Engineer

architecture

how a system's parts fit together and why

What it is

Architecture, in software often called software architecture or system architecture, is the set of fundamental concepts and properties a system has in its environment, carried by its elements, the relationships between them and the principles that govern how it is designed and how it changes. That wording follows the definition in NIST SP 800-160 Volume 1 Revision 1, Engineering Trustworthy Secure Systems, published in November 2022.

Each part of the definition does work. The elements are the pieces: services, databases, queues, the frontend, the modules inside them. The relationships are how those pieces depend on one another and what passes between them. The principles are the rules that decide where new work goes, and they keep the elements and relationships from drifting as the system grows. “In its environment” makes the users, the neighbouring systems and the constraints part of the architecture, and “evolution” means a description that is true only on the day it was drawn is incomplete.

Not every design choice is architectural. The group that maintains the Architectural Decision Record (ADR) format calls an architectural decision “a justified design choice that addresses a functional or non-functional requirement that is architecturally significant”, and a requirement is significant when its effect on the system’s architecture and quality can be measured. The arc42 template is blunter: its decisions section is for “important, expensive, risky or contentious decisions”.

Because no single drawing shows all of it, architecture is described in views. The C4 model, created by Simon Brown, offers diagrams that work like maps you zoom in and out of: a system context diagram showing how the system sits among its users and the systems around it, a container diagram showing the

applications and data stores inside it, a component diagram for one container and, where it helps, a code diagram. arc42 arranges the written description in twelve sections, from goals and constraints to decisions, risks and technical debt.

Use the concept in a decision

Put “What is architecture?” to work in a real decision. We can help relate the concept to the system you are building or buying. The shape of a system worked out on paper first, so the parts fit and the cost is known before anyone writes code.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. One infrastructure plan for 14 departments — 14 research departments on one infrastructure plan

One infrastructure plan for 14 departments Get in touch →

Why it matters

The decision architecture puts in front of a team is less which structure to choose than which choices to make now, deliberately and on the record, and which to leave open.

Settle a structural question early and you pay before you know much: the choice is made with the least information the project will ever have, and it slows the first weeks. Leave it open and the structure gets settled anyway, one convenient change at a time, by whoever touched the code last. The way out is to separate two kinds of choice. A few are expensive to change once code depends on them: how the system divides into parts, which part owns the business rules, how data is reached, what bar every change must clear. Those deserve to be made early and on purpose. Most of the rest are cheap to revisit and can wait until the code shows what it needs.

Recording is the second half of the decision, and it is where the reasoning survives. An ADR captures one decision with its rationale, its trade-offs and its consequences, and the collection of a project’s ADRs forms its decision log. The ADR site credits Michael Nygard’s 2011 post “Documenting Architecture Decisions” with popularising the idea. The cost is a page of writing per decision. The return is that a later reader can tell a deliberate trade from an accident, and can check whether the reason still holds.

Quality goals decide between structures that would all work. arc42 asks for the top three, at most five, quality goals for the architecture, the ones the major stakeholders care about most, and warns that they are distinct from the project’s

goals. Two designs with the same features can differ completely in how they fail, scale or change, and only the quality goals say which difference counts.

What it looks like in practice

The platform's repository opened on 25 November 2025 with nothing in it, and the page on its founding decisions argues that such choices are "cheap to make on day three and close to impossible to reverse by month six". Two of the three made then shaped the structure. The stack was layered so that each part has one job: a Next.js frontend, a Go API that validates and forwards, and a PostgreSQL function layer that owns the business rules. Data access went behind stored functions from the start, because retrofitting it later "would have meant rewriting every handler". Nine months and several thousand commits later, the page reports, the layers have not blurred and no handler talks to a table directly. Its test for a founding decision is not whether it sounded right but whether it survived the volume of work that came after.

Structure left to chance gets set by accident, as the page on GIS map development shows. The map had grown up in pieces: spatial data in PostgreSQL, drawing in Mapbox, ReactJS on the front and NodeJS behind. Each part worked on its own; they "just hadn't been built to fit together". The role covered the full-stack development and the technical direction, and the architecture was set first. After that, responsibilities "kept getting moved back to the layer they belonged in whenever one started leaking into the next", and much of the work was "catching the small inconsistencies before they hardened into architecture".

The page on the container plane treats the written reason as the real result. The industry-default container engine publishes container ports above the host firewall's rules, so the plane is a daemonless engine driving units generated by the operating system's own supervisor, with no published ports at all. One trade is recorded as such: units run at the system level rather than rootless, which keeps the automation simple at the cost of the stricter isolation rootless would give, and the note says which way to move if container escape ever matters more. That is what an ADR holds: a decision with its rationale, its trade-offs and its consequences. The page rates writing the choice down with its reason above the choice itself, since "the next person will be told to use the default by every article they read".

Where it goes wrong

A careful newcomer reverses a decision that was deliberate. The reason lived only in someone's head, so the choice looks like a mistake, and the default every tutorial recommends looks like the fix. Write an ADR for each decision that is expensive to reverse: the context, the options weighed, the one chosen, its consequences and what would make it worth revisiting. Keep the records beside the code, so a change to one is reviewed with the other.

Business rules show up in three layers at once. Each change put its logic wherever it was convenient that afternoon, until the same rule was checked in the frontend, the API and the database, a little differently in each. Name what each layer owns, write it down, and move a responsibility back as soon as it leaks, while the stray copy is still small. Where the platform allows, enforce the boundary too, for example by letting the API call database functions while withholding rights on the tables.

A rule introduced in month six turns into a cleanup project. The decision everything depends on was left open, and every commit since assumed whatever happened to be true. Decide the few expensive things first and leave the rest open on purpose. Say so in the decision log, so an open question is visibly open rather than quietly answered by the next change.

The architecture diagram explains nothing to anyone. One drawing mixes users, services, classes and tables, and the lines between the boxes carry either no label or the word "uses". Draw one diagram per level of zoom, as C4 does, and label every line with the direction and intent of the relationship; between containers, name the technology or protocol too. C4's notation guidance asks for exactly that, and advises against single-word labels such as "Uses".

Further reading

- architecture, NIST CSRC Glossary — the definition used above, beside the FIPS and security-architecture definitions that share the entry.
- NIST SP 800-160 Vol. 1 Rev. 1 — Engineering Trustworthy Secure Systems, the publication the definition comes from.
- Architectural Decision Records — what makes a decision architectural, templates for recording one, and tools that help.
- The C4 model — four levels of diagram, from system context down to code, with notation advice and a review checklist.

- arc42 — practical tips and examples for each of the twelve sections of the arc42 template.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is architecture?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/architecture/