

Architecture

The Software Engineering Handbook

A Judgment and Decision-Making Approach

James C. Davis

Chapter 7 of the full book · Edition 1 · 2026

Architecture

Premise. *Architecture organizes one acceptable realization so that its obligations can coexist.*

Specification tells us which realizations we would accept. It deliberately leaves many choices open. Architecture begins when engineers select one acceptable realization and decide how it should be organized.

A specification may separately describe allowable states, required ordering, valid information, timing bounds, privacy obligations, expected changes, and other properties. Those separate views make individual obligations easier to reason about. But nobody ships a collection of views. The implemented system must realize them together.

State transitions run on the same processors that must satisfy timing obligations. Data described by a schema lives in the same storage governed by privacy requirements. Components that must remain independently changeable may also need to coordinate at runtime. Architecture is where these obligations meet.

DEFINITION • Software architecture

The architecture of a software system is its consequential organization: its major parts, the responsibilities assigned to them, their interfaces, and the rules governing how they may interact.

Suppose a service contains two helper functions. Renaming one or moving it into another source file changes the organization of the program, but usually not its architecture. Splitting the service into two independently deployed processes is different: the new boundary changes communication, failure, deployment, ownership, and perhaps consistency. Later engineering must now work within those consequences.

The word consequential matters. Thousands of structural facts are true of any software system. Architecture concerns the organizational decisions whose consequences matter enough that later engineering should treat them as constraints or affordances.

From specification to one system

The previous chapter (Specification) described specification as a boundary around acceptable realizations. Several different systems might satisfy every stated obligation: Requirements → Specification → Acceptable realizations A, B, C.

Architecture does not extend the specification until only one realization remains. Instead, engineers choose among acceptable realizations and establish the consequential organization of the system they intend to build.

A useful architecture answers four questions:

- What are the major parts, and what is each responsible for?
- Where should the boundaries go?
- How may the parts interact?
- What properties must the organization support?

The first three questions describe structure. The fourth explains why the structure matters. An architectural organization can support or frustrate performance, security, reliability, modifiability, consistency, failure isolation, expected change, and many other properties. Architecture is therefore not box drawing. The boxes and arrows matter only when the distinctions they represent change what becomes easy, difficult, visible, or expensive.

Architecture creates possibilities and constraints

Consider a building architect deciding where a wall should stand. The architect may determine the wall's position and thickness, the space available for building services, and where penetrations are permitted. Those decisions constrain the plumbing without designing it. Many plumbing systems may remain possible, but some routes, pipe sizes, and arrangements have become easy, expensive, or impossible because of the architectural decision.

Software architecture works similarly. Architecture organizes responsibilities, boundaries, and interactions without determining the mechanisms by which each part will fulfill its responsibility. It creates a structured space of possibilities for later Design. A good architectural decision makes important system properties tractable while preserving useful freedom where the choice need not yet be made.

Where should the boundaries go?

One of the most consequential architectural decisions is where to draw boundaries. A boundary is a promise about reasoning and consequence: things on the same side may be understood, changed, and operated together; things on opposite sides should be understandable apart, and consequential effects should cross the boundary only through the interactions it permits.

Suppose the same acceptable system can be organized in two ways. One organization separates collection, classification, summarization, and advice into distinct parts. Another groups collection with storage, groups modeling with advice, and isolates sharing. Neither organization eliminates dependencies among these responsibilities. Each chooses where those dependencies will live. A change to the advice representation may remain local in one organization and cross several boundaries in the other. Failure of storage may affect different capabilities. One decomposition may make independent scaling easy while another simplifies consistency. Architecture does not eliminate coupling. Architecture allocates coupling.

When considering a boundary, ask which responsibilities should change together, fail together, scale together, remain consistent together, be secured together, or be owned together. Evidence that two responsibilities should vary independently argues for separating them. Evidence that they must coordinate tightly argues for keeping them together or accepting the cost of coordination across the boundary.

These forces rarely point in the same direction. For example, suppose two responsibilities change for different reasons and should fail independently. Both facts argue for a boundary. If nearly every operation must nevertheless maintain a strongly consistent

invariant across them, that fact argues against the boundary. The decision is not whether coupling exists, but where the coupling is easiest to understand and manage.

A useful question is therefore:

DECISION · Where should the boundary go?

Draw a boundary so that the interactions and changes you expect are easy, while interactions and changes you do not want are difficult.

What should a boundary make local?

Decomposition is valuable when it makes consequential reasoning local. Different decompositions can produce the same externally correct behavior while distributing understanding, change, reuse, and failure differently. When comparing them, ask:

- **Understanding:** Can a part be understood without reconstructing the whole system?
- **Change:** Can an expected change remain local rather than spreading through unrelated parts?
- **Composition:** Can a part be reused or recombined without importing unnecessary context?
- **Failure:** Can a local failure remain local rather than propagating into unrelated work?

These objectives can conflict. Additional decomposition may improve change or failure containment while introducing more interfaces, dependencies, and concepts. A useful boundary therefore does not maximize modularity; it makes the properties that matter for this system sufficiently local.

Architecture bounds the consequences of uncertainty

Architecture bounds the consequences of uncertainty. Engineers can rarely establish every relevant property of every component with certainty. A boundary can make that residual uncertainty easier to accept by constraining what an unexpected behavior can affect. An isolated process may fail without corrupting another process. A permission boundary may prevent a component from exercising authority it does not need. A transaction may prevent a partially completed operation from leaving the system in an unacceptable state. Resource limits, timeouts, and similar controls can bound how far a failure propagates.

These controls do not establish that the component inside the boundary is correct. They change the consequence of being wrong about it. Architecture can therefore convert an unbounded validation problem into a bounded consequence problem. If engineers cannot economically establish everything a component might do, they may instead establish that whatever it does, important consequences remain constrained. This makes architecture part of the later validation argument: architecture determines not only how easily a system can be tested, but what failures that validation misses can affect. Validation develops this connection when considering whether the available evidence is sufficient to deliver a system.

Architecture also allocates coordination among people

Architectural boundaries do not affect only software. They also affect how engineering work can be divided. Once a system has coarse-grained parts with reasonably clear respon-

sibilities and interfaces, those parts can become units of ownership. A team may be able to change one part largely independently; a change that routinely crosses three boundaries may instead require coordination among three teams.

This relationship between software structure and organizational structure is captured by Conway's Law (Conway 1968): organizations tend to produce systems whose structures reflect their communication structures. The relationship matters in both directions. Existing organizational boundaries can push a system toward particular architectural boundaries, while an architectural choice can make some divisions of engineering work easier than others.

This connects architecture to the coordination problem introduced in Teamwork. A boundary that localizes software change may also localize the communication needed to make that change. A boundary that forces routine work across several parts may create a recurring coordination cost even when the resulting software is technically sound. Architecture allocates coupling in software and coordination among people.

Choosing among acceptable architectures

Satisfying the specification does not determine a unique architecture. Several organizations may preserve the required properties while differing in cost, performance, changeability, failure behavior, operational complexity, and the burden they place on engineers. Once clearly unacceptable alternatives have been removed, architecture becomes a comparative decision among organizations that could plausibly work.

A useful comparison proceeds in four steps.

1. **Separate constraints from objectives.** A constraint determines whether an alternative remains admissible. An objective distinguishes among alternatives that remain admissible. A required deployment boundary, regulatory obligation, or compatibility requirement may eliminate an architecture outright; latency, operating cost, or ease of change may instead provide reasons to prefer one acceptable alternative over another. Confusing the two wastes judgment on choices that have already been made.
2. **Eliminate what does not require judgment.** Some alternatives fail directly against known constraints or are dominated by another alternative on every consequential dimension. Remove them. Engineering judgment is scarce; it should be spent where plausible alternatives remain and their consequences differ in ways that matter.
3. **Compare genuine tradeoffs.** The remaining alternatives usually make different properties easier to preserve. A boundary that isolates failures may increase communication overhead. Centralizing state may simplify consistency while increasing coupling. Replication may improve availability while making ownership and synchronization harder to reason about. There is no general rule that resolves such choices. The engineer must decide which consequences matter most for this system and which compromises its obligations permit.
4. **Buy information when uncertainty could change the decision.** An architectural decision need not be made from assumptions that are cheap to test. If two alternatives differ principally in an uncertain property, ask what evidence would distinguish them and

whether obtaining it is worth the cost. A prototype, benchmark, dependency analysis, failure experiment, or small implementation may reveal enough to choose. The relevant question is not whether more information would be useful, but whether it could change the architectural decision.

This procedure narrows architectural judgment to the choices that actually require it. Constraints remove inadmissible alternatives; dominance removes unnecessary choices; tradeoffs expose the consequential differences among what remains; and targeted evidence reduces uncertainty when that uncertainty matters to the decision.

DECISION · Buy evidence?

If resolving an uncertainty could change a consequential architectural decision, evidence may be worth purchasing. Do not spend effort obtaining information that cannot change the decision. Do not refuse to obtain inexpensive information that could prevent an expensive mistake.

Patterns provide alternatives and expectations

Engineers rarely begin an architectural decision without prior experience. Recurring architectural problems have accumulated recurring solutions: layers, pipelines, repositories, event messaging, ports and adapters, and many others. These patterns are useful because they expand the set of plausible alternatives and carry experience about their likely consequences.

A pattern does not determine the answer. Strict layering can isolate change but make useful cross-layer interactions awkward. A shared repository can simplify consistency while coupling otherwise independent work through shared state. Event messaging can decouple producers and consumers while making ordering, observability, and failure handling harder. Ports and adapters can isolate infrastructure change while adding interfaces and indirection.

Patterns tell us what tends to happen, not what will happen in our context. They provide priors: plausible organizations and expectations about their consequences. If a consequential choice depends on whether those expectations hold here, stronger evidence may be worth buying.

Different questions require different architectural views

An architecture is not one box-and-arrow diagram. Different engineering questions require different reductions of the same system. The useful representation depends on the property engineers need to reason about.

Suppose we ask whether an expected change can remain inside one component. A dependency model can expose which other components know about it. Ask whether a request can satisfy a latency bound, and a flow model annotated with processing and communication costs may be more useful. Ask which failures can occur together, and a deployment model showing where components execute may expose the relevant relationships. The system has not changed. The engineering question has.

This is the same modeling discipline introduced in Specification. A model is a purposeful reduction: preserve the distinctions needed to answer the question and omit details that do not contribute to it. Philippe Kruchten's classic multiple-view account of architecture makes the same underlying point: no single representation serves every architectural concern (Kruchten 1995).

There is therefore no single artifact that is “the architecture diagram.” A diagram earns its place by supporting an engineering question. A drawing that cannot support a claim is decoration. Architecture makes some properties analyzable because its consequential structure can be represented at the level needed to reason about those properties.

Architectural claims require evidence

Choosing an architecture requires making claims about what its organization will accomplish. Different claims require different forms and strengths of evidence. A box-and-arrow diagram may establish that two responsibilities have been separated, but it does not by itself establish that a future change will remain local, that failures will be contained, or that a latency objective will be met. The representation and analysis must expose the property being claimed. A claim of failure containment, for example, requires evidence not merely that a boundary exists but that the boundary actually constrains the resources, authority, state, or interactions through which failure could propagate.

Consider an expected change. Suppose an architecture is intended to isolate changes to an external service behind one component. The relevant claim is not merely that the diagram contains a boundary. It is that a foreseeable change to the service can be absorbed without requiring coordinated changes elsewhere. A dependency or responsibility model can make that claim inspectable: identify what knowledge of the external service crosses the boundary, trace which parts depend on it, and ask whether the expected change can remain local. If the model shows that assumptions about the service have leaked across the system, the architectural claim is weak regardless of how clean the diagram appears.

Quantitative properties require different evidence. If an architectural alternative is intended to satisfy a latency or throughput obligation, engineers need a model that preserves the quantities governing that property. A dependency graph with estimated execution and communication costs may expose a critical path; a queueing or capacity model may expose a bottleneck; a prototype may provide measurements where estimates are too uncertain. The architecture need not predict the finished system perfectly. It must support a sufficiently credible claim that the chosen organization can satisfy the obligation, or reveal what remains uncertain enough to investigate.

The general discipline is the same: state the architectural claim, choose a representation that preserves the information needed to evaluate it, and obtain evidence strong enough for the consequence of being wrong. Architecture is not justified by producing models. Models are useful when they make the consequences of an organizational choice inspectable.

KEY-IDEA · Match the evidence to the claim

The more faithfully the model captures the property we care about, the less we have to bet. A reasoned expectation, structural argument, quantitative prediction, and measurement are all useful. The mistake is presenting one as stronger than it is.

When evidence becomes cheaper

Models, prototypes, workload generators, measurements, and competing implementations all cost engineering effort. Historically, that cost has rationed architectural evidence. Many questions were resolved primarily through experience and judgment because stronger evidence was too expensive to obtain. The decision to gather evidence is therefore itself an engineering decision.

As implementation and analysis become cheaper, the calculation changes. Software agents can help construct throwaway prototypes, generate workloads, instrument competing designs, analyze dependency graphs, or explore alternatives. This does not eliminate architectural tradeoffs. It can reduce the cost of learning about their consequences. When evidence becomes cheaper, more questions become worth investigating.

The engineering objective is therefore not to ask an agent to choose an architecture. Preference, obligation, and responsibility still belong to the engineers accountable for the system. The useful role of automation is to reduce how much of the architectural choice remains a bet.

Measurement for decision-making

The evidence considered so far supports choosing an architecture. Once the parts exist and run together, a second kind of evidence becomes available: whether the properties the organization was supposed to produce actually emerged from the composition.

Architectural decisions make predictions. A boundary is expected to contain failures. A critical path is expected to meet its latency budget. Replicated services are expected to supply sufficient availability. A decomposition is expected to let parts change independently. Each prediction implies an observation. If the architecture claims that a request completes within 200 milliseconds, distributed traces show how that budget is actually spent along the path and which hop consumes most of it. If a boundary is intended to isolate failure, deliberately injecting a fault behind it, or examining what an unplanned incident actually reached, shows whether the failure stayed inside. If two components are supposed to evolve independently, the version-control history can be examined for changes that touched both. A pair of components repeatedly modified in the same commit is not evolving independently, whatever the diagram shows.

The governing principle is to measure at the scope where the claimed property exists (“Validate at the scope of the property”). Measuring each component’s latency does not establish end-to-end latency when queuing and interaction contribute much of the result. Establishing that each service is individually available does not establish that the system is available, because dependencies can fail together. An architectural property belongs to

the composition, so the measurement must reach the composition. A dashboard of healthy component metrics is entirely compatible with an unhealthy system.

When observation repeatedly disagrees with an architectural prediction, the cheap response is to repair whichever component the measurement pointed at. That response is sometimes right and often insufficient. The prediction came from the organization: its responsibilities, boundaries, interactions, resource assumptions, and account of how system properties arise. Persistent disagreement is evidence about that organization, and the architecture is what should then be reconsidered.

A practical architecture decision procedure

The ideas in this chapter can be summarized as a sequence.

DECISION • Defend an architectural choice

1. Generate plausible alternatives. Use decomposition, prior experience, and architectural patterns to identify serious candidates. For consequential boundary choices, ask what should change, fail, scale, remain consistent, be secured, or be owned together—and what consequences should be prevented from crossing the boundary.
2. Eliminate specification violations. A candidate outside the acceptable realization space is not a tradeoff.
3. Eliminate dominated alternatives. If another candidate is at least as good everywhere and better somewhere, discard the dominated candidate.
4. Name the remaining tradeoff. State explicitly what one architecture gains and what it gives up.
5. Ask what is unknown. Identify uncertainties that could change the choice, and state the engineering question whose answer would reduce that uncertainty.
6. Buy evidence when worthwhile. Choose a model, prototype, experiment, or measurement that can answer the question when the information could change a consequential decision and is worth its cost.
7. Decide and accept responsibility. Choose among the remaining tradeoffs according to the obligations, priorities, and consequences that matter.

There need not be one correct architecture. Two engineering teams may make different judgments and both be defensible if each satisfies the specification, understands the tradeoffs, and supports its consequential claims with appropriate evidence.

Architecture is recursive

Architecture does not occupy one fixed level of a system hierarchy. A system may be decomposed into major components. One of those components may itself be too large to reason about directly. Engineers then establish an architecture for that component: its major internal parts, their responsibilities, interfaces, and rules of interaction. Architecture therefore recurs wherever engineers need to impose consequential organization on parts that are still too large to treat as directly understandable units.

What makes a decision architectural is not its position in a system hierarchy. It is that the decision establishes consequential organization that later engineering will take as given.

From architecture to design

Architecture deliberately does not decide everything. Once engineers have selected an organization, each part has an assigned responsibility and operates within boundaries, interfaces, and interaction rules. Those decisions create constraints and affordances for later work, but the parts are not yet implementations. A component responsible for classification may still need internal choices about data structures, algorithms, state ownership, concurrency, failure handling, caching, dependencies, and other mechanisms.

Architecture organizes responsibilities, boundaries, and interactions, thereby constraining the mechanisms available to each part. Design determines how each part realizes its responsibility within those constraints and affordances.

Detailed design may also expose that an inherited architectural decision cannot work as expected. A supposedly local choice may prove consequential to latency, consistency, security, failure isolation, or another system property. In that case, the right response may be to revisit the architecture rather than force a local workaround.

Specification asks: which realizations would we accept? Architecture asks: how should one acceptable realization be organized? Design asks: how should each part actually work?

Summary

Architecture chooses the consequential organization of one acceptable realization: its major parts, their responsibilities, the boundaries between them, and the rules by which they interact. Architectural decisions create constraints and affordances that later engineering must inherit. Boundaries do not eliminate coupling; they allocate it. They can also bound uncertainty: when correctness cannot be established completely, architectural controls can constrain the consequences of behavior that validation fails to anticipate. Ask what should change, fail, scale, remain consistent, be secured, or be owned together. The resulting boundaries affect both technical dependencies and the coordination required among people.

Architectural decisions should begin by eliminating candidates that violate the specification and alternatives that are dominated by others. What remains are genuine tradeoffs requiring engineering judgment. Patterns provide plausible alternatives and expectations about their consequences, not answers. Different engineering questions may require different architectural views; there is no single diagram that answers every question about a system. When uncertainty could change a consequential choice, engineers can buy information through models, prototypes, experiments, or measurement. The stronger the evidence, the less of the architectural decision remains a bet.

Architecture organizes responsibilities and interactions without determining every mechanism. Its decisions constrain the mechanisms available to Design. Design works within those constraints and can reveal when an architectural decision must be revisited.

READ FURTHER

- Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 3rd ed. Addison-Wesley, 2012. The standard treatment of how quality attributes drive architectural decisions.
- Fairbanks, George. *Just Enough Software Architecture: A Risk-Driven Approach*. Marshall & Brainerd, 2010. A risk-driven approach to deciding how much architectural modeling is worth doing.
- Kruchten, Philippe. “The 4+1 View Model of Architecture.” *IEEE Software* 12, no. 6 (1995): 42–50. The classic argument for using multiple architectural views to answer different engineering questions.
- Abowd, Gregory, Len Bass, Paul C. Clements, and Rick Kazman. *Scenario-Based Analysis of Software Architecture*. Technical report. Pittsburgh, PA, 1996. <https://doi.org/10.1109/52.542294>. Shows how to evaluate an architecture by turning quality-attribute concerns into concrete scenarios and tracing each one through the design.
- Kazman, Rick, Mark Klein, and Paul Clements. *ATAM: Method for Architecture Evaluation*. Technical Report CMU/SEI-2000-TR-004. Pittsburgh, PA, 2000. Stages that analysis into a repeatable organizational method, and names what an evaluation yields — risks, sensitivity points, and tradeoff points.
- Wan, Zhiyuan, Yun Zhang, Xin Xia, Yi Jiang, and David Lo. “Software Architecture in Practice: Challenges and Opportunities.” In “Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).” Special issue, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)* (New York, NY), 2023, 1457–69. <https://doi.org/10.1145/3611643.3616367>. An empirical account of how practitioners actually conduct architectural work, useful for seeing which parts of the methods literature reach practice.