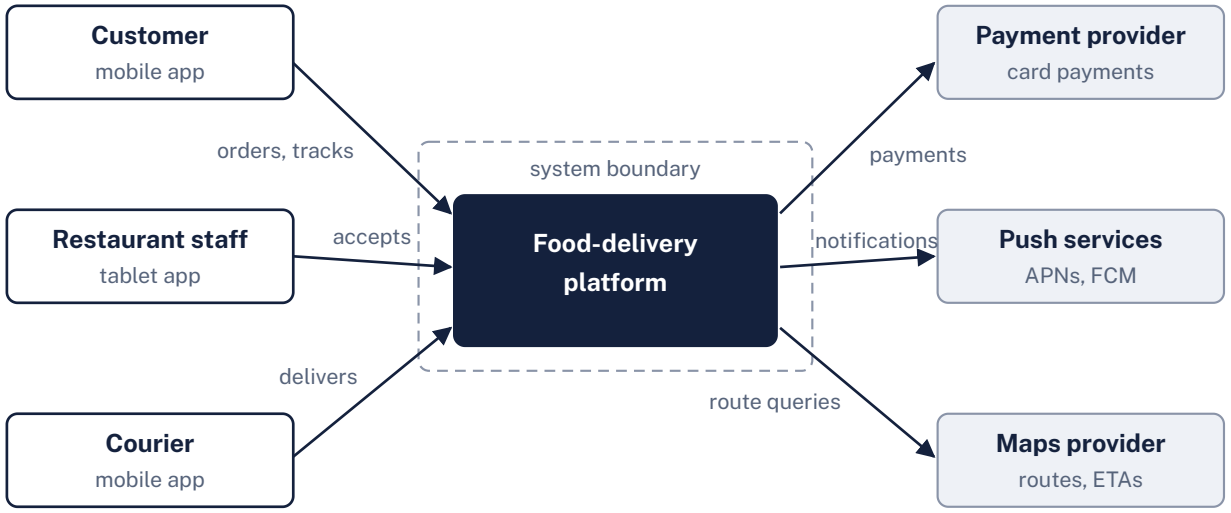
- The food-delivery team already keeps two C4 diagrams (system context and containers), a decision log in docs/decisions and its quality scenarios in docs/quality.md. It now organises them with arc42.
- The diagrams go into their drawers: the context diagram in section 3, and the container diagram as level 1 of the building block view in section 5.
- For section 9, architecture decisions, someone suggests pasting in the full text of every decision record, so that everything sits in one document.
- Then ADR-0015 supersedes ADR-0009, as in the previous section. The pull request that adds ADR-0015 also changes the decision log, but nobody remembers the pasted copy, which still presents ADR-0009 as accepted.
- A reader now finds two answers: the arc42 description says push notifications go through a third-party SDK, while the decision log says the team now runs its own push delivery. They cannot tell which is true, and begin to distrust the whole description.
- So section 9 holds a link to docs/decisions and the numbers of a few must-read records. Every other section works the same way: the quality scenarios stay in docs/quality.md, the deployment in the infrastructure repository, and a section with nothing to add is deleted or says why in one line.
- **The lesson:** arc42 tells readers where to look; it does not have to hold everything itself. Give each fact one home and let the template point to it, and the description stays short and true.

Worked example: arc42-lite for the food-delivery platform

The page below is a complete arc42-lite description of the food-delivery platform used throughout this chapter. It fits on one printed page plus the context diagram, yet it answers the questions a new engineer, an operations lead or an auditor would ask first, and it points to

where the detail lives.

Section 3: business context



The context of the platform: three groups of users on the left, three external services on the right, each arrow labelled with what is exchanged; channels and protocols belong in the technical context.

WORKED EXAMPLE

Worked example: a one-page arc42-lite description

Food-delivery platform, architecture description, version reviewed 2026-10-01 by the principal architect. Sections that would add nothing are stated as such rather than left empty.

1. Goals	Let customers order from local restaurants and track delivery in real time. Quality goals in priority order: (1) no order is lost or charged twice; (2) order confirmation p99 under 1.5 s at 120 orders/s; (3) a new consumer of order data can be added without changing the ordering service. Stakeholders: customers, restaurants, couriers, operations, finance, the data protection officer, the payment provider.
2. Constraints	Card data never touches our systems (hosted card fields from the payment provider, to limit PCI DSS scope). UK and EU personal data stays in EU regions. Five squads; server-side code in Java, Go or TypeScript only.
3. Context	See the context diagram. External interfaces: payment provider REST API and webhooks; APNs and FCM for push; maps provider for routes and ETAs. Interface owners and contracts are listed in `docs/interfaces.md`.
4. Solution strategy	Separate services for ordering, dispatch and notifications (goal 3); asynchronous order events through a managed broker with a transactional outbox (goals 1 and 2, ADR-0012); managed PostgreSQL for orders (goal 1, ADR-0007); idempotency keys on every payment call (goal 1).
5. Building blocks	Level 1 is the C4 container diagram in `docs/diagrams/containers.dsl`: three client apps, API gateway, ordering, dispatch and notification services, order database and the order events topic on the managed broker. Level 2 exists only for ordering, the riskiest service.
6. Runtime	Two scenarios: *place and pay for an order* (sequence diagram) and *payment provider timeout* (shows the retry with idempotency key and the pending state).
7. Deployment	One cloud provider, two EU regions, managed Kubernetes and managed databases; defined in the infrastructure repository, which is the source of truth.
8. Concepts	Domain model (order lifecycle states); idempotency and retries; observability (trace IDs carried in every event); personal data handling and retention.
9. Decisions	See the decision log in `docs/decisions/`. Must-read records: ADR-0004, ADR-0007, ADR-0012.

10. Quality	Scenarios QS-1 to QS-5 in `docs/quality.md`, each with a response measure; QS-3 (confirmation latency) and QS-1 (no lost orders) have automated fitness functions.
11. Risks and debt	Single broker vendor (lock-in, accepted in ADR-0012); dispatch still reads the order database directly (debt, removal planned for Q1); maps provider rate limits at peak (risk, owner: dispatch squad).
12. Glossary	Order, basket, acceptance, pickup, drop-off, dispatch, ETA, outbox, idempotency key: defined in `docs/glossary.md`.

Every section is either filled briefly or points to a single source of truth; nothing is copied, so nothing can drift out of step.

The quality goals in section 1 drive the strategy in section 4, which cites the decisions in section 9: a reader can trace each major choice back to a goal.

Other templates

The SEI *Views and Beyond* template. The Software Engineering Institute publishes a documentation template that follows *Documenting Software Architectures: Views and Beyond* by Paul Clements and colleagues. Its top level has six parts: a documentation roadmap and overview, architecture background, the views, the mapping between views, referenced materials, and a glossary with acronyms. Each view is documented in the same way, with a primary presentation, an element catalogue, a context diagram, a variability guide and the rationale (described in the "Going deeper" box of Section 4.2). It is more thorough than arc42 and suits large, long-lived or contractually specified systems where every view must be complete and cross-referenced; its mapping between views is a strength that arc42 leaves implicit.

IEEE 1016. IEEE Std 1016-2009, *Software Design Descriptions*, specifies the content and organisation of a software design description (SDD), the detailed design document traditionally required in defence, aerospace and other contract-driven projects. It borrowed the concepts of stakeholder, concern, viewpoint and view from IEEE 1471 (the predecessor of 42010) and lists design viewpoints such as context, composition, logical, dependency, information, interface, interaction, state dynamics, algorithm and resource. IEEE moved it to *inactive-reserved* status in 2020, but it is still cited in contracts and in teaching, and older programmes still deliver SDDs in its form. It sits closer to detailed design than to architecture.

Design documents and RFCs, used by many engineering organisations, are a different kind of template: a proposal for one change rather than a description of a whole system. They are covered in the next section but one, communicating architecture.

Conformance with ISO/IEC/IEEE 42010

ISO/IEC/IEEE 42010:2022 sets requirements for architecture descriptions and for architecture description frameworks (ADFs): established conventions, such as arc42 or the Views and Beyond approach, for producing descriptions in a given community. It does not prescribe a template, a notation or a process. A description *conforms* when it contains the information the standard requires, whatever template organises it.

In broad terms that means identifying the system described and its environment, identifying the stakeholders and their concerns, stating which viewpoints are used and which concerns each frames, providing views that follow those viewpoints, recording correspondences (how elements in different views relate) and recording the rationale for key decisions. arc42 does not claim conformance out of the box, and plain arc42 descriptions often lack an explicit link from each concern to the view that addresses it. The gaps are small and easy to close, as the table shows. Most teams do not need formal conformance; those in regulated industries or bound by contract sometimes do, and it is reassuring that a lightweight template can meet the standard with a few additions.

Making an arc42 description conform to 42010

42010 asks for	Where arc42 has it	What to add
Stakeholders and their concerns	Section 1.3 stakeholder table; quality goals in 1.2 and 10	List concerns explicitly per stakeholder, not just expectations.
Viewpoints, each framing named concerns	Implicit in sections 3, 5, 6 and 7	A short table naming each viewpoint, the concerns it frames and its notation (for example C4 container diagrams).
Views that follow the viewpoints	Sections 3, 5, 6 and 7	Nothing, if each view states which viewpoint it follows.
Correspondences between views	Partly in section 7 (what runs where)	Mapping tables, for example building block to deployment node, building block to owning team.
Rationale for key decisions	Sections 4 and 9 (ADRs)	Link each decision to the concerns it addresses.

WATCH OUT

Common mistakes with templates

- Filling every section because it exists. An empty section with one line explaining why is better than a page of filler.
- Copying requirements, decisions and infrastructure details into the template, where they drift out of step with the originals. Link instead.
- A building block view with dozens of boxes at level 1. Level 1 should fit on one screen; detail belongs in lower levels, and only where it pays.
- Writing the description once at the start and never again. Put it under version control, review it with significant changes and show a “last reviewed” date.
- Mixing templates without a map. If you use arc42 for structure and C4 for diagrams, say so in section 1, so readers know the conventions.

CHECK YOUR UNDERSTANDING

▼ Which arc42 section usually becomes the largest, and how do you stop it growing out of control?

Section 5, the building block view. Keep level 1 to one diagram that fits on a screen and refine to levels 2 and 3 only for complex or risky building blocks.

▼ Where in arc42 do quality goals appear, and how do sections 1 and 10 differ?

Section 1 lists the top three to five quality goals in priority order for orientation; section 10 refines them, and any further qualities, into a quality tree and concrete testable scenarios.

▼ Your team already keeps ADRs in the repository. What should arc42 section 9 contain?

A link to the decision log, perhaps with a short list of the decisions every newcomer must read, rather than a copy of the records.

▼ Does using arc42 make an architecture description conform to ISO/IEC/IEEE 42010?

Not automatically. Conformance depends on content: stakeholders, concerns, viewpoints, views, correspondences and rationale must be explicit, which usually needs a few small additions to arc42.

▼ Try it: A three-person team runs a small restaurant-menu service. Their manager tells them to fill in all twelve arc42 sections. What would you advise instead, and how should sections they do not need be treated?

Use arc42-lite. Write three to five quality scenarios, the context, a brief solution strategy, the level-1 building blocks and the key cross-cutting concepts, ideally on about one page. Link to ADRs and infrastructure code instead of copying them. Every section is optional, so state deliberately that a section is not needed, rather than leaving it blank or padding it. arc42 says where information goes; it was never meant to demand volume.

SOURCES

- arc42 overview: https://arc42.org/overview
- arc42 documentation: section 1, Introduction and Goals: https://docs.arc42.org/section-1/
- arc42 FAQ: What is the minimal amount of an arc42 documentation?: https://faq.arc42.org/questions/B-4/
- SEI Software Architecture Documentation Template (Views and Beyond): https://wiki.sei.cmu.edu/confluence/display/SAD/Software+Architecture+Documentation+Template
- IEEE 1016-2009: Software Design Descriptions: https://standards.ieee.org/ieee/1016/4502/

SECTION 4.7 OF 4.8 · INTERMEDIATE · 22 MIN READ · LAST REVIEWED NOVEMBER 2026

Diagrams as code and keeping documentation current

BEFORE YOU READ THIS

This section builds on:

- 0.8 How software gets built
- 4.1 Why document
- 4.3 The C4 model

Terms it uses from earlier: Version control, Pull request, Continuous integration (CI), Sequence diagram, Living documentation

THE IDEA IN BRIEF

Imagine a family recipe book where every change is pencilled in by whoever cooks, with no record of who changed what. After a few years nobody trusts it. Now imagine the recipes are typed, every change is suggested, checked by a second cook and dated, and the printed book is reprinted automatically after each approved change. Diagrams as code applies that discipline to architecture diagrams: they are written as short text files, kept with the software, checked when the software changes and redrawn automatically.

WHAT YOU NEED TO KNOW

- Diagrams as code means writing diagrams in a text language and letting a tool draw them, so they can be versioned, diffed, reviewed in pull requests and rendered by the build pipeline.
- PlantUML (2009) is strongest for UML, especially sequence diagrams; Mermaid (2014) renders directly in Markdown on GitHub and GitLab; D2 (2022) offers clean layouts; Structurizr DSL defines one C4 model from which many views are generated.
- Drawing tools know only shapes, so the same element can drift between diagrams; modelling tools keep elements and relationships in a model, so every view stays consistent and the model can be queried and checked.
- A docs-as-code pipeline renders diagrams, checks links, builds the documentation site on every pull request and publishes from main, so readers see documentation that matches the code.
- Documentation stays current through habits, not tools: named owners, review alongside code changes, last-reviewed dates, links between docs, code and decisions, and deleting stale pages (ADRs are superseded, not deleted).

What diagrams as code means

Diagrams as code means writing a diagram as a short text file in a small language and letting a tool lay it out and draw it. Instead of dragging boxes on a canvas, you write `ordering → db: SQL`, and the tool produces the picture. The text file lives in the code repository and is treated like code: versioned, reviewed, checked by automated builds and published by the pipeline.

It is one part of docs-as-code, the practice, championed by the Write the Docs community, of producing documentation with the same tools and workflow as software: issue trackers, version control (usually Git), plain-text markup such as Markdown or AsciiDoc, code review and automated tests. The previous sections produced the material: views and C4 diagrams, UML where precision matters, ADRs and an arc42 structure. This section is about keeping that material accurate for the life of the system, which is where most architecture documentation fails. The reasons it fails were set out in why and how much to document: stale wikis, big documents nobody owns and diagrams that no longer match the system.

Why put diagrams in text

Version control. A diagram in text has a history. You can see who changed it, when and why, and compare any two versions line by line; a binary image or a file in a drawing tool's format shows only that something changed. You can also branch it: the diagram for a proposed change lives on the same branch as the code that implements it.

Review in pull requests. When a developer adds a new dependency between services, the change to the diagram appears in the same pull request as the code, as a readable text diff. Reviewers can ask whether the new arrow is wanted, which is an architecture review at the moment it is cheapest.

Consistency. The tool applies the same shapes, fonts and layout rules to every diagram, so diagrams drawn by twenty people still look like one set. With a modelling language (below), the same element appears with the same name and description in every view.

Generation. Text can be produced by programs. Diagrams can be generated from the code itself (dependency graphs), from infrastructure-as-code (deployment diagrams), from API specifications or from a central model, and then checked automatically against the system.

The trade-offs are real. Automatic layout is convenient but sometimes ugly, and fighting a layout engine to place one box can take longer than drawing it; most tools offer hints such as direction or grouping, not exact positions. Text languages have a learning curve for people who do not code, such as product managers. And a beautiful sketch on a whiteboard during a design discussion is still the fastest way to think; diagrams as code are for the diagrams you intend to keep.

Four languages, one example

Four languages dominate. Each example below describes part of the same food-delivery platform.

PlantUML, created by Arnaud Roques and first released in 2009, is the oldest of the four. It covers all the common UML diagrams (sequence, component, deployment, state, activity, class) and many others, and uses Graphviz to lay out most diagram types. The C4-PlantUML library adds C4 shapes. Its sequence diagrams are particularly good, which makes it a natural choice for the runtime scenarios of UML essentials.

```
@startuml
title Place an order (happy path)
actor Customer
participant "Customer app" as App
participant "Ordering service" as Ord
participant "Payment provider" as Pay
queue "Order events topic" as MB

Customer → App : Tap "Pay"
App → Ord : POST /orders (HTTPS)
Ord → Pay : Authorise payment
Pay ⟶ Ord : Authorised
Ord ⟶» MB : OrderPlaced (async)
Ord ⟶ App : 201 Created
@enduml
```

PlantUML: a sequence diagram of placing an order; `⟶»` draws an asynchronous message and `⟶` a reply

Mermaid, created by Knut Sveidqvist in 2014, is written in JavaScript and renders in the browser. Its biggest advantage is reach: GitHub has rendered Mermaid blocks in Markdown files since February 2022, and GitLab and many wiki and note-taking tools do the same, so a diagram in a README needs no build step at all. It supports flowcharts, sequence, class, state and entity-relationship diagrams, among others, and has experimental C4 support. Its layout control is limited for large diagrams.

```
flowchart LR
    app["Customer app<br/>Kotlin, Swift"]
    gw["API gateway"]
    ord["Ordering service<br/>Java"]
    db[("Order database<br/>PostgreSQL")]
    topic[["Order events<br/>Kafka topic"]]
    disp["Dispatch service<br/>Go"]
    notif["Notification service<br/>Node.js"]

    app ⟶|HTTPS/JSON| gw
    gw ⟶|HTTPS/JSON| ord
    ord ⟶|SQL| db
    ord ⟶|publishes OrderPlaced| topic
    disp ⟶|consumes| topic
    notif ⟶|consumes| topic
```

Mermaid: a container-level flowchart; `[( )]` draws a database cylinder and labels sit on the arrows; the topic is drawn as a container, as Section 4.3 recommends

Structurizr DSL, created by Simon Brown alongside the C4 model, is different in kind. It does not describe a diagram; it describes a *model* (people, software systems, containers, components and the relationships between them) and then defines *views* of that model. Each element is defined once, with its name, description and technology, and appears in every view that includes it. Relationships between containers are automatically implied at the system level, so the context view needs no separate arrows. The DSL can also attach Markdown documentation and a folder of ADRs to the workspace, and the tooling can export views to PlantUML, Mermaid, D2 and other formats.

```
workspace "Food delivery" {
    model {
        customer = person "Customer" "Orders food from local restaurants."
        payments = softwareSystem "Payment provider" "Takes card payments." "External"
        platform = softwareSystem "Food-delivery platform" {
            app = container "Customer app" "Ordering and tracking." "Kotlin, Swift"
            gateway = container "API gateway" "Authenticates and routes requests."
"Managed API gateway"
            ordering = container "Ordering service" "Accepts and tracks orders." "Java"
            db = container "Order database" "Orders and their states." "PostgreSQL"
"Database"
            orderEvents = container "Order events" "Order lifecycle events." "Kafka
topic" "Topic"
            dispatch = container "Dispatch service" "Assigns couriers." "Go"
        }
        customer → app "Places and tracks orders using"
        app → gateway "Makes API calls to" "JSON/HTTPS"
        gateway → ordering "Routes requests to" "JSON/HTTPS"
        ordering → db "Reads from and writes to" "SQL/TCP"
        ordering → payments "Takes card payments using" "JSON/HTTPS"
        ordering → orderEvents "Publishes order events to" "Kafka"
        dispatch → orderEvents "Consumes order events from" "Kafka"
    }
    views {
        systemContext platform "Context" {
            include *
            autoLayout lr
        }
        container platform "Containers" {
            include *
            autoLayout lr
        }
        styles {
            element "Database" {
                shape cylinder
            }
            element "Topic" {
                shape pipe
            }
        }
    }
}
```

Structurizr DSL: one model and two views of the food-delivery platform; the context view is generated from the same definitions

D2 (Declarative Diagramming), open-sourced by Terrastruct in 2022, is the newest. It aims at clean, modern output and readable syntax, supports nested containers (written with braces and referred to with dotted names), and offers a choice of layout engines: dagre and ELK are

bundled and free, and TALA, designed for software architecture diagrams, is a separate commercial engine.

```
direction: right

customer: Customer {shape: person}
payments: Payment provider

platform: Food-delivery platform {
  app: Customer app
  ordering: Ordering service
  db: Order database {shape: cylinder}
  events: Order events topic {shape: queue}
  dispatch: Dispatch service

  app → ordering: Places order (HTTPS)
  ordering → db: SQL
  ordering → events: Publishes OrderPlaced
  dispatch → events: Consumes OrderPlaced
}

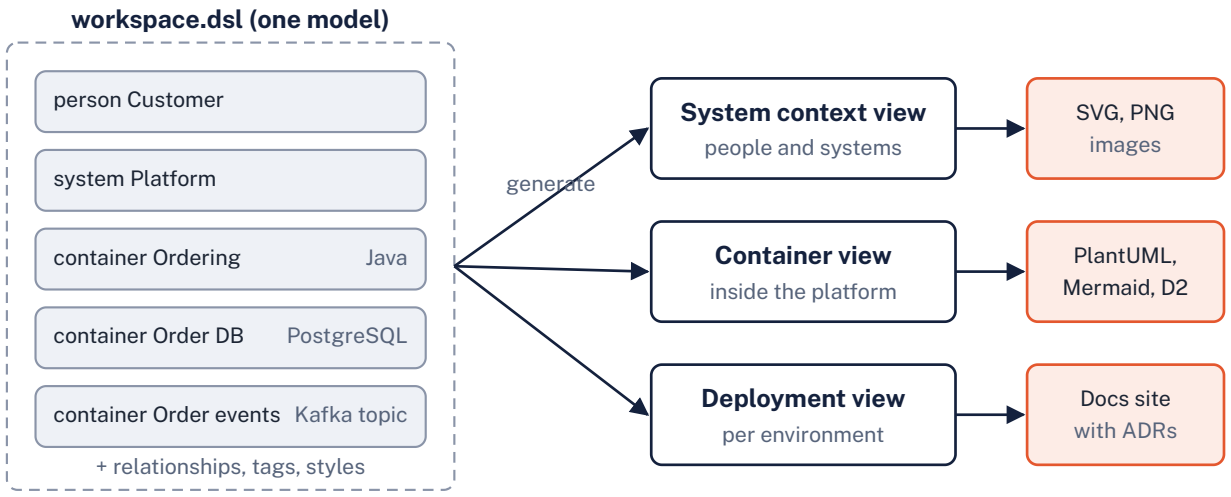
customer → platform.app: Places orders using
platform.ordering → payments: Authorise (HTTPS)
```

D2: nested containers, shapes and labelled connections

Comparing the four

	PlantUML	Mermaid	Structurizr DSL	D2
Describes	One diagram per file	One diagram per block	A model plus views of it	One diagram per file
Strongest at	UML, especially sequence diagrams	Diagrams inside Markdown with no build step	Consistent C4 diagrams at several levels	Attractive general architecture diagrams
Renders	Java tool or server; Graphviz for most layouts	In the browser; native on GitHub and GitLab	Structurizr tools; exports to other formats	Command-line tool; choice of layout engine
Watch out for	Dated default look; layout control by hints only	Limited layout control on large diagrams	Only C4 concepts; needs its own tooling	Best layout engine is commercial

One model, many views



Rename Ordering once in the model and every view and export changes with it.

A modelling language defines each element once; views and exports are generated, so they cannot disagree with each other.

Drawing tools and modelling tools

The difference between Structurizr and the other three is an instance of the distinction between drawing and modelling tools introduced in Section 4.3. A drawing tool (diagrams.net, Excalidraw, Visio, an online whiteboard, and also PlantUML, Mermaid and D2) knows about shapes, lines and text. A box labelled “Ordering service” in one diagram has no connection to the box with the same label in another. Rename the service and you must find and fix every diagram by hand; misspell it once and you have, as far as the tool knows, two services.

A modelling tool (Structurizr, Archi for ArchiMate models, Sparx Enterprise Architect for UML and SysML, Capella) keeps a model: a repository of elements and relationships with properties. Diagrams are views onto the model. Rename an element and every view changes. The model can be queried (“which containers depend on the broker?”), checked (“no component in the customer app may call the database”) and compared with the running system. The price is more up-front structure and, with the heavyweight commercial tools, licences, training and a tendency to model far more than anyone reads.

A practical rule: use drawing tools, text or graphical, for sketches, one-off explanations and diagrams that stand alone, such as a single sequence diagram in an ADR. Use a model when the same elements appear in many diagrams, when several teams contribute, or when you want to check the documentation against the code. Many teams combine them: a Structurizr model for the C4 structure and PlantUML or Mermaid for a few sequence and state diagrams.

IF YOU ARE NEW TO THIS

Step by step: one rename, three files

This section's hardest idea is that writing diagrams as text does not, by itself, keep them consistent. Section 4.3 split tools into drawing tools and modelling tools, and this section adds that PlantUML, Mermaid and D2 are drawing tools, even though you type them, and only a modelling tool such as Structurizr knows that two boxes are the same thing.

- The food-delivery team keeps three diagrams as text: the PlantUML sequence diagram of placing an order, the Mermaid container flowchart and the D2 diagram. Each file spells out the label 'Ordering service' itself.
- Suppose the team renames the service 'Order service'. Someone must find every file that mentions it and edit each label by hand, because nothing links the three boxes: they are just text that happens to match.
- Miss one file and the diagrams disagree. Misspell one label, say 'Order servce', and as far as that tool knows it is a different service. The pipeline still renders every diagram, because each file is valid on its own.
- In the Structurizr DSL file, the ordering service is defined once, in the model, with its name, description and technology. The container view, and any other view that includes it, is generated from that one definition.
- Change the name there and every view picks it up at the next build. Because the model holds elements and relationships, it can also be asked which containers depend on the order events topic, or checked against a rule.
- **The lesson:** text gives diagrams a history, review in pull requests and automatic rendering; only a model makes the same element mean the same thing everywhere. Use one when the same parts appear in many diagrams.

Generating documentation in the pipeline

Once documentation is text in the repository, the continuous integration pipeline from Chapter 0 can build and check it like code. A typical pipeline renders every diagram (failing the build on a syntax error), checks links, builds a static documentation site with a generator such as MkDocs, Antora or Docusaurus, and publishes it when changes reach the main branch. Readers then always see the documentation that matches the code on main, and nobody needs to remember to export a picture and upload it to a wiki.

Further checks can be added as the team matures: a spell checker and prose linter (Vale is popular), a check that every ADR has a valid status, a warning when a page's “last reviewed” date is older than its review interval, and architecture checks that compare the documented dependencies with the actual ones in the code, which are a form of fitness function. Some

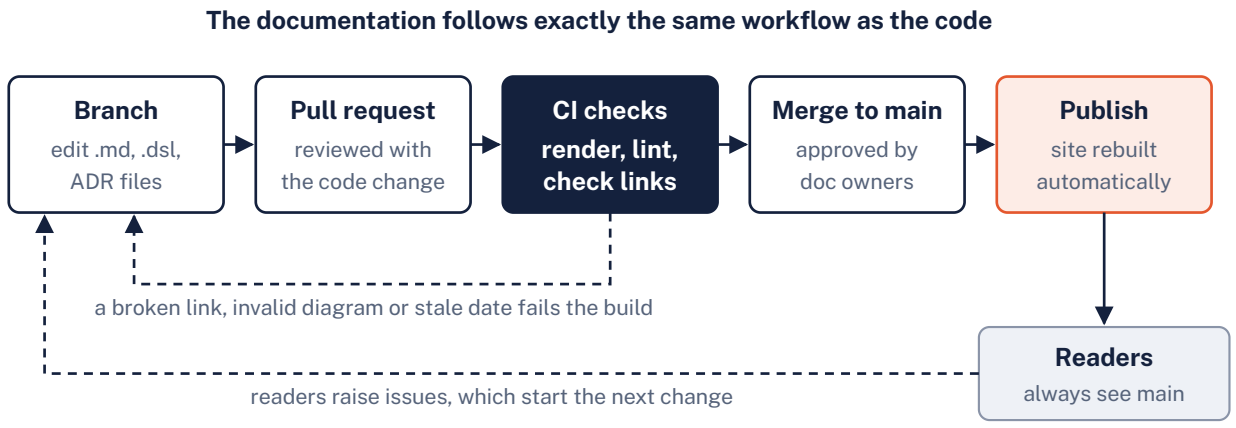
teams generate parts of the documentation from the system itself: dependency graphs from the build, API reference from OpenAPI specifications, deployment diagrams from infrastructure-as-code.

```
name: architecture-docs
on:
  pull_request:
    paths: ["docs/**"]
  push:
    branches: [main]

jobs:
  docs:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      # assumes plantuml, d2, lychee and mkdocs are available on the runner
      - name: Render diagrams (fails on syntax errors)
        run: |
          plantuml -tsvg docs/diagrams/*.puml
          for f in docs/diagrams/*.d2; do d2 "$f" "${f%.d2}.svg"; done
      - name: Check links
        run: lychee --offline docs/
      - name: Build site (warnings such as broken links fail the build)
        run: mkdocs build --strict
      - name: Publish
        if: github.ref == 'refs/heads/main'
        run: ./scripts/publish-docs.sh
```

A GitHub Actions workflow that renders diagrams, checks links, builds the documentation site on every pull request and publishes from main

The docs-as-code workflow



Documentation changes travel the same path as code changes: branch, review, automated checks, merge and automatic publication.

Keeping documentation current

Tools do not keep documentation true; habits do. Five habits make the difference.

Ownership. Every document has a named owner, normally the team that owns the code it describes. In GitHub and GitLab a `CODEOWNERS` file can make that team a required reviewer for changes under its `docs/` folder. Documents owned by “everyone” are maintained by no one.

Review with code changes. A pull request template that asks “Does this change an interface, a dependency, a deployment or a decision? If so, which document or ADR did you update?” catches most drift at the moment it happens. Some teams go further and block merges of features that lack documentation, a practice the Write the Docs guide mentions.

Last-reviewed dates. A small header on each page records its owner, the date it was last checked against reality and how often it should be reviewed. Readers can judge how far to trust it, and the pipeline can flag pages that are overdue. A page that says “last reviewed 2023” is honest; a page that silently describes the system of 2023 is a trap.

Linking documentation and code. Pages point to the code they describe (paths, modules), code comments point to ADRs, and commit messages cite the records they implement. Links in both directions make it obvious which documents a change might affect.

Deleting stale material. Wrong documentation is worse than none, because readers act on it. When a page no longer describes the system and nobody will fix it, delete it or move it to a clearly marked archive. ADRs are the exception: they are records of history, so they are superseded or deprecated, not deleted. This is the living documentation idea that Cyrille Martraire developed in his 2019 book of that name: documentation that evolves at the same pace as the system, ideally because much of it is derived from the system.

```
---
title: "Ordering service: building block view (level 2)"
owner: "@ordering-team"
last_reviewed: 2026-09-14
review_every: 6 months
code: src/ordering/
decisions: [ADR-0007, ADR-0012]
---
```

Front matter at the top of a Markdown page: owner, review date and interval, and links to the code and decisions it describes

Checklist: documentation that stays current

Use this when setting up a repository's architecture documentation, and revisit it every few months.

- [ ] **Documentation lives in the repository it describes**
  Markdown or AsciiDoc pages, diagram sources and ADRs under `docs/`, not in a separate wiki that nobody updates in the same change.
- [ ] **Diagram sources are text and are rendered by the pipeline**
  No hand-exported images that silently fall behind their sources; the build fails on an invalid diagram.
- [ ] **Each element is defined once where diagrams overlap**
  Use a model (such as Structurizr DSL) when the same containers appear in several diagrams.
- [ ] **Every page has an owner and a last-reviewed date**
  Enforced by `CODEOWNERS` and a pipeline check that warns when the review interval has passed.
- [ ] **Pull requests ask about documentation**
  The template asks whether an interface, dependency, deployment or decision changed and which document was updated.
- [ ] **Links are checked automatically**
  Broken internal links and references to deleted files fail the build.
- [ ] **Docs point to code and code points to decisions**
  Paths in the documentation, ADR numbers in code comments and commit messages.
- [ ] **The published site is rebuilt from main**
  Readers always see the version that matches the deployed code, with no manual publishing step.
- [ ] **Stale pages are deleted or archived**
  Pages nobody will fix are removed; ADRs are superseded or deprecated, never deleted.
- [ ] **At least one check compares documentation with reality**
  For example, a test that fails if a dependency appears in the code that the container diagram does not show.

WATCH OUT

Common mistakes

- Choosing a tool before deciding what to document. The tool matters far less than the habits of ownership and review.
- Putting everything in one giant diagram source that nobody can read as text or as a picture.
- Committing rendered images without their sources, or sources without a build that renders them, so the two drift apart.
- Treating diagrams as code as an excuse to skip legends, titles and labels: the C4 notation rules still apply to generated diagrams.
- Making documentation changes optional in review, so that they are always the first thing dropped under deadline pressure.

CHECK YOUR UNDERSTANDING

▼ What can a reviewer see in a pull request when a diagram is code that they cannot see when it is an exported image?

A readable line-by-line diff of what changed, such as a new relationship between two services, in the same pull request as the code that introduced it.

▼ Why does renaming a service cause more work with a drawing tool than with a modelling tool?

In a drawing tool each diagram holds its own copy of the label, so every diagram must be found and edited; in a modelling tool the element is defined once and every view picks up the new name.

▼ Why should stale documentation be deleted rather than left in place, and what is the exception?

Readers act on what they read, so wrong documentation causes wrong decisions. ADRs are the exception: they record history and are marked superseded or deprecated instead.

▼ Name two checks a documentation pipeline can run besides rendering diagrams.

Link checking, prose or spelling linting, a strict site build, a warning for overdue review dates, and comparing documented dependencies with those in the code.

▼ **Try it: Your team has eight diagrams in a drawing tool. After renaming a service, three still show the old name, and a dispatch arrow differs between two. What would you change, in tooling and in habits, and why?**

Drawing tools know only shapes, so each picture drifts separately. Move to a model, such as Structurizr DSL, kept in the repository: define each element once and generate the views, so a rename reaches every diagram. Render them in the pipeline on each pull request. Tools alone are not enough, so name an owner, review the docs with code changes, show a last-reviewed date and delete stale pages. PlantUML or Mermaid would suit one-off sequence diagrams.

SOURCES

- Docs as Code, Write the Docs: https://www.writethedocs.org/guide/docs-as-code/
- Structurizr DSL language reference: https://docs.structurizr.com/dsl/language
- Mermaid documentation: https://mermaid.js.org/
- PlantUML: https://plantuml.com/
- Include diagrams in your Markdown files with Mermaid, GitHub Blog (2022): https://github.blog/2022-02-14-include-diagrams-markdown-files-mermaid/

SECTION 4.8 OF 4.8 · INTERMEDIATE · 24 MIN READ · LAST REVIEWED NOVEMBER 2026

Communicating architecture to different audiences

BEFORE YOU READ THIS

This section builds on:

- 1.5 Stakeholders and concerns
- 1.6 Decisions and trade-offs
- 4.3 The C4 model

Terms it uses from earlier: Trade-off, Concern, C4 model, Weighted decision matrix, Last responsible moment

THE IDEA IN BRIEF

A tour guide explains the same cathedral differently to schoolchildren, architects and pilgrims: different facts, different words, different routes through the building. Software architects have to do the same. A finance director wants to know what a design costs and what could go wrong; a developer wants to know how the pieces connect; an auditor wants evidence. Good communication starts by asking who is listening and what they need to decide, then shows them just that, clearly and honestly, including the downsides.

WHAT YOU NEED TO KNOW

- Communicating the architecture is a core architectural task; start every explanation by naming the audience and what they must decide or do afterwards.
- Executives need the recommendation, cost, risk and the decision required; developers need structure, interfaces and reasons; operations, security and auditors each need views of failure, trust boundaries and evidence.
- Explain trade-offs as options, criteria, recommendation and consequences, leading with the answer and stating plainly what is being given up.
- A one-page architecture summary and a numbered follow-one-request story make any system understandable; label every box and line when sketching, and capture conclusions after the whiteboard.
- Design documents and RFCs propose a change in writing for comment before it is built; ADRs then record the decisions permanently.
- Clear diagrams have a title, a key, consistent notation, few meaningful colours, a single flow direction and labelled arrows; disagreements are resolved by agreeing criteria, gathering evidence, seeking rough consensus and recording dissent.

Communication is the job

An architecture that only its architect understands has failed, however good it is. Decisions shape what dozens of people build, operate, fund and audit, and they work only if those people understand them well enough to act on them and to notice when reality departs from them. This is why the iSAQB curriculum lists communicating the architecture among an architect's core tasks, and why the architect role described in Chapter 1 depends so heavily on interpersonal skills.

The earlier sections of this chapter supplied the material: views, C4 and UML diagrams, ADRs, an arc42 structure and a pipeline that keeps them current. This section is about using that material with people. The core skill is the same in every setting: decide who the audience is, decide what they need to do after hearing you, and give them exactly what that requires, in the form they can take in. Gregor Hohpe's image of the *architect elevator* (*The Software Architect Elevator*, 2020) captures the range: an architect rides between the boardroom, where strategy is set in business terms, and the engine room, where systems are built, and must be understood on every floor.

Tailoring to the audience

Every audience has its own concerns, in exactly the sense of stakeholders and concerns, and its own vocabulary and attention span. The table is a starting point; the real skill is asking, before you prepare anything, “what will this person decide or do differently after this conversation?”

What different audiences need

Audience	What they care about	What to show	What to avoid
Executives and sponsors	Cost, risk, time to market, strategic options, what they must decide	One page: the recommendation, the business impact, the cost, the top risks and the decision you need from them; at most a system context diagram	Technology names without business meaning; component diagrams; a tour of the options before the answer
Product managers	What becomes possible or harder, delivery dates, user-visible behaviour	Capabilities and constraints in product terms; how the architecture affects the roadmap; a runtime story of a user journey	Internal structure for its own sake
Developers	How to build within the architecture: boundaries, interfaces, conventions, why things are the way they are	C4 container and component diagrams, sequence diagrams, ADRs, code examples, the decision log	Vague principles without examples; diagrams that do not match the code
Operations and SRE	Deployment, failure modes, monitoring, capacity, recovery	Deployment diagrams, runtime failure scenarios, SLOs, runbooks, dependency maps	Happy-path-only flows
Security	Trust boundaries, data flows, identities, attack surface	Data-flow diagrams with trust boundaries marked, authentication and authorisation flows, threat model	Diagrams that hide where personal or payment data goes
Auditors and regulators	Evidence that controls exist and are followed; traceability	Versioned documents with owners and review dates, ADRs, mapping of requirements to controls	Informal sketches with no date, owner or version

Two habits help with every audience. **Lead with the answer.** Busy readers should learn the conclusion in the first sentence, then the supporting reasons, then the detail; this is the structure Barbara Minto taught consultants as the Pyramid Principle, and it is the same idea as Fowler's inverted pyramid for ADRs. **Use their words.** To a finance director, “eventual consistency” is noise; “the restaurant sees a new order up to two seconds after the customer does, and nothing is lost” is a fact they can weigh.

Explaining trade-offs

Most architectural communication is, at heart, explaining a trade-off. Non-specialists distrust recommendations that seem to have no downside, and they are right to. A clear structure, used consistently, makes trade-offs easy to follow: the **options** (including doing nothing), the **criteria** that matter to this audience, how each option fares, the **recommendation** and its **consequences**, including what you are giving up. This is the anatomy of a decision from Chapter 1 and the content of an ADR, reshaped for a listener.

WORKED EXAMPLE

Worked example: explaining a trade-off to the leadership team

The food-delivery platform has had two Friday-evening outages caused by a slow notification service. The architect has five minutes at the leadership meeting to get approval for ADR-0012 (asynchronous order events).

Answer first	"I recommend we change how orders flow between our systems so that a failure in notifications or courier dispatch can no longer stop customers ordering. It costs about six team-weeks and a few hundred pounds a month."
Why now	"On Friday 21 August a notification outage blocked every order for 23 minutes during the Friday peak. At about 40 orders a second, that is roughly 55,000 orders that could not be placed, followed by refunds and complaints."
Options in their terms	"We could (A) keep the current design and add safety limits, which is cheap but still lets a slow service slow every order; (B) the recommended change; or (C) let other systems read our order data directly, which is quick but makes every future change riskier."
What we give up	"Restaurants will see new orders a second or two after the customer's confirmation, and we take on one more managed service to run. We think both are acceptable; operations agrees."
The decision needed	"I need agreement to spend six team-weeks this quarter, which delays the loyalty feature by two weeks."

No technology is named: "message broker" and "transactional outbox" stay in the ADR. The order figure (40 orders a second for 23 minutes) is checked against real data before the meeting.

The cost of the recommendation and the alternatives are stated plainly, so the leaders can disagree on the merits rather than on trust.

The one-page architecture summary

Every system benefits from a one-page architecture summary: a single page that a newcomer, an executive or a reviewer from another team can read in five minutes. It is not a substitute for the full documentation but the front door to it. A useful structure is: what the system does and for whom (two sentences); the top three quality goals; a system context diagram; a container diagram or a list of the main building blocks; the five or so decisions that shape everything (one line each, linking to ADRs); the top risks; and where to find more. The arc42-lite page in arc42 and templates is one such summary.

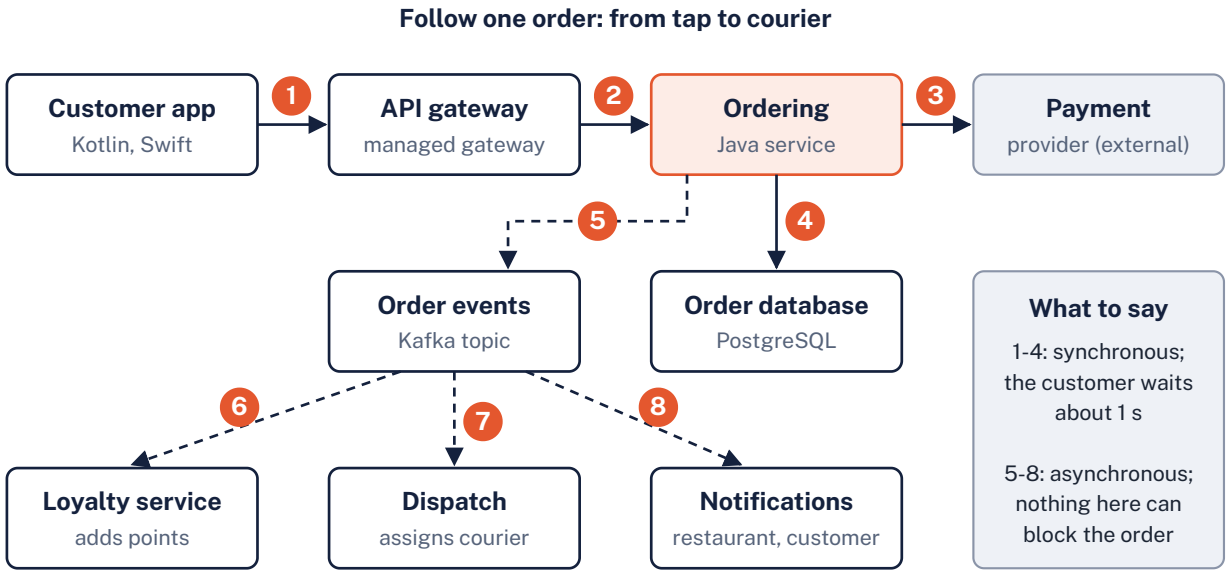
The discipline of fitting the architecture onto one page is valuable in itself. If the main ideas cannot be stated in a page, they are probably not clear to the team either.

Storytelling: follow one request

People remember stories better than structures. A powerful technique for explaining any system is to follow one request: pick a single, concrete, important event, such as a customer placing an order, and walk through what happens to it, step by step, across the architecture. A static box diagram shows what exists; the story shows why it exists and how the parts cooperate. It also reveals the interesting moments naturally: where the customer waits, where data crosses a trust boundary, what happens if the payment provider is slow.

Choose the story for the audience. For developers, the happy path then a failure (“now the payment provider times out”). For operations, the failure and recovery. For executives, the customer's experience with only the business-relevant moments. Number the steps on the diagram so that you can point to them, and keep the same diagram on screen while you talk rather than flipping between slides.

Following one order through the platform



Solid arrows: synchronous calls. Dashed arrows: asynchronous events. Numbers: order of the story.

A numbered story diagram: steps 1 to 4 happen while the customer waits; steps 5 to 8 happen asynchronously and cannot block the order.

Whiteboarding and sketching well

Much architectural communication happens live, at a whiteboard or on a shared online canvas, and a few habits make sketches far more useful. **Say what you are drawing before you draw it** (“this is the container level: things we deploy separately”), so the audience knows the level of abstraction. **Start with the context**: the users and external systems at the edges, then the inside. **Label every box and every line** as you draw, including the direction and what flows; an unlabelled arrow will be read in three different ways by three people. **Draw a small key** in a corner when you use more than one kind of line or box. **Leave space**, because the best ideas arrive mid-discussion and need somewhere to go.

Sketches are thinking tools and are meant to be thrown away, but the conclusions are not. Photograph the board, and within a day turn anything that will matter into a proper diagram, an ADR or a change to the documentation. Simon Brown, who created the C4 model, observed that many teams' whiteboard diagrams are ambiguous boxes and lines precisely because they lack an agreed set of abstractions; using the C4 levels on the whiteboard is one of the simplest ways to fix that.

Design documents and RFCs

Many engineering organisations use a written design document, often called an RFC (request for comments), to propose a significant change before building it. The name echoes the Internet's RFC series, which began with RFC 1 by Steve Crocker in April 1969. The author writes the problem, the context, the goals and non-goals, the proposed design, the alternatives considered and the cross-cutting concerns such as security, privacy and observability; the document is circulated to the people affected, who comment in writing; and it is approved or revised. Malte Ubl's widely read description of design docs at Google (2020) lists exactly those sections and suggests ten to twenty pages for large projects and one to three pages for small “mini design docs”. Gergely Orosz described how Uber used an RFC process from its early years to spread knowledge and catch problems early (2018).

Amazon takes writing further: its 2017 shareholder letter explains that the company does not use slide presentations in meetings but “narratively structured six-page memos”, which attendees read silently at the start. The benefit common to all these practices is that writing forces clear thinking, written comments let quieter and remote colleagues contribute, and the document becomes a record. The risks are also common: long documents nobody finishes, review processes that become bureaucratic gates, and design docs that are never updated after implementation.

Design docs and ADRs complement each other. A design doc explores a problem and proposes a whole design, often covering several decisions; once it is approved, the significant decisions it contains are captured as ADRs, which are shorter, permanent and indexed. A reasonable rule is: design doc to think and persuade, ADR to remember.

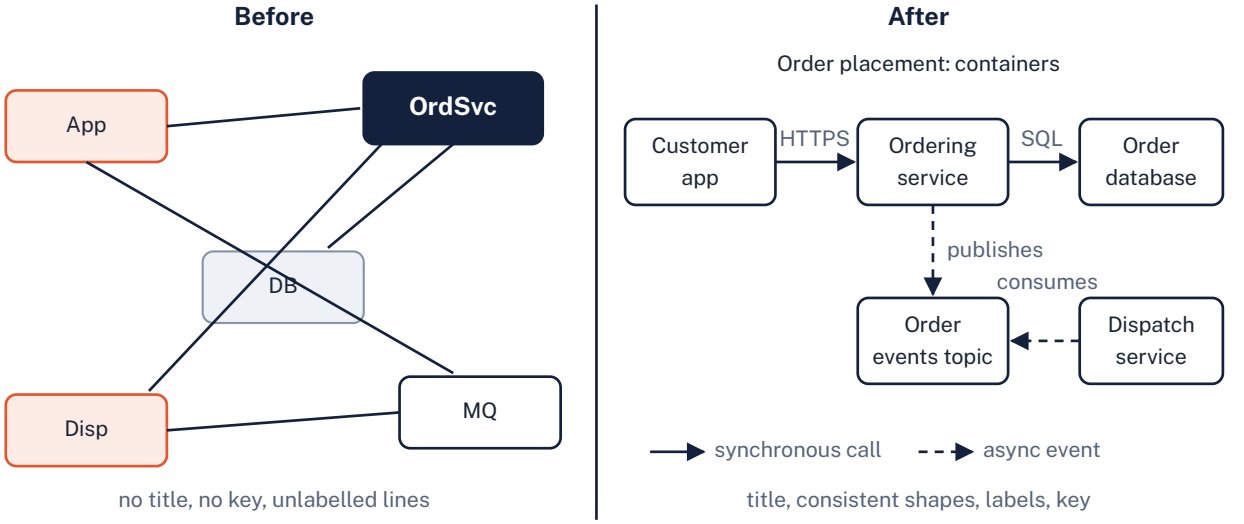
Presenting at an architecture review

An architecture review is a meeting in which other experienced people examine a design to find risks before they become expensive. Structured methods such as ATAM and lighter-weight reviews are covered in Chapter 11: lightweight reviews; here the concern is how to present well. Send the material at least two days ahead: the one-page summary, the key diagrams, the relevant ADRs and the quality scenarios. State at the start what you want from the review: approval, a list of risks, or a choice between two options. Present the drivers before the design, because a reviewer cannot judge a design without knowing which qualities it must achieve. Walk through one or two scenarios using the follow-one-request technique, including a failure. Show the trade-offs you made and the risks you know about; a reviewer who finds a risk you have already listed trusts you more, not less. Record every issue raised, with an owner, and send the list afterwards.

Visual design rules for diagrams

Diagrams communicate before anyone reads a word, so their visual design matters. The rules below are drawn from the C4 notation guidance in the C4 model and from general information design. The before-and-after diagram applies them to the same five elements.

The same diagram, before and after



On the left, colour and abbreviations carry no meaning and lines cross unlabelled; on the right, a title, consistent shapes, a left-to-right flow, labelled arrows and a key.

Rules for clear architecture diagrams

1. **Give every diagram a title**
   State the type and scope, such as “Containers: food-delivery platform”, so the reader knows the level of abstraction.

2. **Use one notation consistently**
   Same shape for the same kind of thing across all diagrams; do not mix C4 and UML symbols on one diagram without saying so.

3. **Include a key**
   Explain every shape, colour, line style and arrowhead that carries meaning. If you cannot explain it, remove it.

4. **Limit colours and give them meaning**
   Two or three colours, each meaning something (external, new, at risk). Never rely on colour alone: diagrams are printed in black and white and many readers have colour vision deficiency.

5 **Flow left to right or top to bottom**
Follow the reading direction of your audience, with requests flowing one way, so the eye follows the story.

6 **Label every element and arrow**
Names without unexplained abbreviations; arrows one-directional, with what flows and, between containers, the protocol.

7 **Avoid crossing lines where possible**
Rearrange boxes first; if crossings remain, split the diagram. A diagram with many crossings usually tries to show too much.

8 **Keep it to one screen**
Roughly a dozen or two elements at most. Larger pictures belong at a lower level or in several diagrams.

IF YOU ARE NEW TO THIS

Step by step: deciding when not everyone agrees

The idea that needs the most care here is rough consensus: a decision can be made once each objection has had a fair hearing and a reasoned reply, even if some still prefer another option. Imagine how one proposal might go.

- The ordering team opens ADR-0012, publishing order events through a message broker, as a proposed record in a pull request. The restaurant, courier and notifications teams and SRE are asked to comment.
- Two objections arrive. A notifications engineer prefers to keep the synchronous calls with timeouts and retries, the option needing least new infrastructure. A restaurant-team engineer worries that tablets will show new orders late.
- Each objection is answered in writing. The record explains that with synchronous calls a slow service still delays every order, and admits that a restaurant tablet may lag a second or two behind the customer's confirmation: a cost it records.
- The notifications engineer is still not convinced. Their objection has had a fair hearing and a reply, but is not accommodated: the design does not change to suit them. For rough consensus, that is enough.
- Before deciding, the deciders ask: 'Can anyone not live with this, and why?' No new blocking issue comes up. Had an objection been left unanswered, there would be no rough consensus yet, however many people were in favour.
- The record is accepted with the engineer's dissent and reasons written into it. The engineer commits to making it work; if they are later proved right, the record shows what to revisit.
- **The lesson:** aim to answer every objection, not to make everyone agree. Record the dissent with the decision, and settle in advance who decides if consensus cannot be reached.

Handling disagreement and building consensus

Disagreement about architecture is normal and healthy: it usually means people see different risks. The aim is not to win but to reach a decision that the people who must live with it understand and will support. Several practices help.

Separate interests from positions. “We must use Kafka” is a position; “I am worried we will lose events during deploys” is an interest, and interests can often be met in several ways. **Agree the criteria before comparing options**, as in a weighted decision matrix; most heated arguments are really disagreements about what matters, and they cool down once that is

explicit. **Resolve uncertainty with evidence**: a two-day spike or a load test settles many debates that meetings cannot. **Write it down**: a proposed ADR or design doc lets everyone comment in their own time and gives quieter colleagues a voice.

Aim for consent rather than unanimity. The IETF's RFC 7282 (Pete Resnick, 2014) describes rough consensus: a decision is reached when every objection has been addressed, which means considered and answered, though not necessarily accommodated. Asking “can anyone not live with this, and why?” surfaces blocking issues faster than asking whether everyone agrees. When consensus is not possible, someone must decide, and it should be clear in advance who. Jeff Bezos's 2016 shareholder letter popularised *disagree and commit*: people state their disagreement openly, then commit fully to the decision once it is made. Record the dissent and the reasons in the ADR; if the dissenters prove right, the record shows what to revisit and why.

Two failure modes deserve a warning. The architect who wins every argument by seniority stops hearing about problems. The team that cannot decide without unanimity lets important decisions be made by default, which is the opposite of the last responsible moment.

Checklist: an architecture presentation

Use this before any presentation of architecture, from a five-minute leadership update to a formal review.

- [ ] **Audience and outcome named**
  You can say in one sentence who will be in the room and what they should decide or do afterwards.
- [ ] **Answer first**
  The recommendation or main message is on the first slide or in the first paragraph.
- [ ] **Drivers before design**
  The business goals, quality goals and constraints are shown before the solution.
- [ ] **Level of abstraction matches the audience**
  Context level for executives, container and component levels for engineers; no more than two levels in one session.
- [ ] **One story told end to end**
  A follow-one-request walk-through, including one failure for technical audiences.
- [ ] **Trade-offs and risks shown honestly**
  Options, what each costs and what you are giving up; known risks listed before anyone else finds them.
- [ ] **Every diagram has a title, key and labels**
  Consistent notation, limited colours with meaning, readable from the back of the room.
- [ ] **Material sent in advance**
  For reviews and decisions, the summary and ADRs go out at least two days before, so the meeting is for discussion.
- [ ] **The ask is explicit**
  Approval, risks, a choice between options or simply awareness; say which, and by when.
- [ ] **Follow-up captured**
  Questions, objections and actions recorded with owners; decisions turned into ADRs; documentation updated.

CHECK YOUR UNDERSTANDING

▼ What question should you answer before preparing any architecture explanation?

Who the audience is and what they should decide or do differently afterwards; the content, level of abstraction and vocabulary follow from that.

▼ Why tell a follow-one-request story rather than only showing a box diagram?

A box diagram shows what exists; the story shows how the parts cooperate and why, and naturally reveals waits, trust boundaries and failure points that a static picture hides.

▼ How do a design document and an ADR differ in purpose?

A design document explores a problem and proposes a whole design for comment before building; an ADR is a short, permanent record of one decision. Approved design docs typically produce several ADRs.

▼ What does rough consensus mean in RFC 7282?

Every objection has been considered and answered, though not necessarily accommodated; it is not unanimity or a majority vote.

▼ Try it: An architect tells the leadership team: "We need a transactional outbox and a broker to decouple our synchronous order path." Rewrite this for them in about three sentences, including the recommendation, the cost and what they must decide.

"I recommend changing how orders pass between our systems, so that a fault in notifications can no longer stop customers ordering, as it did twice recently. It would take about six team-weeks and a small monthly running cost; the catch is that some updates would arrive a few seconds later. Please approve this today." The answer comes first, in business terms, with costs and the downside stated plainly and no jargon.

SOURCES

- C4 model: notation: https://c4model.com/diagrams/notation
- Design Docs at Google, Malte Ubl (2020): https://www.industrialempathy.com/posts/design-docs-at-google/
- Scaling Engineering Teams via RFCs: Writing Things Down, Gergely Orosz: https://blog.pragmaticengineer.com/scaling-engineering-teams-via-writing-things-down-rfcs/
- RFC 7282: On Consensus and Humming in the IETF: https://www.rfc-editor.org/rfc/rfc7282.html
- Amazon 2017 letter to shareholders: https://www.aboutamazon.com/news/company-news/2017-letter-to-shareholders

Further reading

- *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 most thorough treatment of architecture documentation: views, styles, the seven rules for sound documentation and a documentation template.
  https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd edition*, Nick Rozanski and Eóin Woods (2012), Addison-Wesley. A practical catalogue of seven viewpoints and of perspectives such as security and performance, aimed at large information systems.
  https://www.viewpoints-and-perspectives.info/
- *Architectural Blueprints: The "4+1" View Model of Software Architecture*, Philippe Kruchten (1995), IEEE Software 12(6). The short paper that popularised multi-view architecture description and scenario-based validation.
  https://arxiv.org/abs/2006.04975
- *ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description*, ISO/IEC JTC 1/SC 7 and IEEE (2022), ISO. The international standard defining stakeholders, concerns, viewpoints, views and model kinds.
  https://www.iso.org/standard/74393.html
- *The C4 Model: Visualizing Software Architecture*, Simon Brown (2026), O'Reilly Media. The creator's own account of the C4 abstractions, diagrams and notation; the website covers the essentials free of charge.
  https://c4model.com/
- *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edition*, Martin Fowler (2003), Addison-Wesley. A short, practical guide to the parts of UML worth knowing, and the source of the sketch, blueprint and programming-language distinction.
- *Living Documentation: Continuous Knowledge Sharing by Design*, Cyrille Martraire (2019), Addison-Wesley. How to derive documentation from the system itself and keep hand-written material to what only people can supply.

Glossary

4+1 view model

Philippe Kruchten's 1995 approach that describes a system through logical, process, development and physical views, tied together by a small set of scenarios.

Activity diagram
A UML diagram showing a workflow as actions, decisions, parallel forks and joins, often divided into swimlanes for who does what.

Advice process
A decentralised way of making decisions in which anyone may decide, provided they first seek advice from those affected and from relevant experts.

Allocation view
A view that maps software onto things outside it: infrastructure (deployment), files and builds (implementation) or teams (work assignment).

arc42
A free, twelve-section template for documenting and communicating software architecture, created by Gernot Starke and Peter Hruschka.

Architecture description framework
In ISO/IEC/IEEE 42010:2022, an established set of conventions for producing architecture descriptions, such as viewpoints, templates and rules.

Building block
In arc42, any part of a system's static structure, such as a subsystem, service, component or module, shown in the building block view.

C4 model
Simon Brown's approach to architecture diagrams: a hierarchy of system context, container, component and code diagrams, plus a few supplementary diagrams.

Component (C4)
In the C4 model, a group of related functionality behind a clear interface, running inside a container and deployed with it.

Component-and-connector view
A view of what exists at runtime, such as services, clients, databases and queues, and the connectors (calls, messages, shared data) between them.

Container (C4)
In the C4 model, a separately runnable or deployable unit that runs code or stores data, such as a web app, mobile app, service or database. It is not a Docker container.

Container diagram
A C4 diagram showing the apps, services and data stores inside one system, with their technologies and how they communicate.

Correspondence
A recorded relation between elements of an architecture description, for example between a container in one view and the server it runs on in another.

Cross-cutting concept
A rule or solution that applies across many parts of a system, such as security, error handling, logging or persistence.

Decision driver
A force or criterion, such as a quality goal or constraint, that an architectural decision must satisfy and against which options are compared.

Decision log
The ordered collection of a system's ADRs, usually with an index listing each record's number, title, status and date.

Deployment diagram
A diagram (in C4 or UML) showing how software is mapped onto infrastructure in one environment, including nodes, replicas and network paths.

Design document (RFC)
A written proposal for a significant change, covering problem, goals, design and alternatives, circulated for comment before the change is built.

Diagrams as code
Writing diagrams as text in a small language, such as PlantUML, Mermaid, D2 or Structurizr DSL, which a tool lays out and draws.

Docs as code
Producing documentation with the same tools and workflow as software: plain-text files in version control, code review, automated checks and publishing.

Domain model
A small diagram or description of the key business concepts of a system, their attributes and how they relate, shared by developers and domain experts.

Drawing tool
A diagramming tool that knows only shapes, lines and text, so the same element drawn in two diagrams has no link between them.

Dynamic diagram
A supplementary C4 diagram showing how elements collaborate at runtime in one scenario, usually as numbered interactions.

Entity of interest
In ISO/IEC/IEEE 42010:2022, the thing whose architecture is being described: a software system, a system of systems, a product line or an enterprise.

Follow one request
Explaining a system by tracing a single concrete event, such as one order, step by step through every part it touches.

Guard
A condition in square brackets on a transition or flow that must be true for it to be taken, such as [code matches].

Just-enough documentation
Documentation sized to what its readers actually need, so that writing and maintaining more would cost more than it returns.

Lifeline
In a sequence diagram, the dashed vertical line below a participant, along which its messages and activations are shown in time order.

Living documentation
Documentation that evolves with the system it describes, often because it is generated from or checked against the system itself.

MADR
Markdown Architectural Decision Records: a widely used ADR template with decision drivers, considered options, the outcome, consequences and confirmation.

Model kind
In ISO/IEC/IEEE 42010, the conventions for one type of model, such as a UML sequence diagram or a C4 container diagram.

Modelling tool
A tool that stores elements and relationships in a model and generates diagrams as views of it, keeping every view consistent.

Module view
A view of the units of implementation (packages, libraries, layers) and their dependencies, as they exist at development time.

Multiplicity
In a UML class diagram, how many objects may take part at each end of an association, such as 1, 0..1, 0..* or 1..*.

One-page architecture summary
A single page giving a system's purpose, quality goals, context, main building blocks, key decisions and risks, as the front door to fuller documentation.

Perspective
In Rozanski and Woods's approach, a quality concern such as security or performance that is applied across all views, rather than drawn as a view of its own.

Provided interface
In a UML component diagram, an interface a component offers to others, drawn as a small circle (a ball) on a stick.

Required interface
In a UML component diagram, an interface a component needs another component to provide, drawn as a half-circle (a socket).

Rough consensus
A decision reached when every objection has been considered and answered, though not necessarily accommodated; described in IETF RFC 7282.

Sequence diagram
A UML diagram showing the messages exchanged between participants over time for one scenario, with time running down the page.

Software design description (SDD)
A document describing a software design in detail, whose content IEEE Std 1016-2009 specifies; common in contract-driven projects.

Software system (C4)
In the C4 model, the highest level of abstraction: something that delivers value to its users, usually owned by one team or organisation.

State machine diagram
A UML diagram showing the states an object can be in and the transitions between them, each triggered by an event.

Structurizr
Simon Brown's tooling for the C4 model, in which a text-based model generates many consistent diagrams.

System context diagram
A C4 diagram showing one system as a single box, the people who use it and the other systems it interacts with.

Unified Modeling Language (UML)

A general-purpose graphical modelling language standardised by the Object Management Group. The current version is 2.5.1 (2017).

View component

In ISO/IEC/IEEE 42010:2022, one part of a view, such as a diagram, table or model, governed by a model kind. The 2011 edition called it an architecture model.

Y-statement

A one-sentence decision record by Olaf Zimmermann: in the context of, facing, we decided for, and neglected, to achieve, accepting that.