THE SOFTWARE ENGINEERING

HANDBOOK

A Judgment and Decision-Making Approach

JAMES C. DAVIS

FIRST EDITION

Edition 1 — first published 2026-09-12 · last modified 2026-09-16

This work was supported by the U.S. National Science Foundation under grants #2541917, #2452533, and #2343596.

Preface

Once upon a time, engineering was taught via apprenticeship. A novice worked alongside experienced practitioners, first observing their decisions and then making increasingly consequential decisions under supervision. The apprentice learned the field's formal knowledge, but also something harder to write down: which questions to ask, which details matter, when a familiar solution applies, when it does not, and how to make a defensible choice when no answer is obviously correct. Modern engineering education cannot reproduce that arrangement at scale. We teach principles, methods, tools, and examples instead. These are valuable, but they can leave students knowing about engineering without yet knowing how an engineer thinks.

Welcome to your apprenticeship. This book attempts to teach software engineering through the judgments its practitioners must make. Why choose one development process rather than another? What should we promise to build? Which distinctions belong in a specification, and which choices should remain open? Where should an architectural boundary go? When is additional analysis worth its cost? There are methods that help answer these questions, but the methods are not the point. The point is learning to recognize the decision, identify the alternatives and consequences, and exercise engineering judgment.

I wrote this book because generative AI has made that distinction unusually important. As implementation becomes cheaper, producing code is less often the limiting step. The difficult questions remain: What should we build? What must be true of it? Which choices matter enough to constrain? What evidence is sufficient? When should we revisit an earlier decision? These have always been software-engineering questions. AI did not create them. It has simply made the scarcity of engineering judgment much easier to see. That is why this handbook begins by treating software engineering as judgment and decision-making rather than as a catalog of activities and terminology.

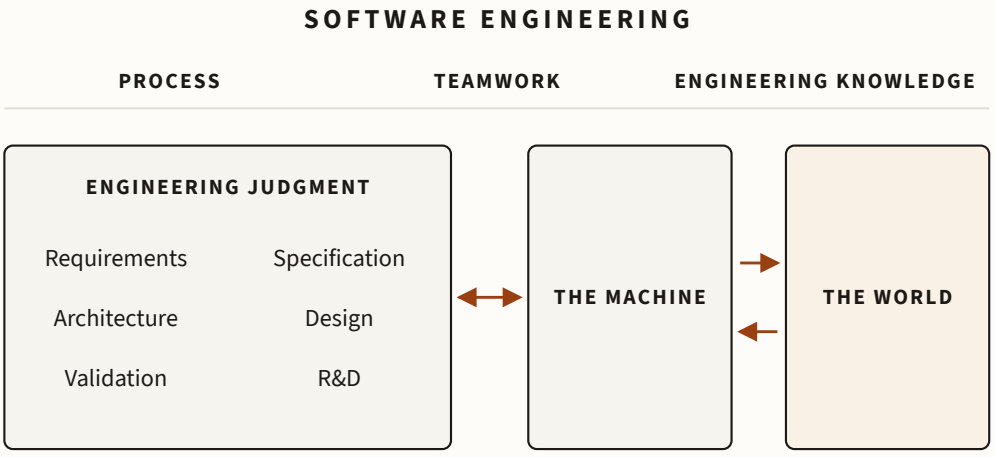
This handbook is also a practical companion to Model-Based Agentic Software Engineering (MAGE). MAGE argues that engineering with capable agents depends on making consequential knowledge explicit, important obligations enforceable, and recurring judgment durable. Applying those ideas requires knowing what is consequential in the first place. An engineer must decide what should be modeled, which obligations deserve enforcement, what can safely remain free, and when experience should become a rule that future work inherits. MAGE provides a theory for engineering with increasingly capable agents. This handbook develops the judgment needed to put that theory into practice.

So the chapters that follow are deliberately organized around decisions. Requirements engineering asks what we should promise. Specification asks what exactly that promise means. Architecture asks how competing obligations can coexist in one system. Design asks how the resulting parts should actually work. The same pattern continues throughout software engineering: identify the engineering problem, understand the choices, reason about their consequences, make a decision, and learn when reality tells you to reconsider it. No handbook can substitute for working beside excellent engineers. But it can make some of what they would teach you explicit. That is the apprenticeship I hope to offer here.

Introduction

This handbook covers software engineering, process, teamwork, engineering knowledge, requirements, specification, architecture, design, validation, and research and development. The chapters follow a linear sequence to show the shifting concerns of engineering work, as well as the judgments that are sustained throughout the process. Real engineering is less orderly. Requirements change as engineers learn, design can expose architectural problems, and validation can reopen earlier decisions.

In this handbook, I make one change to the usual sequence of engineering work. Research and development (R&D) typically occurs near the beginning, but we place it at the end. R&D asks engineers to decide where scarce effort should go when they do not yet know what is possible, whether an obstacle requires a genuinely new idea, and what evidence would establish an advance. These decisions are difficult to appreciate without some engineering experience. By the end of the handbook, the reader will have developed more of the judgment needed to understand them.



Contents

Preface i
Introduction ii
1. Software Engineering: The Engineering Discipline of Controlled Change 1
The properties of the software medium 1
We put change in software 2
Software gives engineers unusual leverage 2
Change creates risk 3
Technology moves the bottleneck 4
Summary 5
2. Process 7
The engineered medium shapes the process 7
Two ways to arrange the same work 8
A model for process choice 8
Certainty: what can we know before we build? 8
Changeability: how expensive is it to change our minds? 9
Decomposability: how much must we build at once? 10
Consequence of failure is a different question 10
The dimensions in practice 11
Engineers can move the dimensions 11
Closing 12
Summary 12
3. Teamwork 14
Individual capability is more than implementation 14
Team capability is an emergent property 15
Why teams do not scale linearly 15
Coordination can be engineered 16
Four dimensions of coordination 16
Coordination mechanisms 17
Engineering management allocates and develops capability 17
GenAI changes both sides of the equation 23
Closing 23
Summary 24
4. Engineering Knowledge 25
Engineering organizations must remember 25
Engineering knowledge comes from many places 26
Not everything should be remembered 28
Knowledge can be represented with different strengths 29
Representations are evidence, not reality 30
Records are not neutral 31
Engineering knowledge has a lifecycle 32

Consequential knowledge should change engineering . . . 32
Summary . . . 33

5. Requirements . . . 35
What would create value? . . . 36
What can we responsibly promise? . . . 38
The judgments remain coupled . . . 41
From requirements to specification . . . 41
Summary . . . 41

6. Specification . . . 43
Prioritizing specification effort . . . 43
Allocating responsibility between environment and machine . . . 44
From responsibility to acceptable realizations . . . 45
Not constrained is not the same as free . . . 46
Representing consequential properties . . . 46
Constrain, leave open, or learn more . . . 47
Software lets specifications learn . . . 48
From specification to architecture . . . 48
Summary . . . 49

7. Architecture . . . 51
From specification to one system . . . 51
Architecture creates possibilities and constraints . . . 52
Where should the boundaries go? . . . 52
Architecture also allocates coordination among people . . . 53
Choosing among acceptable architectures . . . 53
Patterns provide alternatives and expectations . . . 54
Different questions require different architectural views . . . 55
Architectural claims require evidence . . . 55
When evidence becomes cheaper . . . 56
A practical architecture decision procedure . . . 56
Architecture is recursive . . . 57
From architecture to design . . . 57
Summary . . . 58

8. Design . . . 59
Working within an architecture . . . 59
What Design inherits . . . 60
Choosing mechanisms . . . 60
Evidence for a Design decision . . . 62
Making Design reasoning inspectable . . . 62
Local decisions, system consequences . . . 63
When Design exposes a larger problem . . . 64
Cheap implementation changes the evidence . . . 65

Coda: Implementation in the GenAI Era 66
Summary 67
9. Validation 68
Acceptance is a professional judgment 69
Consequence changes the amount of evidence we should demand 71
Claims and evidence 72
Choosing among sources of evidence 74
The strength of evidence 74
Deciding when to stop 75
Evidence after delivery 76
Summary 77
10. Research and Development 79
What is worth pursuing? 80
Does it require a new idea? 80
How large is the advance? 82
What would establish the advance? 82
R&D is a bet on learning 83
Summary 83
Conclusion 85
Making an engineering decision 85

CHAPTER 1

Software Engineering: The Engineering Discipline of Controlled Change

Premise. *Software is an unusually changeable expressive medium. Engineering that medium means exploiting its capacity for change while keeping consequential change under control.*

> **DEFINITION · Software**
> Software is an expressive medium in which behavior is represented for execution by machines, including through both conventional instruction-based computation and machine reasoning.
>
> A sorting program represents behavior as instructions that a processor executes. A coding agent represents some behavior differently: its software may supply tools, context, constraints, and an objective while the agent reasons about which actions to take. Both use software to represent behavior that a machine can execute.

The properties of the software medium

Every engineering discipline works within the possibilities and limitations of its materials. Concrete can carry enormous compressive loads. Steel can provide strength with comparatively small structural members. Electronic circuits can process signals at speeds impossible for mechanical mechanisms. The properties of the medium shape what engineers can build and how they can build it. Software is a different kind of medium, with several unusual properties.

Software is changeable. Much of a software system can be altered without manufacturing a new physical artifact. A behavior represented in code can often be modified, tested, and replaced far more cheaply than a corresponding behavior fixed into hardware. This does not make every software change cheap. Interfaces accumulate users, data representations accumulate data, architectural choices constrain later work, and deployed systems acquire dependencies. But software gives engineers an extraordinary range of decisions that can remain revisable after initial construction.

Software is copyable. Once an implementation exists, producing another copy costs almost nothing. The engineering effort required to create the first instance need not be repeated for the millionth. A successful software design can therefore be replicated at a scale unusual among engineered artifacts.

Software is transferable. Software behavior is not permanently tied to the particular physical object on which it was first realized. The same program can often be transferred to another machine, deployed in another location, or incorporated into another engineered system. Appropriate interfaces and abstractions can separate a capability from much of the machinery underneath it.

Software is updateable. Engineering can continue after deployment. A system already in use can receive new behavior, new rules, repaired defects, or adaptations to circumstances

that did not exist when it was released. Deployment is therefore not necessarily the end of construction. It can be one point in a continuing engineering process.

These properties reinforce one another. Software can be copied widely and then updated. It can be transferred to new environments and adapted to them. A capability can remain in service while engineers continue to modify its realization. The result is a medium unusually suited to systems that must change.

We put change in software

Modern engineered systems contain some decisions that should remain stable and others that we expect to revisit. Where engineers have a choice, the latter increasingly migrate toward software.

Consider an automobile. Its physical dimensions cannot be changed remotely after it leaves the factory. But software can alter how the powertrain responds to driver input, how the battery is managed, what information is displayed, or how driver-assistance features behave. An industrial machine may have a physical structure intended to last for decades while its control logic changes with products and operating conditions. Communications infrastructure, aircraft, medical devices, scientific instruments, and energy systems similarly combine relatively stable physical machinery with software that supplies adaptable behavior.

Software also creates engineered systems with almost no visible physical counterpart: search engines, financial systems, social networks, operating systems, databases, business applications, and cloud services. Their purposes differ enormously, but they exploit the same basic property. Behavior represented in software can be revised as needs and circumstances change.

Software engineering is therefore not merely the engineering discipline for producing programs. It is increasingly the discipline through which other fields make parts of their own engineered systems adaptable. We put behavior in software partly because we expect that behavior to change.

But software does not change once. Systems remain in use while requirements, users, dependencies, platforms, regulations, and the engineers responsible for them change. Decisions that were sensible when a system was created may eventually become constraints, liabilities, or simply mysteries to the people maintaining it.

The engineering problem is therefore not merely to produce an acceptable artifact once. It is to keep an artifact acceptable as both the artifact and its environment change.

Software gives engineers unusual leverage

Software's copyability has another consequence: a single engineering decision can affect an extraordinary number of people. Most physical engineering is constrained by the number of artifacts that can be built or modified. A civil engineer can make a consequential decision about a bridge, but changing that bridge does not automatically change every other bridge in the world. Manufacturing can replicate a physical design at scale, but doing so still requires material, production, transportation, and installation for each additional artifact.

Software behaves differently. Once a change has been made, the cost of propagating it can be tiny compared with the cost of creating it. An engineer can modify one codebase and, through an update or deployment, alter software running on millions or even billions of devices. A change to a cloud service can affect its entire user population at once. The distance between an individual engineering decision and its effects on the world can therefore be remarkably short.

> **NOTE · Software engineering has unusual leverage**
> A software engineer need not personally construct a million artifacts to affect a million systems. Copyability and updateability allow one engineering decision to be reproduced across the entire deployed population.

This leverage is one reason software engineering carries substantial responsibility even when the artifact itself seems intangible. A small team, or sometimes a single engineer, can make decisions whose consequences rival those of enormous physical engineering efforts. The same mechanism that lets a beneficial repair reach every deployed instance can propagate a defect, vulnerability, or mistaken assumption just as efficiently.

Change creates risk

A system that is easy to change is also easy to change incorrectly. A new feature can violate an old requirement. A repair in one component can break an assumption made by another. An apparently harmless dependency can undermine a security boundary. A new data representation can make old data unreadable. A performance optimization can weaken reliability. A library update can alter behavior throughout a system. An engineer solving today's problem can unintentionally destroy a decision made years earlier for reasons the engineer no longer knows.

The more widely software is copied, the farther a defective change can propagate. The more interconnected software becomes, the more consequences a local change can have elsewhere. The more frequently software is updated, the more often the system is exposed to the possibility of unintended change.

This creates a tension at the center of software engineering. We want software to remain easy to change because changeability is one of its greatest advantages. But we also need important properties to survive those changes.

The problem is therefore one of engineering control.

> **DEFINITION · Engineering**
> Engineering is the discipline of exercising informed control over consequential systems and accepting responsibility for their outcomes. Engineers need not personally perform every act required to realize a system. They must retain sufficient command to understand and direct it, evaluate the evidence for its consequential properties, recognize when its assumptions fail, and intervene when necessary. They remain answerable for the consequential decisions made under their authority.
>
> — *Model-Based Agentic Software Engineering* (Davis 2026)

An engineer responsible for a payment service need not write every component or personally operate every server. The engineer must nevertheless be able to direct the system's development, judge whether the evidence justifies trusting it, recognize when assumptions about dependencies or operating conditions fail, and intervene when necessary.

Software's unusual capacity for change creates the need for engineering control. The medium makes consequential behavior unusually easy to alter and propagate; engineering requires that those consequences remain subject to informed control and responsibility.

DEFINITION · Software engineering

Software engineering is the engineering discipline of controlled change in software systems. It creates and evolves software while preserving sufficient control over the properties and consequences that matter.

Suppose the payment service must add a new payment method. Producing code that implements the method is only part of the problem. The change must also preserve existing security properties, remain compatible with clients and stored data, and provide enough evidence to justify deployment. Software makes the behavior changeable; engineering keeps the consequences of that change under control.

Controlled change does not mean preventing change. Nor does it require engineers to determine every detail themselves. Software engineering exploits the freedom the medium provides while maintaining sufficient control to determine what should change, what must remain true, what evidence is needed, and when an earlier decision or assumption must be reconsidered.

The chapters that follow examine different parts of that problem. Requirements engineering asks what outcomes matter enough to become engineering commitments. Specification determines what those commitments mean precisely enough to constrain acceptable realizations. Architecture organizes a system so that its obligations can coexist and so that expected changes do not require reasoning about everything at once. Design resolves the choices that remain within that organization. Validation asks what evidence justifies believing that the resulting system is acceptable. Maintenance and evolution continue the process as the system and its environment change. Software process, to which we turn next, asks how to organize all of this work.

Technology moves the bottleneck

What engineers must control remains, but the cost of exercising that control changes with technology. New technologies make capabilities that were once expensive comparatively abundant, and engineering reorganizes around the constraints that remain.

The steam engine made mechanical power available at a scale that human and animal labor could not provide. Factories greatly expanded physical production. Integrated circuits made computation extraordinarily inexpensive. These developments did not eliminate engineering. They changed its economics. As one capability became abundant, problems

elsewhere in the system became relatively more important: controlling the new capability, coordinating its use, and deciding what to do with it.

Software itself has repeatedly undergone the same process. Higher-level languages made programmers less responsible for machine instructions. Operating systems and libraries supplied capabilities that once had to be constructed locally. Open-source ecosystems made enormous bodies of implementation available for reuse. Cloud platforms made computing infrastructure available on demand. Each development changed what engineers had to do themselves, while creating new decisions about how the resulting capabilities should be selected, combined, controlled, and trusted.

Generative AI and coding agents continue that progression. Implementation that once required substantial human effort can increasingly be generated, transformed, tested, or explored by machine. The important question is therefore not whether an engineer personally typed the resulting code. Engineering has never been defined by personally performing every act required to realize a system. The question is whether engineers retain sufficient control to direct the work, judge its consequences, evaluate the evidence for the properties that matter, and intervene when necessary.

Greater implementation capacity therefore does not make software engineering obsolete. If implementation becomes cheaper, deciding what to build, controlling what may change, preserving what must remain true, and determining whether the result is acceptable become a larger fraction of the engineering problem. The bottleneck moves.

The tools will continue to change. So will the systems we build with them. The enduring problem is learning how to exploit a medium built for change without surrendering control of what those changes mean.

Summary

Software is an expressive medium in which behavior is represented for execution by machines. Its unusual changeability, copyability, transferability, and updateability make it valuable precisely because engineers can continue to alter behavior after a system has been created. Those same properties give individual changes unusual reach and allow mistakes, vulnerabilities, and mistaken assumptions to propagate just as readily as improvements.

Engineering requires informed control over consequential systems and responsibility for their outcomes. Software engineering applies that responsibility to a medium built for change. Its central problem is controlled change: exploiting software's adaptability while preserving sufficient control over the properties and consequences that matter.

Technologies such as generative AI can radically reduce the cost of producing implementations, but they do not remove this problem. As implementation becomes more abundant, judgment about what to build, what may change, what must remain true, and what evidence is sufficient becomes a larger share of the engineering work.

READ FURTHER

Davis, James C. *Model-Based Agentic Software Engineering*. 1st ed. 2026. Develops a theory for maintaining engineering control as capable agents perform more of the work.

Brooks, Frederick P., Jr. "No Silver Bullet—Essence and Accident in Software Engineering." *Computer* 20, no. 4 (1987): 10–19. The classic distinction between the essential difficulty of software and accidental difficulties of its implementation.

CHAPTER 2

Process

Premise. *The engineered medium affects the engineering process.*

Software engineering involves many kinds of work. Engineers determine what a system should do, design ways to accomplish it, implement those designs, evaluate the result, release it, repair it, and adapt it as circumstances change. A software process organizes these activities.

> **DEFINITION · Software process**
> A software process organizes engineering activities: deciding what to build, designing it, implementing it, validating it, and learning from the result. There is no universally correct ordering of these activities. Process is an engineering choice shaped by properties of the system and its environment.

The engineered medium shapes the process

Software Engineering described software as an unusually changeable, copyable, transferable, and updateable medium. Those properties do more than make software useful. They affect how it should be engineered.

Imagine constructing a bridge one span at a time, opening each new span to traffic, observing how drivers respond, and then using that feedback to decide how to design the next span. This sounds absurd, and the absurdity is informative.

Bridge engineering certainly includes requirements, design, construction, and validation. The problem is not that iteration or feedback are somehow foreign to civil engineering. The problem is that the physical medium changes the economics and usefulness of particular arrangements. Some commitments are extraordinarily expensive to reverse. A partially constructed bridge may provide little of the intended value. Learning from ordinary use may come much too late.

Software has different properties. Because it is changeable, copyable, transferable, and updateable, repeated cycles of construction, observation, and revision are unusually practical. We can build something, run it, learn from it, change it, and distribute the changed version — sometimes in minutes.

But software is not uniformly changeable. A line of internal implementation may be easy to replace. An API used by hundreds of clients is not. A data representation becomes harder to change after billions of records have been stored in it. An architectural choice becomes harder to reverse as other choices accumulate around it. Certification, external integrations, deployed dependencies, and organizational commitments can turn an initially flexible software decision into an expensive one.

The useful question is therefore not simply, “Is this software?” It is: *What properties of this particular engineering problem should determine how we organize the work?*

Two ways to arrange the same work

Consider two familiar process families. In a strongly plan-driven process, work is organized primarily by engineering activity. Requirements are established across much of the product before substantial design; design precedes much of the implementation; implementation precedes final validation and release. Progress between stages normally requires some evidence that the preceding work is sufficiently complete.

This does not mean that engineers are forbidden to go backward. Implementation may expose a design error. Validation may expose a misunderstood requirement. Rework occurs. The important feature is that the process makes relatively large commitments to one kind of engineering work before proceeding to the next.

An incremental process partitions the work differently. Instead of moving the whole product through requirements, design, implementation, and validation, engineers take a smaller product increment through those activities. Then they take another increment through them. The activities have not disappeared; their arrangement has changed.

> **NOTE · Activity versus increment**
> A strongly plan-driven process partitions work primarily by engineering activity. An incremental process partitions work more strongly by product increment.

These are not mutually exclusive. A project might establish substantial system requirements and architecture up front while designing, implementing, and validating individual capabilities incrementally.

This distinction is more useful than treating "Waterfall" and "Agile" as competing doctrines. The engineering question is why one partitioning of the work makes more sense than another.

A model for process choice

Plan-driven and incremental development give us alternatives, not an answer. Choosing between them requires reasoning about the engineering problem.

A useful model begins with three questions:

- What can we know before we build?
- How expensive is it to change what we build?
- Can a partial system be validated or deliver value?

These dimensions explain why no methodology is universally appropriate: different systems occupy different points in this space.

Certainty: what can we know before we build?

Planning has value when it lets us act on information we already possess. At one extreme, the problem is stable and well understood. Important requirements and constraints can be established before implementation. Engineers can reason about them, detect conflicts, and design around them before committing resources to a realization.

At the other extreme, some consequential information does not yet exist. Consider a new consumer application. Engineers can specify that it needs accounts, search, location

services, or a particular interaction. They may nevertheless be unable to predict which capabilities users will actually value, how users will behave, or which apparently minor feature will become important. Interviews and analysis can reduce this uncertainty, but some information may emerge only when people use a working system.

In such a case, implementation does more than produce the product. It produces information. That changes the value of iteration. Building an increment, observing it, and revising the system is useful when the observation changes what engineers know about what they should build next.

> **DECISION · Certainty**
> Ask: What consequential information is available before implementation, and what can only be learned by building?
>
> When important knowledge is available in advance, exploit it through planning. When important knowledge emerges through construction and use, create feedback loops that expose it early.

Uncertainty is not itself a failure. The process needs a way to resolve the uncertainty that matters.

Changeability: how expensive is it to change our minds?

Software Engineering identified changeability as a defining advantage of software. Here we encounter its process consequence: when changing a decision is cheap, committing to it early has less value.

Suppose an engineer can spend three days determining the perfect value for a configuration parameter or choose a reasonable value now and change it in twenty minutes tomorrow. Extensive up-front analysis may cost more than being wrong.

Now suppose the decision concerns a public interface that hundreds of systems will depend upon. Being wrong may require coordinated changes across organizations years later. More reasoning before commitment becomes much cheaper than reversal afterward.

The relevant quantity is therefore not simply the cost of making a decision. It is the cost of changing that decision after commitment.

Software decisions occupy a wide range. UI text, configuration, local algorithms, and isolated implementation details may be inexpensive to revise. Interfaces, persistent data representations, architectural boundaries, security protocols, external integrations, and certified behavior can become expensive or effectively irreversible.

This is also where copyability and updateability cut both ways. They can make a repair extraordinarily cheap to distribute: one corrected implementation can replace millions of deployed copies. But widespread deployment can also create enormous dependence on existing behavior. A decision that was cheap before release may become expensive once other systems, users, data, and organizations rely upon it.

> **DECISION · Changeability**
> Ask: How expensive will this decision be to reverse after we make it?

> Expensive commitments justify greater reasoning before commitment. Cheap and reversible choices permit more experimentation and correction.

Or, in intentionally simplified form: if it is expensive to change, try to get it right before committing; if it is cheap to change, do not spend more avoiding error than the error costs to correct.

This is one reason software can support processes that would be uneconomical for many physical systems. The medium gives engineers more opportunities to make provisional decisions. Good software engineering can create still more of them.

Decomposability: how much must we build at once?

The third question concerns partial systems. A half-completed bridge does not provide half the transportation value of a completed bridge. But many software systems can be divided into portions that are meaningful before the entire envisioned system exists, and there are actually two different thresholds at which a portion becomes meaningful.

First, a partial system may be validatable. It may not yet provide useful service, but engineers can build it, test it, measure it, or place it in a realistic environment. Doing so creates information about whether assumptions, interfaces, or designs are correct.

Second, a partial system may be useful. An online store might provide search and basic checkout before recommendations, wish lists, or every payment method exist. Users can receive value from the partial system while engineers continue developing the rest.

The distinction matters. Incremental construction can be worthwhile even when an increment cannot yet be released, because validation can reduce uncertainty. If the increment can also deliver useful value, the case for incremental development becomes stronger.

> **DECISION · Decomposability**
>
> Ask two questions: Can a partial system produce useful evidence? Can a partial system produce useful outcomes?
>
> The smaller the portion required for either, the more opportunities the process has for early learning. Independent usefulness additionally creates opportunities for incremental delivery.

A partial realization that can generate information or value is what makes incremental work pay off; software's updateability then lets later increments revise what has already been deployed.

Consequence of failure is a different question

One apparently obvious factor is absent from the three dimensions: how bad would failure be? That omission is deliberate.

Suppose a failure could cause serious harm. Does that tell us whether the requirements can be known before implementation? No. Does it tell us whether a design decision is cheap

to reverse? No. Does it tell us whether a partial system can be meaningfully validated? Again, no. A consequential system can lie anywhere on each of the three dimensions.

Consequence instead answers a different engineering question: how much evidence should we demand before accepting the system or a change to it? A low-consequence product might reasonably release an imperfect capability, observe its behavior, and repair problems afterward. That is not an acceptable strategy when a corresponding failure could injure someone. The latter system requires stronger evidence before the change is permitted to affect the world.

> **NOTE · Process and assurance**
> Certainty, changeability, and decomposability help determine how to organize the work. Consequence helps determine how much assurance to require before accepting the work.

Do not confuse a need for strong assurance with a requirement for one particular process model. This distinction will matter throughout the handbook. Highly iterative development and rigorous assurance are not opposites. The process determines how engineering work is organized; assurance determines what evidence must accompany consequential decisions.

The dimensions in practice

Certainty, changeability, and decomposability explain why superficially similar projects can rationally use different processes. Consider a consumer application whose success depends strongly on user preferences. Some important requirements will emerge through use. Many product decisions can be revised comparatively cheaply. Useful portions of the product can be released independently. These conditions favor small commitments, active feedback, and frequent iteration.

Now consider software embedded in a complex aircraft system. Many consequential obligations and interfaces can be established in advance. Hardware integration, deployed equipment, certification, and dependencies make some decisions expensive to reverse. Subsystems can be developed and validated independently, but useful operational capability ultimately depends on their successful integration. These conditions create stronger reasons for explicit up-front requirements and interfaces, carefully controlled commitments, and substantial verification before changes are accepted.

Both systems can iterate. Both can prototype. Both can plan. The conclusion is not that consumer applications use Agile while aircraft use Waterfall. The conclusion is that the properties of the engineering problem determine the value of different arrangements of work.

Engineers can move the dimensions

The three dimensions are not merely properties that engineers inherit. They are also properties that engineers can sometimes change.

If uncertainty is too high, engineers can invest in certainty. They can study the domain, interview stakeholders, construct prototypes, run experiments, model important relationships, or validate assumptions before making expensive commitments.

If decisions are too expensive to reverse, engineers can invest in changeability. Modularity can isolate decisions. Interfaces can prevent assumptions from spreading. Automated tests can reduce the cost of checking a revision. Continuous integration can make small changes easier to evaluate. Loose coupling can reduce the number of components affected when one component changes.

If a system must be nearly complete before anything can be learned or used, engineers can invest in decomposability. They can establish interfaces that permit components to be tested separately, identify vertical slices that provide end-to-end behavior, or restructure responsibilities so that capabilities can evolve independently.

These investments are themselves engineering decisions. A more modular architecture may be easier to change but harder to understand or operate. Independently deployable components require infrastructure. Prototypes consume time. Extensive requirements analysis can delay feedback. Automated validation costs money to create and maintain.

> **DECISION · Move the problem or adapt to it?**
>
> First ask where the project lies on certainty, changeability, and decomposability. Then ask whether it is worth moving any of them.
>
> Engineering can increase what we know before commitment, reduce the cost of changing our minds, and create smaller units from which we can learn or deliver value. But moving those dimensions has a cost. Invest when the resulting flexibility, information, or control is worth more than the investment required.

Closing

Software engineering therefore does more than exploit the changeability of its medium. Engineers deliberately change the economics of future change. Requirements work can reduce uncertainty before commitment. Architecture and design can make some changes cheaper and more independent. Validation infrastructure can make revisions cheaper to evaluate. A good process takes advantage of these properties rather than assuming them.

Choosing a process is therefore not simply a matter of asking, What kind of project do we have? It also asks, What kind of project is it worth engineering this into?

Summary

A software process is an arrangement of engineering work, not a universal sequence of phases. The appropriate arrangement depends on the problem and on the properties of the engineered medium. Three questions are especially useful: how much can we know before building, how expensive will our decisions be to change, and how much can we learn from or deliver through a partial system?

These dimensions explain why plan-driven and incremental arrangements can both be rational. Consequence of failure is a separate question: it determines how much assurance we require, not by itself how work should be partitioned. Engineers can also change the dimensions by investing in knowledge, changeability, and decomposability. Process choice

therefore asks both what kind of project we have and what kind of project it is worth engineering this into.

READ FURTHER

Sommerville, Ian. *Software Engineering*. 10th ed. Boston: Pearson, 2016. A clear survey of plan-driven and incremental process models.

Boehm, Barry, and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley, 2003. Frames the balance between up-front discipline and later adaptation as an engineering tradeoff.

Winters, Titus, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. Sebastopol, CA: O'Reilly Media, 2020. Treats software engineering as programming integrated over time and change.

CHAPTER 3

Teamwork

Premise. *A software team is a system for coordinating engineering capability.*

Software engineering requires more than assembling capable individuals. As projects grow, engineers must divide work, maintain shared context, make compatible decisions, integrate changes, and detect when their understanding has diverged. This creates an apparent puzzle: if interaction among engineers is costly, why use teams at all?

The answer is capability. One engineer has limited time, knowledge, and attention. Different engineers contribute different expertise; work can proceed in parallel; people can check one another's reasoning; and systems can outlive the individuals who originally built them. Large systems also exceed what one person can understand and maintain. Teams therefore buy capability, and coordination is part of the price.

> **DEFINITION · Coordination cost**
> Coordination cost is the communication, dependency, handoff, and integration work that team members incur to keep their contributions compatible. Adding engineers adds capability, but it also adds coordination cost — which is why engineering organizations do not scale linearly simply by adding people.

Teams also change as they work. Engineers develop expertise, knowledge becomes concentrated or distributed, people join and leave, and management decisions determine which capabilities receive attention and which are developed for the future. We will therefore consider not only how engineering capability is coordinated, but how an organization allocates, sustains, and develops that capability over time.

Individual capability is more than implementation

Effective engineers do more than implement software. Li, Ko, and Zhu's study of experienced software engineers found that implementation competence was important but insufficient to explain unusually effective engineers (Li et al. 2015). Four capabilities are particularly useful here.

- **Learn and adapt.** Effective engineers learn unfamiliar domains and revise their understanding when conditions change.
- **Exercise judgment.** They make defensible decisions under incomplete information, including recognizing which decisions can safely remain reversible.
- **Understand the system.** They reason about the product, organization, customers, tools, and consequences surrounding the code rather than only the implementation immediately before them.
- **Enable others.** They create shared context, communicate according to what others know, provide credible information, and help other engineers succeed.

The last capability changes the unit of analysis. A good engineer does not merely produce more personally. A good engineer increases the effectiveness of the system around them.

Team capability is an emergent property

A collection of effective engineers does not automatically form an effective team. Some useful properties belong to the team rather than to any individual member.

- **Shared context.** Team members need sufficiently compatible understandings of goals, priorities, system state, ownership, past decisions, and current changes. Shared context does not mean identical knowledge. On a large system that would be impossible. It means enough compatible knowledge for people to coordinate correctly.
- **Trust.** Information exchanged within the team must be usable. Engineers need to report status, uncertainty, risk, and mistakes credibly, and other engineers must be able to act on those reports. A team without trust may produce abundant communication while transmitting little reliable information.
- **Coordination.** Work needs ownership, dependencies must be managed, parallel work divided, and independently produced pieces integrated. Communication helps, but communication alone does not determine who decides or who is responsible.

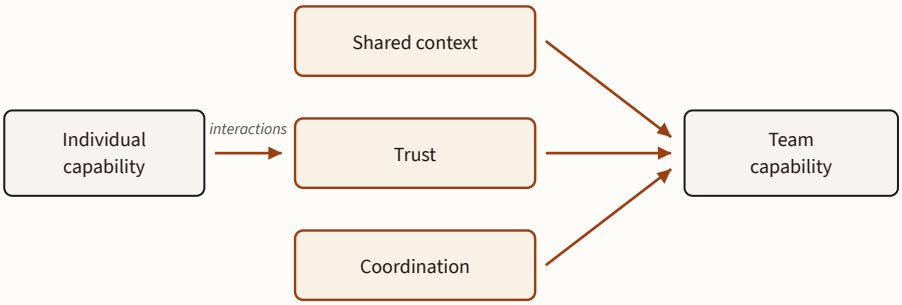


Figure 1. Individual capability becomes team capability only through interactions that build shared context, trust, and coordination. These team-level properties are what a team's engineering output actually flows through.

As Figure 1 suggests, these properties are produced by interaction and consumed by engineering work. A team that neglects them does not lose capability all at once; it discovers the loss later, as integration failures, duplicated work, and decisions that quietly contradict one another.

Why teams do not scale linearly

Adding an engineer adds capability, but it also creates potential interactions. Five engineers have ten possible pairs; fifteen have 105. This does not mean that every engineer communicates equally with every other engineer. In a well-designed organization they should not. The combinatorial growth instead illustrates what unconstrained interaction would cost (Brooks 1995).

Communication is only one source of coordination cost. More engineers also create more dependencies, onboarding, independently changing work, integration, and shared context to maintain.

KEY-IDEA · Scale capability faster than coordination cost

As an engineering organization grows, useful capacity should grow with it. Coordination cost will also grow, but an organization cannot scale if coordination grows as quickly as, or faster than, the capability being added. The engineering objective is therefore not to eliminate coordination. It is to make coordination scale more slowly than capability.

Coordination can be engineered

Coordination cost is not a fixed tax. Like the process dimensions of the previous chapter, it is a property engineers can deliberately move. Four families of strategy recur.

Strategy	Examples
Reduce dependencies	ownership, stable interfaces, modularity
Make knowledge reusable	documentation, conventions, discoverable expertise
Automate coordination	version control, automated tests, CI, tooling
Scale decision structures	review rules, ownership rules, team boundaries

Table 1. Strategies for engineering coordination.

What the strategies in Table 1 share is leverage: one decision should serve many future interactions.

Architecture can change who needs to coordinate with whom. Stable interfaces allow one group to work without continuously reconstructing the internal decisions of another. A software boundary can therefore also become an organizational boundary: knowledge and coordination can remain bounded on each side.

This is not an argument that organizational and software boundaries must always coincide. It is an argument that architecture affects the topology of coordination. A boundary that hides useful implementation details can reduce both technical coupling and the number of human interactions required to make a change. Architecture develops the technical side of this idea; here it is enough to notice that an architectural decision is also a decision about who must talk to whom.

Four dimensions of coordination

Coordination mechanisms can be compared along at least four dimensions.

- **Topology:** Who must coordinate with whom?
- **Frequency:** How often must coordination occur?
- **Cost:** How expensive is each interaction?
- **Reliability:** Does the necessary coordination actually occur correctly?

These dimensions turn familiar practices into engineering choices. A stable interface may reduce frequency. A team boundary may change topology. An asynchronous message may

reduce cost relative to a meeting. An automated check may improve reliability by ensuring that a repeated obligation is evaluated every time.

> **DECISION · Engineer the coordination**
> For a proposed coordination mechanism, ask: (1) Who must interact? (2) How often? (3) At what cost? (4) How reliably must the interaction occur? Then choose the simplest mechanism that satisfies those needs.

Coordination mechanisms

The familiar tools of engineering teams are instances of this model, not separate topics.

Communication. Start with what must be shared, with whom, how quickly, and whether it must persist. Then choose synchronous or asynchronous, and ephemeral or durable, communication.

Meetings. A meeting exchanges simultaneous attention for rapid context-sharing and interactive decisions. Its cost scales with attendance. Use one when synchronization is worth that cost.

Durable decisions. A decision that exists only in memory must eventually be reconstructed or lost. Recording it turns one act of coordination into something many future engineers can reuse.

Visible work state. Project management is not fundamentally ticket-tracker bureaucracy; it externalizes what exists, who owns it, its state, blockers, and what happens next.

Version control and automation. Git is a protocol for concurrent change. Branch isolates work; commit records change; a pull request exposes proposed integration; review creates a decision point; merge establishes shared state. Automated checks move some repeated coordination from human memory and judgment into machinery.

Engineering coordination tells us how multiple contributions can remain compatible, but it does not determine which contributions should be pursued or who should make them. A team still has to decide where scarce capability should go. Because those decisions also change the capabilities of the people involved, coordination leads naturally to a second problem: engineering management.

Engineering management allocates and develops capability

A software team rarely has enough capability to pursue every useful objective at once. Work must be prioritized, responsibilities assigned, scarce expertise directed toward particular problems, and plans revised as new information arrives. These decisions are part of engineering management.

Engineering management does more than allocate a fixed supply of labor. The people who perform engineering work learn from it: they acquire expertise, system knowledge, relationships, confidence, and judgment, while repeated assignments can also narrow their experience, overload them, or make particular capabilities dependent on particular people. Engineers eventually change roles or leave. An allocation decision must therefore

be evaluated against at least two states of the organization: what capability does this work require now, and what capability will this allocation leave behind?

> **DEFINITION · Engineering management**
>
> Engineering management allocates, sustains, and develops engineering capability over time. It is a class of engineering judgment, not merely a job title.
>
> Technical leads exercise it when deciding whether scarce expertise should solve a problem directly or help someone else learn to solve it; senior engineers exercise it when choosing what to delegate; project leaders exercise it when deciding whether additional parallelism is worth its coordination cost. In each case, the engineering question is how capability should be used given both immediate obligations and consequences for future work.

People are not substitutable

It is convenient for planning to describe a team as a number of engineers or a quantity of engineering capacity. That abstraction is sometimes useful, but it hides an important property of real teams: people are not substitutable.

Two engineers with the same title may have very different capabilities. One may understand a subsystem's history, another a customer's workflow, and another the failure modes of a particular technology. Engineers differ in technical expertise, judgment, relationships, interests, and experience. Replacing one engineer with another does not preserve all of these properties simply because both occupy the same position in an organizational chart.

This matters when allocating work. Suppose one engineer understands a critical database subsystem much better than anyone else. Assigning every database problem to that engineer may minimize the time required for each individual task. Over time, however, the organization becomes increasingly dependent on one person. Other engineers lose opportunities to develop the expertise. The expert becomes difficult to move to other work and may tire of being permanently assigned the same class of problems. A sequence of locally efficient assignments can therefore leave the organization less capable.

Because people are not substitutable, acquiring additional capability often requires coordination. A second engineer can learn the database subsystem, but doing so requires time with the expert, shared work, review, explanation, and perhaps temporarily slower execution. The organization gains a less concentrated distribution of capability by paying some of the coordination cost described earlier in this chapter. Coordination is often the price of acquiring non-substitutable capability.

The question *Who can perform this task most efficiently?* is consequently incomplete. When alternative assignments are plausible, compare their consequences: What does each assignment produce now? What capability does it develop? What scarce capability does it consume? What dependency does it create or reinforce? What happens if the person receiving the work later becomes unavailable? The fastest assignment may still be correct, especially when consequences are urgent. The point is that speed is one consequence of the decision rather than the decision rule itself.

Assigning work changes capability

An engineering assignment can produce two kinds of outcome: an outcome for the product and an outcome for the organization. Consider assigning a difficult but nonurgent change to an experienced engineer or to a less experienced engineer working with that person. The experienced engineer may complete the change faster alone. Working together consumes additional capability today, but the less experienced engineer may learn the subsystem, practice making the relevant decisions, and become able to handle similar work independently later. The second allocation has produced both the software change and additional engineering capability.

This makes mentorship, stretch assignments, rotation, and specialization alternative mechanisms for changing the future distribution of capability, not practices that are good in themselves. A stretch assignment may develop needed expertise, but it may be irresponsible when the cost of failure is high. Rotation can spread system knowledge, but it can also sacrifice valuable specialization. Pairing engineers can transfer expertise, but it consumes the attention of both. Conversely, repeatedly routing work to the current expert may be exactly right during an emergency even though it reinforces a long-term dependency. The management decision is therefore not *Should we mentor?* or *Should we rotate?* It is *What distribution of capability do we need, and which allocation of real work can move us toward it at acceptable cost and risk?*

Engineering capability is a resource with unusual properties. Using it consumes time and attention, yet doing work can also create more of it. The cost of an assignment should not always be evaluated solely by the effort required to produce its immediate artifact.

> **DECISION · Who should do the work?**
>
> An assignment changes both the product and the organization. When the choice is consequential, ask: What capability does the work require? Where does that capability exist now? How urgent are the immediate consequences? Which assignment develops useful future capability? Which assignment concentrates or reduces a dependency? What is the cost and risk of using the work as a learning opportunity? What future work do we expect the organization to perform?
>
> The best assignment need not maximize today's throughput. It should produce an acceptable product outcome and an acceptable future state of engineering capability.

Capability must be sustained

Capability must also be evaluated for sustainability. An organization that possesses a capability only because one particular engineer remains willing and able to provide it has a dependency on that person. That dependency may be perfectly reasonable: unusual expertise is valuable precisely because it is unusual. But engineers' interests and circumstances change; they seek different work and greater responsibility, become overloaded, change roles, leave organizations, and eventually retire. A consequential allocation decision should therefore consider not merely whether the required capability exists, but whether the organization can reasonably expect to retain access to it for as long as the capability will matter.

An engineer's interests are relevant evidence in an allocation decision. If repeated assignments conflict with the work an engineer wants to develop toward, the organization should not assume that the present allocation can continue indefinitely. Nor should it treat dissatisfaction merely as a personnel issue separate from engineering: if the allocation is creating both a retention risk and a single-person technical dependency, the two problems have the same cause. The relevant judgment is whether the short-term value of continuing the allocation justifies the capability risk it is accumulating.

A manager deciding who should perform recurring legacy-system work, for example, might discover a dangerous cycle. The expert receives the work because the expert is fastest. Nobody else develops the expertise because the expert receives the work. The expert cannot move toward work they would rather do because nobody else has the expertise. If the expert eventually leaves, the organization loses both the person and a capability it repeatedly chose not to develop elsewhere. Management must ask not only *Who should do this work?* but *What will repeatedly assigning this work to this person do to the person and to the organization?*

Organizations must plan for capability to leave

Turnover, promotion, reassignment, and retirement are normal conditions of an engineering organization. Management should therefore understand which important capabilities would disappear if particular people became unavailable.

Discovering a concentrated capability creates another decision: should the organization preserve it, distribute it, externalize what can be externalized, reacquire it when needed, or deliberately accept its loss? The answer depends on the consequence of losing the capability, the likelihood and urgency of future need, the cost of recreating it, and the cost of preserving it. Mentorship, overlapping ownership, hiring, training, and explicit representations are possible responses to that decision, not ends in themselves. Some deep tacit expertise may require substantial overlap between people to transfer; other capability may be cheaper to reacquire later; obsolete capability may deserve no investment at all.

The objective is not to make every person replaceable. That would contradict the very reason experienced engineers are valuable. Instead, managers must understand where the organization has deliberately accepted dependence on unusual individual capability and where it has merely accumulated that dependence accidentally.

This also reveals a limit of management through staffing alone. Moving knowledge among people can make capability more resilient, but consequential knowledge need not remain dependent on a person remembering it: documents, models, tests, and tools can carry engineering knowledge beyond the person who first held it (Engineering Knowledge takes up how consequential discoveries become engineering knowledge).

GenAI introduces substitutable capability

This chapter has argued that people are not substitutable, and that position stands: engineers differ in expertise, judgment, relationships, interests, and experience, and the work assigned to them changes the capabilities they develop. But generative AI is substitutable in a way that people are not. Model capability can increasingly be purchased on demand, replicated across many tasks, and replaced by another sufficiently capable model. An orga-

nization does not need to recruit, mentor, retain, and develop each new instance of model intelligence. Generative AI therefore introduces something unusual into engineering organizations: a comparatively commodity form of intelligence alongside human capability that remains individual, accumulated, and difficult to replace.

This difference changes the allocation model. Management traditionally allocates capability that is expensive to acquire, heterogeneous, and changed by the work assigned to it. Commodity intelligence can often be acquired when needed and replicated in parallel. As that capability becomes inexpensive and abundant, its complements can become the scarce resources: determining what should be built, supplying organizational and domain context, recognizing consequential tradeoffs, evaluating evidence, exercising authority, and accepting responsibility. The management question therefore changes from simply *Where should we allocate our available intelligence?* toward *Which intelligence should we buy, which capability must we develop, and which scarce complements constrain what either can accomplish?*

"Commodity" does not mean identical. Models differ substantially, and agents can accumulate task state and operate within rich engineering environments. The important distinction concerns acquisition and substitution. An organization cannot obtain another engineer with twenty years of accumulated experience by requesting another instance; it may obtain substantially more machine capability simply by purchasing more inference or adopting a better model. Engineering management must reason simultaneously about capability that can be acquired and substituted and capability that must be cultivated and retained.

The entry-rung problem

The contrast becomes particularly important when organizations develop junior engineers. Much of the traditional path toward engineering expertise has run through implementation work. Junior engineers implement changes, encounter unfamiliar systems, make mistakes, receive review, debug failures, and gradually acquire the context and judgment required for greater responsibility.

Generative AI can perform some of the work that historically occupied the beginning of that progression. This creates an entry-rung problem: if organizations automate work through which inexperienced engineers previously became experienced, how will they produce the senior engineers they will later need?

The entry-rung problem appears as a concrete allocation decision. Suppose an agent can perform a task for less money and time than a junior engineer. *Agent alone* may maximize immediate efficiency. *Junior engineer alone* may produce the artifact more slowly while developing experience. *Junior engineer with an agent* may accelerate both production and learning, or may merely hide the reasoning from the engineer. These alternatives cannot be compared solely by the cost of the artifact because they leave the organization in different states. The manager must decide how much the learning opportunity is worth, whether this task provides the right kind of learning, and whether its risk permits using it that way.

Delegation produces very different learning loops. An engineer who delegates a task, accepts the resulting artifact, and remains outside the reasoning may gain little capability

from the work. An engineer who forms a hypothesis, uses an agent to realize or test it, examines the resulting evidence, diagnoses failures, and revises the approach may experience a much faster cycle of engineering learning than implementation previously allowed. Delegation can remove engineers from the learning loop, or it can accelerate the loop.

Management should consequently evaluate automation not only by the labor it replaces. It should also consider what human capability its use creates or prevents the organization from creating. An allocation that minimizes today's cost may be poor management if it removes the mechanism through which tomorrow's scarce capability would have developed.

What human capability does management require?

The commodity-intelligence argument also applies to management itself. Rather than assume that management is an indivisible human capability, decompose the work and ask the same question we have asked elsewhere: what judgment is actually required here? Collecting status, summarizing information, tracking dependencies, maintaining schedules, and propagating routine decisions may require less scarce human capability as agents improve. Decisions about readiness for greater responsibility, conflicting legitimate interests, acceptable engineering risk, future expertise, or consequences for people's careers may require different information, judgment, authority, and accountability. The relevant boundary is not engineering work versus management work, or even machine work versus human work. It is which decisions can be delegated under what conditions, and which capabilities and authority each decision requires.

It would be tempting to declare these activities inherently human. We do not know that. Improvements in artificial intelligence may change which forms of judgment can be delegated, just as they are changing implementation work. The durable management question is therefore not *Which management jobs will survive?* It is *What consequential intelligence remains scarce, who or what can supply it, and who should have authority to act on it?*

Even if machine capability eventually crosses a particular judgment boundary, delegation does not follow automatically, because capability and authority are separate dimensions of the decision. An agent might become capable of producing an excellent staffing recommendation without an organization deciding that it should have authority to make the staffing decision. The decision may affect employees differently, privilege some objectives over others, or require someone to remain answerable for its consequences. For consequential management decisions, ask both *Can this actor make the decision well?* and *Should this actor be permitted to make it?*

Generative AI therefore puts pressure not simply on the number of managers, but on what management is for. Organizations must distinguish managerial activities that exist because information processing and coordination are expensive from those that exercise scarce judgment, develop people, allocate authority, reconcile competing interests, or establish responsibility. That boundary will itself change as machine capability changes.

Managing for future capability

Engineering management evaluates an allocation against both the work it accomplishes and the organizational state it leaves behind. The fastest assignment may concentrate expertise, eliminate a valuable learning opportunity, exhaust an engineer, or create an

unsustainable dependency; an assignment optimized entirely for future development may fail an urgent obligation today. Neither *maximize throughput* nor *develop people* is a sufficient rule. The decision is whether the immediate outcome and the resulting distribution of capability are acceptable given the organization's present obligations and plausible future work.

Generative AI makes this judgment harder because the relative scarcity of capabilities is changing quickly. Organizations may discover that capabilities they once spent heavily to develop can increasingly be purchased as commodities, while capabilities they previously took for granted become bottlenecks. They must therefore make allocation and development decisions without knowing precisely which human or machine capabilities future engineering will require. There is no fixed staffing formula that resolves that uncertainty; there is only the recurring engineering task of identifying what capability the organization needs, comparing ways to obtain or develop it, evaluating the consequences of those choices, and revising the allocation as evidence changes.

Engineering management is responsible not only for what the organization produces, but for the engineering capability the organization becomes.

GenAI changes both sides of the equation

GenAI can increase the surface that one engineer can meaningfully own. Faster comprehension, implementation, testing, and exploration may allow work that previously crossed several people to remain with one engineer. In that case some coordination does not merely become cheaper. It disappears.

But higher individual throughput can also increase the rate at which the team must absorb changes, review decisions, maintain shared context, and integrate work. If the surrounding coordination system does not improve, amplified individual capability can simply move the bottleneck outward. An amplifier amplifies a well-designed coordination system — or overwhelms a bad one.

> **NOTE · Not all interaction is overhead**
>
> Teams are also learning and social systems. Human interaction transfers tacit knowledge, supports mentorship, develops shared understanding, and creates relationships through which future coordination becomes easier. The objective is therefore not minimum human interaction. It is to eliminate interaction whose purpose is merely reconstructing work state while preserving interaction that creates engineering or organizational value.

Closing

The practical engineering question is therefore not simply *how do we make each engineer more productive?* It is also: *how should work and communication be structured so that individual capability becomes reliable team capability?* And, because every allocation changes future capability: *where should scarce capability go, and what should it leave the organization able to do?*

Summary

A software team is more than the sum of its individual engineers. Team capability emerges from how people combine expertise, share context, divide responsibility, and coordinate dependent work. Coordination mechanisms can reduce the cost of those interactions, but they do not eliminate the need to decide where capability should be used.

Engineering management adds a temporal dimension to that problem. Assigning work affects not only today's product but the expertise, motivation, dependencies, and capabilities the organization will possess tomorrow. Generative AI makes some forms of intelligence comparatively substitutable and abundant, increasing the importance of understanding which complementary capabilities remain scarce and which human capabilities the organization still needs to develop.

The result is a broader view of teamwork: engineers contribute capability; coordination combines it; management allocates, sustains, and develops it.

Some of that capability depends on knowledge that currently resides in particular people. Teams can spread knowledge through coordination and mentorship, but consequential engineering knowledge need not disappear when those people become unavailable. The next chapter takes up that problem directly: how should what an engineering organization learns persist, in what form, and with how much authority?

READ FURTHER

Li, Paul Luo, Amy J. Ko, and Jiamin Zhu. "What Makes a Great Software Engineer?" In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, 700–710. IEEE, 2015. An interview study of why experienced engineers rate judgment, system understanding, and enabling colleagues alongside implementation skill.

Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering*. Anniversary ed. Reading, MA: Addison-Wesley, 1995. The title essay makes the classic argument that adding people adds coordination work as well as capacity.

Davis, James C. *Model-Based Agentic Software Engineering*. 1st ed. 2026. Examines where scarce engineering effort moves when implementation is abundant, and how junior engineers will still develop judgment. Focus on Part 7, "The Profession," especially §§7.1 and 7.3.

CHAPTER 4

Engineering Knowledge

How should what engineers learn persist?

Premise. *Software engineering depends on knowledge that must outlive the people, systems, and circumstances in which it was discovered.*

Teamwork lets engineers combine capabilities that no individual possesses alone, but coordination in the moment is not enough. People forget and leave teams. Specialists know different parts of a system. Decisions made in one year constrain engineers working years later, while production failures can reveal facts nobody knew when those decisions were made. An engineering organization therefore needs memory: enough of what it learns must persist that future engineers can make decisions without repeatedly reconstructing the past.

This problem is particularly important because software is the engineering discipline of change. The artifact changes, and so does the environment in which it operates. A representation created yesterday may already describe an earlier version of the system. That does not make the representation useless. It means engineers must judge what it still tells them and how much weight it deserves in the decision they face now.

Knowledge management is therefore not the task of documenting everything. It is a problem of engineering judgment:

What should the organization remember, in what form, and with how much confidence?

> **DEFINITION · Engineering knowledge**
>
> Engineering knowledge is information about a system, its environment, or the decisions surrounding it that can support future engineering judgment.
>
> A developer discovers that a third-party service silently throttles requests above a particular rate. That observation becomes engineering knowledge when it can inform future decisions about batching, retries, capacity, or replacement of the dependency.

Engineering knowledge includes facts about the artifact, but it is not limited to them. Engineers also need to know why decisions were made, which assumptions supported them, what alternatives were rejected, what happened when a system encountered the world, and what remains uncertain. Knowledge management is the deliberate acquisition, representation, preservation, use, and revision of such knowledge when it may matter to future engineering decisions. Recording why a database was selected, retaining the measurements that supported the choice, and later reconsidering the decision when workload assumptions change are all knowledge-management activities.

Engineering organizations must remember

An engineering organization can know more than any person in it. One engineer may understand why a subsystem has an unusual interface, another which customer workflow motivated it, and another the production incident that caused a particular safeguard to be

added. Nobody needs to know everything. The organization needs to be able to recover the relevant knowledge when a decision requires it.

That capability is distributed across people, representations, tools, and the relationships among them (Hutchins 1995). An engineer may not personally know why a constraint exists, for example, yet still make a sound decision because the constraint is visible in a specification, embodied in an interface, or enforced by a check. What the organization can know and do therefore depends partly on how knowledge is distributed through its engineering environment.

Without some form of engineering memory, that distributed knowledge decays with the conversations and people that carried it. Engineers rediscover old constraints, repeat experiments, reopen settled arguments without knowing why earlier alternatives were rejected, or remove peculiar-looking mechanisms whose purpose has become invisible. The cost appears later as repeated work and decisions made without information the organization once possessed.

Preserving knowledge is not free either. Someone must record it, organize it, find it, interpret it, and sometimes maintain it as the system changes. A repository containing every conversation, intermediate thought, measurement, and obsolete plan might preserve enormous amounts of information while making useful knowledge harder to find.

The engineering problem is therefore selective:

What will future engineers regret having to rediscover?

Consequential knowledge, frequently reused knowledge, expensive discoveries, and decisions with long-lived effects are stronger candidates for preservation; ephemeral details whose rediscovery is cheap are weaker ones. The purpose of engineering memory is not to preserve everything. It is to reduce consequential rediscovery.

Engineering knowledge comes from many places

Engineering organizations learn continuously. Some knowledge comes from deliberate engineering activity. Some arrives from outside the engineering organization. Some is supplied by the behavior of the system itself.

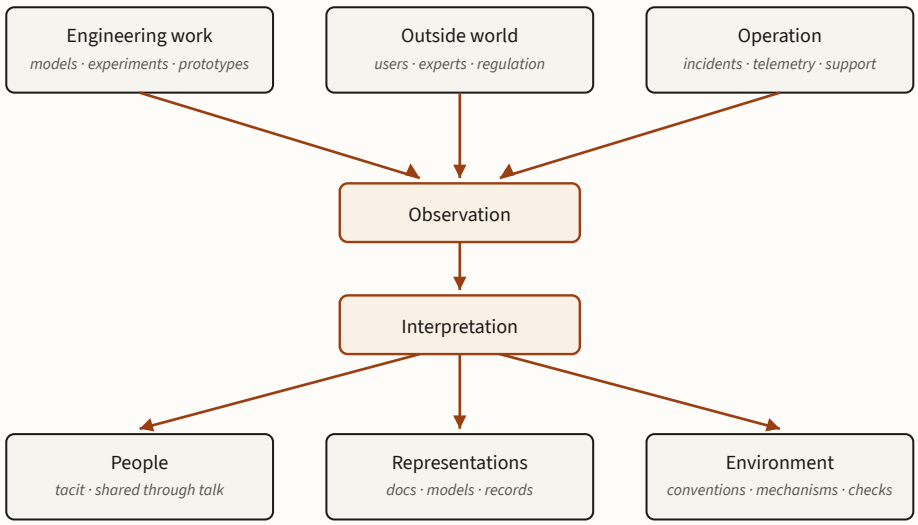


Figure 2. Engineering organizations learn from engineering work, the surrounding world, and operation of the system. Judgment determines what those observations mean and whether the resulting knowledge should remain with people, be represented explicitly, or be incorporated into the engineering environment.

Internal engineering work

Engineering produces information as well as artifacts. A prototype can reveal that a proposed mechanism is too slow; a dependency analysis can expose unexpected coupling; a design exercise can reveal an assumption that architecture left unresolved. Code review, experiments, measurements, and implementation itself similarly teach engineers about the system they are constructing.

Unsuccessful work can be valuable for the same reason. A rejected design may establish why an apparently attractive approach fails under an important constraint. Preserving the reasoning can prevent a future engineer from paying the same cost to rediscover the same limit.

Stakeholders and the external environment

Knowledge also enters from outside the engineering organization. Customers describe problems and workflows; users report confusing behavior; regulators impose obligations; domain experts explain facts about the world the software represents. Contracts establish commitments, suppliers change interfaces, standards evolve, and changing technologies alter what solutions are feasible.

These inputs do not arrive as perfectly formed engineering truths. A customer request may describe a proposed solution rather than the underlying need. Two users may want incompatible behavior. A regulation may require interpretation; a domain expert may be mistaken; an external dependency may behave differently from its documentation. Engineering therefore has to interpret external knowledge before deciding what consequences it should have for the system.

Operation, failures, and incidents

Once software operates in the world, its behavior becomes another source of knowledge. Measurements reveal workloads that estimates did not predict. Support cases reveal workflows designers did not anticipate. Security reports expose assumptions attackers can violate, while failures reveal combinations of circumstances that design-time reasoning missed.

Incidents are especially valuable because they provide evidence about assumptions the engineering organization actually made. A timeout may expose an unrealistic latency assumption. Data loss may reveal that two components disagreed about which held authoritative state. A deployment failure may expose a dependency missing from the architectural model. Repairing the immediate failure restores service; learning from it asks what the incident revealed about the organization's previous understanding.

An incident is therefore not only something to repair. It is an opportunity to learn what the organization misunderstood.

> **NOTE · Observation is not yet engineering knowledge**
>
> A customer complaint, benchmark result, incident, engineer's intuition, or regulatory interpretation is an input to engineering judgment. It does not automatically establish what is true or what engineers should do.
>
> Ask what the observation supports, what alternative explanations remain, how broadly the lesson generalizes, and whether it is consequential enough to affect future decisions. Capturing inputs without interpreting them can preserve information without producing useful engineering knowledge.

Not everything should be remembered

Once engineers recognize knowledge as valuable, preserving more of it can seem obviously beneficial. Past a point, the strategy defeats itself. Documents must be searched and interpreted; models compete for attention; old decisions must be distinguished from current ones; contradictory records require reconciliation. Material that nobody trusts becomes another body of information future engineers must search before they can act.

Capturing knowledge also creates maintenance costs. A representation intended to describe the current system may require revision as the system evolves. The more representations an organization creates, the more relationships it may need to maintain among them.

Whether knowledge deserves durable representation therefore depends on its expected future value.

> **DECISION · Should this knowledge persist?**
>
> Ask:
>
> - **What happens if it is forgotten?** Forgetting the command used for a one-time migration may be harmless. Forgetting why a safety check exists may not be.

- **How likely are engineers to need it again?** Recurring decisions justify more durable treatment than genuine exceptions.
- **How expensive would rediscovery be?** A two-minute experiment may be cheaper to repeat than maintain. Months of investigation are different.
- **How many future decisions could it affect?** A local implementation detail may affect one engineer; a system-wide constraint may affect hundreds of changes.
- **What will preservation cost?** Capturing, finding, interpreting, revising, and reconciling knowledge also consume engineering capacity.

The question is therefore not simply whether something is worth knowing. Engineers must compare the expected value of future use against the cost of making the knowledge persist.

Knowledge can be represented with different strengths

Knowledge can persist through people, communication, representations, or the engineering environment. These forms make different tradeoffs. Tacit knowledge held by an experienced engineer can be rich and adaptable, but access depends on that person. Conversation spreads knowledge among people, but the resulting shared context can disappear as membership changes. Explicit representations such as prose, diagrams, models, requirements, measurements, runbooks, and decision records survive the original conversation, although every representation selects some facts and omits others.

Recurring knowledge can also be incorporated into the engineering environment. A recurring decision may become a convention or shared library. A recurring mistake may become a static check. An obligation may become an automated test or control. Future engineers no longer need to retrieve the lesson from another person's memory before benefiting from it; they encounter the accumulated knowledge in the environment in which they work.

These forms do not constitute a maturity ladder. A conversation may be exactly right for an ephemeral issue, while prose may preserve rationale that would be meaningless as an automated rule. A model can expose relationships that prose obscures. Automation is useful only when the underlying knowledge is stable and precise enough to support it. The appropriate representation depends on how the knowledge is expected to be used.

KEY-IDEA · Knowledge can move

A lesson may begin in one engineer's experience, become shared through conversation, be represented in a document or model, and eventually become a convention or automated control.

Move knowledge when the expected benefit of making it more durable, discoverable, precise, or enforceable exceeds the cost.

NOTE · Knowledge graphs and engineering agents

Documents are not the only way to represent engineering knowledge. A knowledge graph represents entities and relationships explicitly: a component depends on a service;

a requirement constrains an interface; a test provides evidence for a claim; a decision supersedes an earlier decision. Graph structure makes relationships available for traversal and computation rather than leaving them implicit across prose and files.

This possibility has become more consequential with engineering agents. A human engineer can often reconstruct relationships by reading several documents and using experience to connect them. An agent likewise benefits when important relationships are represented explicitly rather than requiring them to be repeatedly inferred from text. Knowledge graphs can therefore serve as part of an engineering environment through which both humans and agents retrieve context, follow dependencies, and identify relevant knowledge.

A graph does not solve the knowledge-management problem by itself. Its nodes and relationships can still be incomplete, stale, ambiguous, or wrong. The same questions developed in this chapter still apply: Where did this knowledge come from? What claim does it support? How current is it? And how much weight should an engineer or agent place on it?

Representations are evidence, not reality

Every representation leaves something out. A dependency graph omits details irrelevant to dependencies; an architectural diagram selects relationships at a particular level of abstraction; meeting notes preserve a summary rather than the meeting itself. This selectivity is what makes representations useful. A representation that reproduced reality in every detail would offer little reduction in the reasoning required to understand it.

Software adds another complication because the represented system keeps changing. Suppose an architectural diagram accurately describes a system when it is created. Engineers later add dependencies, split components, introduce caches, replace services, and create exceptions. The diagram rarely becomes false at one identifiable moment. Its relationship to the current system instead becomes increasingly approximate.

Approximation does not erase the diagram's value. It may remain excellent evidence of the original architecture, explain why interfaces have their present shapes, preserve terminology still used by the team, or reveal the organizing idea from which the current system evolved. None of those uses establishes that it still describes every current dependency.

A representation can remain useful as it drifts from current reality, but the weight engineers place on it should reflect its fidelity to the claim being considered now.

This is a routine problem rather than an exceptional failure of documentation. Software is built to change, so engineers routinely make present decisions using knowledge produced under earlier conditions. The relevant question is not simply whether a representation is "up to date." It is what claim the engineer is trying to support, how the representation was produced, and what relevant parts of reality have changed since then.

Provenance helps answer that question:

- Who produced this representation?
- When was it produced?

- What observations or evidence supported it?
- What purpose was it intended to serve?
- What claim did it actually make?
- What relevant conditions have changed since then?
- What incentives or interests may have shaped what was recorded?
- What claim are we trying to support with it now?

These questions do not produce a mechanical confidence score. They establish the context in which an engineer can judge how much evidentiary weight a representation deserves for the particular decision at hand.

EXAMPLE · One document, different evidence

Consider an architectural diagram created in 2023 showing a payment service as three components: an API, a payment processor, and a database. By 2026 the system has added a queue, a fraud service, and direct reporting access to the database.

The diagram remains strong evidence of the decomposition the architects intended in 2023 and may explain why several interfaces still look the way they do. It is weaker evidence about today's dependency structure and provides almost no evidence that the workload assumptions behind the original architecture remain valid. The document did not suddenly become useless when the first change occurred. Different claims simply require different evidence.

Age does not determine usefulness. The claim being made determines the evidence required.

Records are not neutral

Representations are also produced by people inside organizations. Those people have incomplete information, interpret events differently, possess different authority, and may face incentives that affect what gets measured, estimated, emphasized, or recorded. Engineering records therefore carry the circumstances of their production along with their technical content.

WARNING · Records are not neutral

Engineering records are created by people inside organizations. They may be incomplete, mistaken, selective, optimistic, politically convenient, or occasionally deliberately false. Estimates can reflect incentives as well as forecasts. Meeting notes reflect what the recorder noticed, understood, and chose to preserve. Decision records may describe a cleaner rationale than the one that actually produced the decision. Representation gives knowledge persistence. It does not give it truth.

Records can also acquire audiences their authors never anticipated. Internal messages, estimates, meeting notes, design discussions, and incident records may later be read by executives, auditors, regulators, courts, journalists, or legislators. This is not an argument for avoiding records or disguising uncomfortable facts. Engineers should distinguish observation from interpretation, preserve material uncertainty and disagreement, attribute

decisions accurately, and write records that can fairly explain what was known and decided at the time.

Who records a decision influences what the organization later remembers about it. Treat that influence as an engineering responsibility.

The appropriate response is neither cynicism nor blind trust. It is the same response engineers apply to other evidence: understand its provenance, understand what claim it supports, and calibrate the weight placed on it.

Engineering knowledge has a lifecycle

Representing knowledge is not the end of knowledge management. A useful simplified lifecycle is:

Observe → Interpret → Represent → Use → Reassess

A production incident reveals that an upstream service sometimes takes thirty seconds to respond. Engineers first observe the timeout pattern, then interpret it as evidence that an assumed latency bound is unsafe. They represent the conclusion in a dependency record and establish a timeout policy. Future engineers use that knowledge when designing calls to the service. If the dependency or its service-level guarantees later change, the organization must reassess whether the old conclusion still deserves authority.

The lifecycle matters because engineering knowledge is acquired under particular circumstances and later used under circumstances that may differ. Measurements describe workloads that may change; design decisions depend on assumptions that may disappear; customer needs evolve; new evidence may contradict earlier conclusions.

Reassessment does not always mean revision. Historical knowledge can remain useful precisely because it describes the past. An old decision record may no longer govern the current architecture while remaining the best explanation of why the architecture evolved in a particular direction.

It is therefore useful to distinguish historical value from current authority. Superseding an engineering record need not mean deleting it. Future engineers may need both the current decision and the history that explains how the organization reached it. Conversely, preserving an old record without making its changed status visible can be dangerous. An engineer may reasonably treat an apparently current runbook, requirement, or architectural rule as authoritative when the organization no longer does.

Good engineering memory requires both remembering and knowing what no longer deserves authority.

Consequential knowledge should change engineering

The strongest evidence that knowledge matters is often that engineers keep needing it. Repeated rediscovery is evidence about the representation itself. A fact that engineers continually reconstruct may need to become easier to find. An explanation repeated to every new team member may deserve durable representation. Repeated mistakes can indicate that documentation is too weak a form for the knowledge being preserved. When

incidents repeatedly expose the same assumption, the organization may need to change the engineering environment rather than remind engineers of the lesson again.

Consequential knowledge can therefore migrate into more durable engineering structure, and the appropriate destination depends on what future decision the knowledge should constrain. Some knowledge becomes a requirement because it describes what stakeholders need from the system. Some becomes specification because engineers must make acceptable behavior precise. Some becomes architecture because later engineering should inherit a consequential organizational decision. Some becomes a design convention or shared mechanism because engineers should not repeatedly solve the same local problem. Some becomes validation evidence or a continuing check because the organization needs grounds for believing that an important claim remains true. Choosing among those destinations requires understanding what kind of engineering decision the knowledge should support; the remaining chapters develop those decisions in turn.

Learning should sometimes change how future engineering is performed. An organization that repeatedly experiences the same surprise without changing what it remembers, represents, or does has collected experience without accumulating much engineering knowledge.

Summary

Engineering organizations learn from many sources. Engineers discover facts through design, implementation, modeling, experiments, and measurement. Stakeholders and the external environment supply needs, constraints, claims, and changes. Deployed systems supply observations through normal operation, failures, and incidents. None of these inputs interprets itself.

Knowledge management determines which lessons deserve to persist and how they should be represented. Tacit understanding, communication, documents, models, conventions, mechanisms, and automated controls provide different combinations of richness, durability, discoverability, precision, and authority. More durable representation is valuable only when its future benefit justifies its cost.

Representations are evidence rather than reality. They abstract, they age, and the systems they describe evolve. Their usefulness need not disappear as they drift from current reality, but the weight placed on them should reflect their provenance and their fidelity to the particular claim under consideration. Representation also does not guarantee truth: organizational incentives, incomplete information, interpretation, and authority affect what gets recorded.

Engineering knowledge therefore has a lifecycle. Engineers observe, interpret, represent, use, and reassess what they know. Consequential recurring knowledge should sometimes become part of the structure within which future engineering occurs.

Engineering knowledge lets yesterday's learning reduce today's rediscovery. The next question is what the system ought to accomplish in the first place.

READ FURTHER

Hutchins, Edwin. "How a Cockpit Remembers Its Speeds." *Cognitive Science* 19, no. 3 (1995): 265–288. The canonical demonstration of distributed cognition across people, procedures, instruments, and representations.

Rus, Ioana, and Mikael Lindvall. "Knowledge Management in Software Engineering." *IEEE Software* 19, no. 3 (2002): 26–38. Introduces knowledge management as a software-engineering problem of creating, sharing, and preserving organizational knowledge.

Hansen, Morten T., Nitin Nohria, and Thomas Tierney. "What's Your Strategy for Managing Knowledge?" *Harvard Business Review* 77, no. 2 (1999): 106–116. Contrasts codifying knowledge with connecting the people who possess it.

Hogan, Aidan, et al. "Knowledge Graphs." *ACM Computing Surveys* 54, no. 4 (2021), Article 71. A comprehensive survey of knowledge graphs as explicit, processable representations of entities and relationships. Read the introductory material and the sections on representation rather than the entire survey.

CHAPTER 5

Requirements

Premise. *Requirements engineering turns uncertainty about what matters into engineering commitments.*

Software systems are built for purposes in the world. Before engineers can decide how a system should work, they must decide what would create value and what they can responsibly promise to deliver. Neither decision is simply a matter of asking people what they want. A user usually describes a solution they can imagine, using the vocabulary of systems they already know. Their request may reveal something important without literally describing what should be built. Different stakeholders may want different things. Important needs may remain implicit until something violates them. Obligations may also come from laws, contracts, existing systems, operations, standards, or other parts of the surrounding domain.

The requirements engineer is therefore not a stenographer. Requirements engineering requires interpretation, and engineers are responsible for the interpretations and commitments they make. Discovery and commitment inform one another: what appears valuable affects what engineers consider promising, while feasibility, cost, risk, and new evidence can change what appears valuable.

> **DEFINITION · Requirements engineering**
> Requirements engineering connects the purposes a system serves to decisions about what the system should do and what constraints it must satisfy. It discovers candidate obligations, evaluates them, and determines which engineers can responsibly accept as commitments.

That definition compresses the two coupled judgments this chapter unpacks: *What would create value?* and *What can we responsibly promise?* Discovery answers the first and produces candidate obligations. Commitment answers the second: a candidate can be accepted, revised, investigated further, or rejected. Neither judgment is final while the other is changing. What engineers learn while deciding—an estimate, a dependency, a stakeholder conflict—can change what appears valuable. What engineers learn about value can likewise change which commitments are worth investigating.

> **KEY-IDEA · The standard requirements work must meet**
> Requirements engineering does not eliminate uncertainty before commitment. It reduces uncertainty enough that a commitment can be defended.

That standard governs everything that follows. Discovery buys information a defensible commitment needs; commitment judges whether enough has been bought.

What would create value?

The first problem is discovering what would create value. A requirement is meaningful relative to some purpose. Requirements connect what a system is for to what engineers promise the system will do. Locally, the relationship appears simple: Purpose → Requirements → System.

In practice, purpose does not arrive as a complete, coherent description. Different stakeholders see different parts of the problem. A manager may understand organizational goals and constraints while an operator understands daily work, exceptions, and workarounds. A customer may understand the outcome they want without knowing which system behavior would produce it. Stakeholders may disagree about priorities or even about the problem being solved. Requirements engineering exists partly because purpose must be discovered, interpreted, reconciled, and converted into obligations that engineering can act upon.

Users and customers are important sources of candidates, but they are not the only ones. Candidate requirements can come from users, customers, operations, existing systems, technology, laws and standards, contracts, competitors, and the surrounding domain. Each source sees different obligations. An operations engineer may need logging, rate limiting, observability, or a way to disable a malfunctioning feature even though no end user would ask for them. An existing system may impose a data format that other systems have come to depend upon. A contract may contain an obligation that nobody currently working on the software negotiated. Laws and standards can impose obligations on an organization that ultimately require particular software behavior.

These sources can also differ in authority and precision. A regulation, for example, may impose a binding requirement to provide security appropriate to some risk without prescribing exactly what technical measures satisfy it. The requirement is authoritative, but engineering work remains. Authority and precision are different properties: a requirement can be binding while leaving important engineering decisions open.

Requirements are commonly described in several overlapping categories — useful as ways of checking that the candidate space has been covered, not as the conceptual model of requirements engineering. Functional requirements describe behavior the system should provide — for example: email an administrator when a project has no moderator. Quality requirements describe how well the system should behave or limits its behavior should satisfy. Some are already close to measurable: 95% of searches return within one second. Others express a meaningful obligation while leaving substantial interpretation unresolved: the system follows security best practices. Domain and external requirements arise from the environment in which the system exists — for example, a regulation may require that withdrawing consent be as easy as giving it.

These examples differ greatly in precision. Being a requirement does not imply being sufficiently precise to implement or validate. “95% of searches return within one second” already supplies a measurable threshold. “The system follows security best practices” leaves important questions unanswered: which practices, against which threats, and what evidence would establish that the obligation has been satisfied? Resolving those questions is part of the path from requirements to specification.

Candidates also rarely stand alone. Understanding one may require knowing who needs it, what goal it serves, why it exists, which scenarios exercise it, how important it is, which interfaces it affects, and how anyone could determine whether it has been satisfied. Recording only a requirement sentence can discard information needed to interpret the obligation later.

Learning what would create value

Discovery turns the question of what would create value into candidate obligations, and engineers choose discovery techniques according to what they do not yet know.

- **Ask.** Use when people hold the relevant knowledge and can articulate it: interviews, surveys, workshops, and similar techniques expose goals, preferences, constraints, and disagreements that stakeholders can describe.
- **Observe.** Use when important knowledge is tacit or embedded in actual work: observation of users, operational data, existing systems, incidents, and workarounds can expose needs that nobody thought to state.
- **Compare.** Use when competitors, substitutes, or prior systems reveal alternatives that have already been explored: existing solutions can expose both expectations and opportunities.
- **Build.** Use when people cannot meaningfully evaluate the possibility until something concrete exists: a prototype, mock-up, experiment, or partial implementation can turn an imagined possibility into something stakeholders can inspect and criticize.

Each technique buys information, and the choice among them is itself an engineering judgment. The guiding questions are where the missing knowledge lives and what resolving it costs. If the people involved hold the answer and can state it, asking is cheap and direct. If the answer is tacit, embedded in habits, workarounds, and exceptions, asking will return an idealized account, and observing is worth its higher cost. If someone else has already paid to explore the alternative, comparing recovers their investment. If nobody can evaluate the idea until it exists, building is the only technique that produces the missing evidence, and its cost should be sized to the uncertainty it resolves.

Software's changeability makes building unusually useful as a way to learn. When a prototype is expensive, engineers have strong incentives to describe an idea carefully before realizing it. The first substantial reaction from stakeholders may arrive only after considerable work. When a prototype is cheap, the sequence can change: Idea → Build enough to react to → Reaction → Revise → Reaction → Revise. People are often better at criticizing an artifact than specifying one in advance. A partial realization can therefore expose needs, preferences, assumptions, and disagreements that remained invisible in discussion.

This does not make every prototype useful. A prototype should exercise the uncertainty engineers are trying to resolve. Something that looks finished can also acquire commitments nobody intended: stakeholders may interpret a provisional interface, behavior, or technical choice as a decision.

> **NOTE · Prototype ≠ specification**

> A prototype can teach us the requirements. It is not the specification, and it is not necessarily the product.

The engineering question is not whether to think or to build. It is which action buys the information needed for the next consequential decision.

Requirements engineering is therefore iterative rather than a single pass. Discovery produces candidate obligations, while attempts to evaluate those candidates produce new information about value, feasibility, cost, dependencies, and risk. Software processes differ in when requirements are established, how much is established at once, and how often earlier requirements decisions are revisited.

What can we responsibly promise?

Discovering that something would be valuable does not automatically make it a requirement. Discovery produces candidates; commitment decides which of those candidates the engineering effort turns into promises, and that decision is the chapter's second judgment.

> **DEFINITION · Candidate requirement and requirement**
> A candidate requirement is an obligation under consideration. A requirement is an obligation the engineering effort has accepted.

The distinction matters because accepting a requirement creates responsibility, and the decision that separates the two has more resolutions than yes or no. A candidate can be **accepted** as an obligation, **revised** to keep most of its value at lower cost or risk, held while the engineers **learn more**, or **rejected**. All four outcomes are legitimate resolutions; only accept yields a requirement (Figure 3).

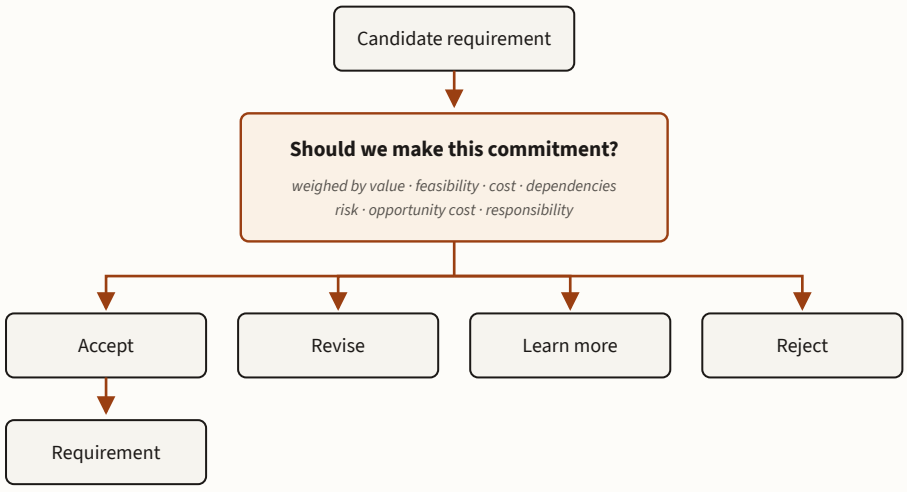


Figure 3. The commitment decision. A candidate requirement is weighed and resolves to one of four peer outcomes; only accept yields a requirement.

Several kinds of information weigh the decision, though none provides a formula for it:

- **Value.** What would the candidate contribute to the purposes the system serves, and for whom?
- **Feasibility.** Can it be satisfied under the available constraints? Engineers should not accept obligations they cannot reasonably satisfy.
- **Cost.** What would delivering and keeping it take? An obligation worth $100,000 may not be worth $10 million.
- **Dependencies.** What else must exist or hold for it to work? Apparently small obligations can require substantial surrounding work.
- **Risk.** What could go wrong in building it, or in having built it? The consequences of being wrong differ.
- **Opportunity cost.** What else could the same effort accomplish?
- **Responsibility.** What consequences would accepting this obligation make us responsible for? A system can be desired, technically feasible, and affordable while still creating consequences engineers should not accept.

Suppose a team learns that customers would value four improvements: accessibility remediation, a new collaboration feature, twice the current performance, and enterprise single sign-on with auditing. The team has enough capacity to deliver only two. Which two should it promise? There is not enough information to answer, and that is the point. Engineers need to know who benefits, which obligations are mandatory, how much value each creates, what dependencies each introduces, what each will cost over its lifetime, what risks each creates, and what opportunities are lost by choosing it.

The same gap appears at larger scale. Consider an organization preparing to replace or substantially change an enterprise system. Stakeholders may already have produced a large spreadsheet of requested features. That list can contain substantial useful information and still be insufficient to establish a credible scope, cost, or implementation plan (Kostova et al. 2019). Different groups often know different parts of the system. Management may understand organizational strategy, regulation, budgets, and desired outcomes. Operational staff may understand actual workflows, exceptions, recurring friction, and workarounds. Technical staff may understand dependencies and constraints invisible to both groups.

Requirements work brings these perspectives together. Workshops, analysis, and negotiation can expose components and dependencies, reconcile competing concerns, identify organizational changes, sequence work, and produce a more credible estimate. The result is not simply a longer requirements list but a better basis for commitment — because *a list of requirements is not the ability to commit.* The second transition in the model is its own engineering work, not a signature at the bottom of the first.

Sometimes the right decision is to learn more

Of the four outcomes, learning more is the one engineers most often fail to take deliberately. When uncertainty could change whether a commitment should be accepted, three questions structure the choice: What don't we know? Could knowing it change the decision? What would it cost to find out? If nothing we could learn would change the decision, more information is not worth buying; decide with what is known. If a cheap investigation could flip the decision, buying it first is the defensible move.

The same mechanisms used to discover value can now buy information about a specific commitment. Interview another stakeholder when the uncertainty is about value. Observe the current workflow when the uncertainty is about tacit needs. Investigate a dependency when the uncertainty is about feasibility. Improve an estimate when the uncertainty is about cost. Build a prototype when nothing short of a concrete artifact will produce the evidence.

Learning more is not free. Investigation consumes the same capacity delivery does, and a deferred decision is sometimes a decision made badly by default. The chapter's standard supplies the test: buy information while it can still change the decision; stop when the commitment can be defended.

Requirements are commitments under scarcity

Why does the commitment decision deserve this machinery? Because engineering capacity is finite and an accepted obligation is long-lived. Accepting one obligation consumes resources that could have served another. The cost of a requirement therefore includes not only what it takes to satisfy it, but also the value of what the organization gives up by doing so. This is opportunity cost. Requirements engineering asks not merely *would this create value?* but *is this an obligation we can responsibly accept?*

Accepting also changes the engineering problem itself. The requirement becomes something future architecture, design, implementation, validation, operations, and maintenance must preserve.

Cheap implementation does not change this. Suppose a stakeholder requests a feature and asks why it should cost much when a software agent can implement it in minutes. The premise may be partly correct: implementation can be cheap. But implementation is only one part of the cost created by accepting an obligation. The feature may still require requirements work, design, integration, validation, security analysis, deployment, operations, maintenance, support, and future changes. It may interact with other obligations or increase the cost of satisfying them.

These surrounding costs are not immutable. Better architecture can isolate change. Automation can reduce validation cost. Reusable infrastructure can reduce operations cost. Better representations can reduce the cost of understanding the system. But cheaper implementation does not reduce the lifecycle cost of a requirement to zero. It changes the economics of what we promise; it does not abolish those economics.

Weighing cost at all requires estimates, and estimates themselves depend on requirements. A simple conceptual model is: Cost ≈ Size × Cost per unit × Cost drivers. Size describes how much capability must be produced. Cost per unit reflects what an organization can historically produce with a given amount of effort. Cost drivers capture properties that make apparently similar amounts of functionality more or less expensive: assurance, integration, performance, migration, unusual expertise, operational constraints, and other obligations. Requirements affect all three.

This creates an unavoidable loop. Engineers need some understanding of scope to estimate cost, while estimates help determine which scope is worth accepting. Requirements

and estimates therefore become more precise together: precise enough, by the chapter's standard, to defend the commitment, not precise enough to eliminate uncertainty.

> **NOTE · Estimate ≠ budget ≠ target ≠ commitment**
> An estimate predicts. A budget allocates resources. A target expresses a desired outcome. A commitment is a promise. Confusing them can turn uncertainty in an estimate into an impossible engineering obligation.

The judgments remain coupled

Discovery and commitment are coupled rather than sequential. Trying to decide what can responsibly be promised often reveals something new about what would create value, and new discoveries can change which commitments engineers are prepared to accept. Attempting to decide generates learning, and the learning re-enters discovery. An estimate may expose a dependency that changes the proposed requirement. A prototype may show that a requested workflow does not solve the underlying problem. Negotiation may reveal that one stakeholder's requirement conflicts with another's. Engineers may reject one candidate and discover an alternative that achieves the same purpose at lower cost or risk.

Nor does acceptance end the coupling. What engineers learn through specification, architecture, design, implementation, validation, operations, and use may expose missing obligations or show that an earlier commitment should change. An accepted requirement is a commitment, not a claim that engineers will never learn anything that warrants reconsidering it.

From requirements to specification

Requirements engineering has now decided what to promise. Discovery produced candidate obligations, and the commitment decision — weighing value, feasibility, cost, dependencies, risk, opportunity cost, and responsibility — accepted, revised, deferred, or rejected each one. Specification determines what the accepted promises require of the machine and its environment, while preserving the freedom they genuinely leave open.

An accepted obligation may still leave many possible systems. "The system follows security best practices" is an obligation, but it does not yet establish what practices count, which threats matter, or what evidence is sufficient. "Alert the user when their sleep pattern suggests illness" says something important about the desired product, yet accepting it leaves unresolved which observations justify an alert, when the alert should occur, and what behavior counts as satisfying the promise. Knowing what we are willing to promise does not yet tell an implementor which realizations would satisfy the promise.

Requirements engineering asks: *what should we promise?* Specification — the subject of Specification — asks: *what exactly are we promising?*

Summary

Requirements engineering turns uncertainty about what matters into engineering commitments. It rests on two coupled judgments: *What would create value?* and *What can we*

responsibly promise? Asking, observing, comparing, and building buy different kinds of information about value, and choosing among them by where the missing knowledge lives is itself an engineering judgment. Discovery produces candidate requirements; the commitment decision weighs value, feasibility, cost, dependencies, risk, opportunity cost, and responsibility. A candidate can be accepted, revised, held while engineers learn more, or rejected, and only acceptance creates a requirement.

The two judgments remain coupled. Attempting to decide generates learning that re-enters discovery, and new discoveries change which commitments engineers are prepared to accept. Requirements engineering therefore does not eliminate uncertainty before commitment; it reduces uncertainty enough that a commitment can be defended. Accepted requirements pass to specification, which determines what those promises require of the machine and its environment.

READ FURTHER

Nuseibeh, Bashar, and Steve Easterbrook. "Requirements Engineering: A Roadmap." In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, 35–46. New York: ACM, 2000. A concise map of requirements engineering: elicitation, modeling, analysis, negotiation, and evolution.

Patton, Jeff. "The New User Story Backlog Is a Map." Jeff Patton & Associates, October 8, 2008. A practitioner's argument for organizing requirements around what users are trying to accomplish rather than a flat feature list.

Kostova, Blagovesta, Lucien Etzlinger, David Derrier, Gil Regev, and Alain Wegmann. "Requirements Elicitation with a Service Canvas for Packaged Enterprise Systems." In *2019 IEEE 27th International Requirements Engineering Conference (RE)*, 340–350. IEEE, 2019. An industrial case study of what a customer's initial requirements list failed to capture.

CHAPTER 6

Specification

Premise. *Requirements describe what we have committed to achieve in the world. Specification makes those commitments actionable by deciding where precision is needed, what the environment may be assumed to provide, and what the machine must guarantee.*

A requirement rarely determines a unique implementation. "Users are warned before illness affects their day" might admit many realizations, some useful and some plainly unacceptable. The requirement does not say exactly what evidence counts as illness, when the warning must occur, what information the machine may rely on, or what should happen when expected information is unavailable. Those distinctions become important when engineers must decide what an acceptable realization may actually do.

Specification is the engineering work of making the consequential distinctions explicit. It does not mean describing every detail before implementation begins. Some choices matter and must be constrained. Some alternatives are genuinely interchangeable and should remain open. Some choices cannot yet be made responsibly because their consequences are not understood. The engineer therefore faces three related judgments.

KEY-IDEA · The three judgments of specification

1. **Where should specification effort go?** Which aspects of the accepted requirements are sufficiently consequential and uncertain to deserve further specification?
2. **Where does responsibility belong?** What may the machine assume about its environment, and what must the machine itself guarantee?
3. **How tightly should the machine be constrained?** Which distinctions must be fixed, which may remain free, and which require more learning before either decision can be made?

Together, these judgments turn a requirement into a defensible boundary around acceptable realizations.

Prioritizing specification effort

Requirements establish commitments, but they do not make every part of those commitments equally deserving of specification effort. Some aspects are consequential but already well understood. They may need to be stated precisely, but little discovery is required. Other aspects are uncertain but inconsequential: several possible answers would be acceptable, so resolving the uncertainty may provide little value. The most valuable targets for specification are often the aspects that are both consequential and uncertain.

Consider the sleep-advisor requirement that users be warned when their sleep pattern suggests illness. Suppose the team already knows that a weekly summary should contain seven days of sleep measurements. The exact schema still needs to be stated, but there may be little uncertainty to resolve. By contrast, the meaning of "warned" may be highly consequential and poorly understood. Is a warning useful only before the user leaves home? Must it appear within a minute of waking? Is an advisory justified after one abnormal

night, or only after a pattern develops? Different answers can substantially change both the usefulness of the product and the behavior the machine must provide.

This gives specification effort an ordering principle: ask which unresolved distinctions could materially change whether the commitment is satisfied, then ask how uncertain those distinctions remain. Consequential and uncertain aspects deserve representation, investigation, and feedback first. This is the same economic logic encountered throughout engineering. Precision has a cost, investigation has a cost, and delay has a cost. The purpose of specification is not to maximize detail but to spend engineering attention where additional precision or knowledge can change an important decision.

Allocating responsibility between environment and machine

A second judgment concerns the boundary of the machine itself. Requirements usually describe desired effects in the world, while a specification constrains behavior the machine can provide. The two are connected by what engineers assume about the environment. Zave and Jackson (1997) express this relationship compactly. Let R denote a requirement about the world, S a specification of machine behavior at its boundary with that world, and E the relevant properties and assumptions of the environment. An adequate specification should support the argument that, given that the assumed environment E holds and the machine satisfies specification S, requirement R must follow: $E \wedge S \Rightarrow R$.

The relationship is important, but the engineering work lies in constructing its terms. A requirement does not uniquely determine E and S. Engineers decide what responsibility may reasonably remain with the environment and what responsibility the machine should assume. Consider a wearable sleep advisor. One possible environmental assumption is that the user wears the device while sleeping and keeps it sufficiently charged. Under those assumptions, the machine might be responsible only for interpreting the measurements it receives and issuing the appropriate advisory. A different system could assume less of the user: it might monitor battery state, remind the user to charge the device, detect missing measurements, and explain when insufficient data prevents an advisory. The requirement may be unchanged, but the boundary of machine responsibility has moved.

This is a pervasive engineering decision. Automation often moves responsibility from the environment into the machine: an action once expected of an operator becomes behavior the software must perform. Reducing scope can move responsibility in the other direction by strengthening the assumptions under which the machine is expected to operate. A first version of a system might require a trained operator, accept only well-formed inputs, or be deployed only in a controlled setting. De-risking can make the same move temporarily, placing uncertain responsibilities outside the machine until engineers understand them well enough to automate safely. Scope expansion moves the boundary in the opposite direction. “The operator will ensure that inputs are valid” becomes “the system should detect and repair invalid inputs”; “the service will run only on the corporate network” becomes “the service should also support public access.” What looks like another feature is often a transfer of responsibility from the environment to the machine.

The complexity has not disappeared in any of these cases. It has been allocated. A simpler machine may require a more capable operator, a more controlled deployment environment, stronger operating procedures, or assumptions that exclude some users. Those consequences matter. An allocation that makes implementation easier may be unacceptable if it places an unreasonable burden on users, undermines accessibility, or relies on assumptions that cannot be trusted in practice. Specification therefore asks not merely, "What must the software do?" It asks, "What must be true for the requirement to hold, and which parts of making it true should be the machine's responsibility?" The formal relationship tells us what must be true for a specification to be adequate; the engineering judgment is choosing the assumptions and obligations that make it so.

From responsibility to acceptable realizations

Once a responsibility has been assigned to the machine, specification still does not determine a unique implementation. It establishes a region of acceptable machine behavior. Suppose the sleep advisor must issue an advisory when an inferred condition crosses an agreed threshold. One realization might evaluate the condition locally on the wearable, another might transmit measurements to a phone, and a third might evaluate them in a cloud service. If the requirement does not make those differences consequential, the specification need not choose among them. Other distinctions may matter greatly: an advisory delivered six hours after waking may not satisfy a requirement whose value depends on warning the user before the day begins; a system that silently produces no result when data are missing may be unacceptable even if its behavior on complete data is correct; a realization that shares information beyond the intended recipient may violate an obligation even though every calculation is accurate.

A specification therefore bounds a realization space. Each obligation excludes some possible realizations while leaving others available. The objective is not to make that space as small as possible, but to exclude realizations that differ in consequential ways while preserving choices whose alternatives are genuinely acceptable. This distinction matters because every additional constraint consumes freedom. A specification that unnecessarily selects a particular data structure, algorithm, deployment topology, or interaction sequence can convert a cheap future decision into an expensive present commitment. Conversely, leaving a consequential distinction unresolved merely postpones a decision that the implementation will eventually make anyway, perhaps accidentally. Good specification is selective constraint: enough to control consequential variation, but no more than the engineering situation justifies.

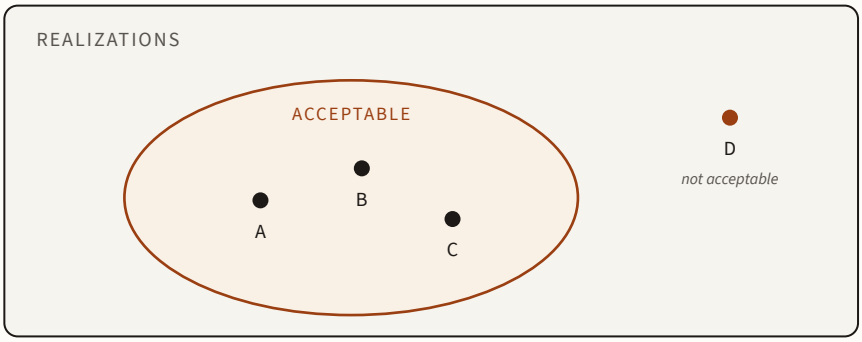


Figure 4. A specification bounds a realization space. Realizations A, B, and C lie within the acceptable region; realization D does not.

Not constrained is not the same as free

An unconstrained choice can have three importantly different meanings. It may be **unknown**: we do not yet understand the consequences well enough to decide, perhaps because we do not know whether a one-minute or ten-minute delay affects usefulness or whether an inference technique behaves adequately across the intended population. It may be **tacit**: a consequential boundary exists, but it has not been made explicit, as when stakeholders believe that "obviously" an advisory should appear before the user leaves home while nothing in the specification says so. Or it may genuinely be **free**: the alternatives are acceptable, such as two internal data structures whose differences have no relevant consequence for the obligations currently being engineered. Only the third is a deliberate degree of freedom.

> **DEFINITION · Degree of freedom**
> A degree of freedom is a choice deliberately left to the implementor because its permitted alternatives are acceptable within the bounds the engineering effort has established.

A useful diagnostic for tacit obligations is the reaction "not that." If an implementation appears to satisfy the written specification but a stakeholder immediately rejects it—"not that late," "not that much information," "not visible to that person"—then some consequential distinction existed outside the specification. The rejection is evidence that an apparent degree of freedom was not actually free. Distinguishing unknown, tacit, and free prevents two opposite mistakes: treating every unspecified choice as free allows consequential decisions to be made accidentally during implementation, while treating every unspecified choice as a defect encourages overspecification and destroys useful freedom.

Representing consequential properties

Knowing that a distinction matters does not tell us how best to reason about it. Different engineering questions expose different properties of the same system, and those properties often require different representations. The A-7E aircraft specification developed by

Alspaugh et al. (1992) provides a useful example. It does not attempt to express the entire system in one notation. It combines prose, defined data, mode-transition tables, timing constraints, descriptions of required subsets, and explicit discussion of expected changes. The representations differ because the engineering questions differ.

The A-7E specification constrains detailed externally visible behavior without prescribing the code that produces it. Modes, conditions, events, and required responses make behavioral cases inspectable while leaving realization choices open. Timing constraints expose obligations that ordinary prose can easily conceal. Requirements for useful subsets rule out realizations whose behavior is correct only when the complete system is assembled. Statements about expected changes distinguish assumptions designers may reasonably treat as stable from aspects of the system that should remain easier to change. These representations do not compete to be the one complete description of the system; each makes a particular class of consequential distinctions easier to see.

DEFINITION · Model
A model is a purposeful view. It preserves the distinctions needed to answer an engineering question and suppresses details that do not matter to that question.

A state model can expose legal states and transitions; an activity or sequence model can expose required ordering; a table can expose combinations of cases, values, and bounds; a schema can constrain the shape of information crossing the machine boundary. These are reasoning instruments rather than checklist artifacts. The pattern is engineering question → representation → property → analysis or check.

DEFINITION · Property and invariant
A property is a claim expressible over a model. An invariant is a property required to hold over its declared domain.

The representation does not create the obligation; it gives engineers a vocabulary in which the obligation can be stated, inspected, discussed, and sometimes checked mechanically.

This explains why specification should begin with consequential uncertainty rather than with a preferred notation. Drawing a state machine because "specifications should have state machines" reverses the reasoning. First identify the distinction whose consequences need to be understood, then choose a representation that makes the relevant property visible. No single representation needs to say everything, and the same machine specification may contain several peer views connected through shared concepts. Hold the system constant, change the engineering question, and the useful model may change with it.

Constrain, leave open, or learn more

For each consequential distinction exposed by specification, the engineer ultimately has three choices. **Constrain** when alternatives differ consequentially and there is sufficient evidence to defend a boundary: if an advisory must occur before a particular event to create value, that timing belongs in the specification. **Leave open** when the alternatives are gen-

uinely acceptable, preserving freedom so that later design and implementation can exploit information not yet available and avoiding payment today for decisions that need not yet be made. **Learn more** when the consequences are important but not sufficiently understood. A prototype may reveal whether users notice a particular advisory, a measurement may establish whether a timing target is feasible, or an experiment may reveal whether two apparently equivalent algorithms behave differently under realistic workloads. Learning is not a failure to specify. When the value of information exceeds the cost of obtaining it, learning is the engineering decision.

These choices correspond to the earlier distinction among unknown, tacit, and free. Unknown choices call for learning when their consequences justify the effort. Tacit choices call for externalization: the hidden boundary must be surfaced and evaluated. Free choices should remain open unless later evidence makes the distinction consequential. The important question is therefore not merely "Is this specified?" but "Do we understand why this choice is constrained, open, or still under investigation?"

Software lets specifications learn

Software's changeability (Software Engineering) makes the timing of specification unusually important. In some engineering settings, important choices must be fixed before fabrication begins because changing them later is prohibitively expensive. Software often allows more decisions to remain reversible. That does not make specification unnecessary; it changes when precision is economical. Some obligations must be settled early because they determine interfaces, data representations, safety properties, regulatory commitments, or architectural choices that become expensive to reverse. Others can remain open until implementation or use provides better information. Engineers can build a realization, observe its consequences, and discover that an apparent freedom was consequential after all.

The resulting process is iterative: specify → build → observe → learn → revise. Feedback can revise any part of the specification argument. Engineers may discover that an environmental assumption was false, requiring the machine to accept more responsibility; that a machine obligation was unnecessarily strong and can safely be relaxed; or that a consequential distinction was captured by neither the requirement nor the specification. Occasionally the requirement itself should be reconsidered. Overspecification is costly precisely because it suppresses this learning by converting cheap future choices into expensive present commitments. Underspecification is costly because consequential choices still get made, but without deliberate reasoning. The goal is not maximum specification. It is enough specification to control consequential freedom while preserving useful optionality.

From specification to architecture

A specification defines what the machine must guarantee while leaving multiple acceptable realizations available. Architecture (Architecture) begins when engineers must organize one realization so that those obligations can coexist. The transition is not absolute: specification decisions can have architectural consequences, while architectural exploration can

expose problems in the specification. Requiring independent deployment, strong isolation, a particular latency bound, or operation under partial failure may rule out large classes of architectures. Conversely, architectural exploration can reveal that a specification is unexpectedly expensive or internally difficult to satisfy and motivate its reconsideration. The distinction is nevertheless useful. Specification asks which machine behaviors and properties are acceptable; architecture asks how a system can be organized to provide them together.

The realization space therefore narrows progressively. Requirements establish the commitments worth making. Specification determines what responsibility belongs to the machine and bounds the machine behaviors that can satisfy those commitments. Architecture chooses a system structure within that space. Design resolves further choices inside that structure, and implementation eventually selects concrete representations of behavior. At every stage, some choices are fixed and others remain open. Good engineering preserves that freedom deliberately rather than accidentally.

Summary

Specification turns accepted requirements into a defensible boundary around acceptable realizations. Engineers first decide where specification effort is worth spending: consequential and uncertain aspects of accepted requirements deserve attention first, while inconsequential uncertainty can often remain unresolved and consequential matters that are already understood may simply need to be stated precisely. They then decide where responsibility belongs. Requirements concern desired effects in the world, but those effects depend jointly on properties of the environment and guarantees made by the machine; automation, scope reduction, scope expansion, and de-risking can all move responsibilities across that boundary. Finally, engineers decide how tightly the machine should be constrained. Consequential distinctions should be constrained when they are understood, genuinely acceptable alternatives should remain free, and consequential uncertainties should trigger learning rather than arbitrary commitment.

Models support these judgments by providing purposeful views of the properties engineers need to reason about. State models, activity models, tables, schemas, and other representations are useful when they expose a consequential distinction; no notation is valuable merely because it is conventional. A good specification is therefore neither the most detailed description nor the one that leaves the most freedom. It makes consequential obligations explicit, places responsibility deliberately, preserves genuine degrees of freedom, and identifies what must still be learned before further commitments can responsibly be made.

READ FURTHER

Zave, Pamela, and Michael Jackson. "Four Dark Corners of Requirements Engineering." *ACM Transactions on Software Engineering and Methodology* 6, no. 1 (1997): 1–30. The classic account of the relationships among requirements, domain assumptions, and specifications.

Alspaugh, Thomas A., Stuart R. Faulk, Kathryn Heninger Britton, R. Alan Parker, David L. Parnas, and John E. Shore. *Software Requirements for the A-7E Aircraft*. NRL/FR/5530-92-9194. Washington, DC: Naval Research Laboratory, 1992. A substantial real specification showing how different representations constrain externally visible behavior without prescribing implementation. Skim it rather than reading linearly.

CHAPTER 7

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: things on the same side may be understood, changed, and operated together; things on opposite sides should be understandable apart.

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 corrupting 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 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 responsibilities 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.

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.

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.
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. 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. Boston: Addison-Wesley, 2012. The standard treatment of how quality attributes drive architectural decisions.

Fairbanks, George. *Just Enough Software Architecture: A Risk-Driven Approach*. Boulder, CO: 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.

CHAPTER 8

Design

Premise. *Design chooses mechanisms by which parts fulfill their responsibilities within inherited constraints.*

Architecture establishes consequential parts, assigns responsibilities, and defines important boundaries and interactions. These decisions constrain the system without determining how each part works internally. A service responsible for processing work may still require choices about algorithms, data structures, state representation, caching, concurrency, failure handling, and resource management. Design resolves these choices so that each part can fulfill its assigned responsibility while satisfying the obligations it inherits.

Architecture can recur during design. Examining a component may reveal that it is itself too large to reason about directly and should be organized into consequential subparts with distinct responsibilities and interactions. This creates another architectural problem at a smaller scope. Once the relevant organization is sufficiently established, however, a different judgment remains: *How should this part work?* That is the central question of design.

Design is therefore better understood as a class of engineering decisions than as a phase of development. A designer identifies consequential choices that remain open, generates plausible mechanisms, reasons about their consequences, and resolves them while recognizing when an apparently local choice belongs elsewhere.

> **DEFINITION · Software design**
> Software design chooses mechanisms by which an architectural part fulfills its responsibility within the obligations, affordances, and constraints it inherits. It operates over the degrees of freedom that remain, progressively resolving them while recognizing when their consequences escape the scope of the part.

This relationship can recur. Opening one part may reveal consequential subparts that require their own architectural organization. Eventually engineers reach functions, data structures, algorithms, or small collaborations whose relevant behavior can be reasoned about directly. At that point, design passes into implementation.

Working within an architecture

The previous chapter considered the wall from the architect's perspective. Now consider it from the plumbing engineer's. The building architect has determined where the wall stands, how deep it is, where service space exists, and what penetrations are allowed. The plumbing engineer inherits those decisions. They constrain the solution without determining it.

Substantial engineering choices remain. The plumbing engineer must determine what pipe diameter can supply the upper floors of a sixty-story building, what materials can withstand the required pressures, how pressure zones should be arranged, and where pumps

or pressure-reducing valves are needed. These choices require calculation, evidence, and judgment. Yet they can ordinarily be made without reconsidering where the wall stands.

Software design works similarly. Design inherits responsibilities, boundaries, and interactions established by the architecture and chooses mechanisms by which each part fulfills its responsibility: algorithms, data structures, internal representations, caching strategies, concurrency mechanisms, resource-management strategies, and other such means. The inherited organization constrains the choice; it does not make the choice.

What Design inherits

Design does not begin from an unconstrained set of possible implementations. Earlier engineering decisions have already reduced that space. Specification establishes properties that must remain true. Architecture assigns responsibilities and establishes boundaries, interfaces, and interaction rules. The engineering environment may supply frameworks, conventions, shared abstractions, policies, and mechanisms that apply across many parts of the system.

The engineering environment can constrain design substantially without appearing on an architectural diagram. An organization may standardize dependency injection, persistent-state access, cross-service deadlines, background queues, retries and dead-letter handling, logging, authentication, configuration, serialization, or transaction management. These mechanisms represent decisions that local designers ordinarily should not make again. A good engineering environment converts recurring judgment into engineering structure, reducing the number of independent choices that must be understood across the system.

The choices not already determined are the remaining degrees of freedom. For each consequential choice, the designer must determine what kind of choice it is:

- **Follow** when an inherited decision already determines or sufficiently constrains the mechanism.
- **Choose** when several mechanisms remain viable and their consequential differences can be resolved within the responsibility being designed.
- **Escalate** when no satisfactory mechanism can be selected without reconsidering an inherited decision.

The distinctive work of design lies primarily in choose. The responsibility of the part and the constraints under which it must operate are known, but several mechanisms could plausibly satisfy them. The engineering task is to identify which differences among those mechanisms matter and select accordingly.

Choosing mechanisms

Consider a document-processing service deployed on a cloud worker. Its architecture assigns responsibility for processing documents to the service, while system requirements constrain operating cost. Suppose the cloud provider's pricing makes memory consumption consequential: a worker requiring more than 4 GB of memory must use a more expensive tier, causing the system to exceed its cost target.

The architecture need not determine how the service remains below 4 GB. Several mechanisms may satisfy the same responsibility. The service might load an entire document into memory, process it through a bounded stream, or construct a compact intermediate representation. These alternatives differ in peak memory, latency, implementation complexity, and changeability. If whole-document processing requires 6 GB while streaming requires 1 GB, the inherited cost constraint makes memory consumption a consequential property of the design.

Design decisions take many forms. Engineers select algorithms, data structures, state representations, caching and batching policies, scheduling and concurrency mechanisms, retry strategies, memory lifetimes, and internal control flow. Computer science supplies many of these mechanisms and theories for understanding their properties. Design concerns their use in a particular engineering context: *Which mechanism should be used here, given the obligations this part must satisfy?*

A choice is rarely determined by one property. An in-memory representation may simplify an algorithm while consuming excessive memory. A cache may reduce latency while introducing invalidation and consistency problems. Asynchronous processing may improve throughput and failure isolation while complicating ordering and retries. Indirection may isolate an expected change while introducing another abstraction that engineers must understand and maintain. The relevant comparison therefore depends on the obligations of the particular system rather than on whether a mechanism is generally considered desirable.

Recurring Design questions

Many design choices recur across systems. Prior experience supplies candidate mechanisms and known consequences, but does not determine which mechanism fits the obligations of the part being designed. Common questions include:

- **Who owns the truth?** When several representations contain the same information, identify which is authoritative when they disagree. Caches, replicas, and derived views may improve other properties without acquiring authority.
- **What consistency must copies provide?** Stronger observation guarantees simplify assumptions for clients but usually require more coordination. Weaker guarantees can improve autonomy, availability, or latency while requiring the system to tolerate temporary disagreement.
- **Must this work happen now?** Synchronous work provides simple completion semantics but places its latency and failures on the caller's critical path. Deferred work can isolate latency and failure while introducing retries, idempotence, ordering, and eventual-completion concerns.
- **How directly should parts depend on one another?** Indirection can isolate expected change but introduces concepts and relationships that engineers must understand. Ask what consequential change or property the additional seam protects.
- **What resources does the mechanism consume?** Memory, compute, storage, network traffic, locks, connections, and other finite resources can turn an otherwise local implementation choice into an engineering decision.

- **What happens when the mechanism fails?** Retries, fallback, partial progress, duplicate execution, cleanup, and recovery may matter as much as the successful path.

Design patterns, frameworks, and conventions package accumulated answers to recurring questions such as these. Their value is not that a named pattern should be used whenever it applies syntactically. They expand the candidate set by preserving experience about a recurring problem, plausible mechanisms, and their consequences. The designer must still determine whether those consequences fit this system.

Evidence for a Design decision

A consequential design choice requires enough evidence to distinguish among plausible mechanisms. The appropriate evidence depends on the properties that separate the alternatives.

For the cloud worker, a memory model or prototype measurement might establish whether a representation remains below 4 GB. If latency distinguishes two algorithms, an analytical model or benchmark may be appropriate. If concurrent workers can process the same job, a lifecycle or state model may expose whether a coordination mechanism preserves ownership. When implementation complexity is the principal uncertainty, implementing small versions of competing alternatives may provide better evidence than attempting to predict their relative costs.

Different choices require different evidence. A behavioral model may expose ordering; an ownership or lifecycle model may expose shared state; a dependency model may expose replaceability; a quantitative model may expose latency, memory, or cost. The same part can therefore have several useful representations because each removes details irrelevant to a different engineering question.

There is no single artifact that is "the design." Models, analyses, prototypes, measurements, and implementations are means of reducing uncertainty about a design choice. Use the smallest representation that exposes the consequential difference among the alternatives with sufficient confidence to decide.

> **KEY-IDEA · Representations serve decisions**
> A representation earns its place by helping resolve an engineering decision. Start with the alternatives and the consequential difference among them; then choose the evidence that makes that difference tractable.

Making Design reasoning inspectable

Consequential reasoning should not disappear into the resulting code. An implementation often reveals what was chosen while concealing which alternatives were considered, which assumptions mattered, why the chosen mechanism prevailed, and what evidence would justify revisiting it.

A useful design document preserves enough of that argument for another engineer to inspect it. The format can vary, but the substance usually includes:

1. Context and inherited constraints. What responsibility and obligations are already fixed?

2. Open decision. What consequential degree of freedom remains?
3. Alternatives. What serious mechanisms could satisfy it?
4. Consequences. What properties distinguish those alternatives?
5. Evidence. What model, analysis, prototype, measurement, or experience supports the comparison?
6. Decision and rationale. What was chosen, and why?
7. Uncertainty. What assumptions remain, and what evidence should cause the decision to be revisited?

Review then becomes part of design rather than a ceremonial approval step. Another engineer can challenge assumptions, identify alternatives, or expose consequences before the choice becomes expensive to reverse.

Local decisions, system consequences

A design can be sound within the responsibility assigned to a part and still contribute to an unsound system. Specification establishes what the machine must guarantee; Architecture organizes responsibilities, boundaries, and interactions so that those obligations can be realized together and their satisfaction can be assessed at the level of the whole machine. Design occurs within that organization, but individually reasonable choices can accumulate or interact in ways that cause the resulting machine to violate its specification. **Systems thinking** requires engineers to reason about these aggregate effects rather than evaluating each design choice only within its local scope.

A mickle and a mickle makes a muckle. Small local costs accumulate. Suppose an architecture establishes an end-to-end latency budget of 500 ms across five sequential components. Each component might independently choose a mechanism that responds within 400 ms and reasonably conclude that its own performance is acceptable. The resulting system is not. The architectural property concerns the path through the components, not the local acceptability of any one component.

The same problem can arise from gaps rather than accumulation. Suppose components A and B both participate in constructing a database operation. The design of A assumes that B will ensure untrusted input cannot alter the structure of the query, while the design of B assumes that A has already established that property. Each design may appear reasonable locally, yet their composition leaves the obligation unsatisfied. The problem is not either mechanism considered alone; it is the uncovered responsibility between them.

Systems thinking therefore asks whether the collection of design choices preserves the properties that the architecture sought to control. In particular, designers should ask whether:

- **Budgets compose.** Local consumption of latency, memory, cost, error, or another finite budget remains acceptable in aggregate.
- **Responsibilities cover the obligation.** Something is actually responsible for each required property; assumptions do not leave gaps between parts.
- **Assumptions compose.** What one part expects of another is actually guaranteed there.
- **Mechanisms interact safely.** Individually sound choices do not interfere when combined.

- **Architectural properties remain true.** Local choices preserve the isolation, dependency, security, reliability, or other properties for which the architecture was organized.

Local evidence is therefore necessary but not always sufficient. Design must sometimes establish not merely that each mechanism works, but that the mechanisms work together.

When Design exposes a larger problem

Some apparent degrees of freedom should not be resolved locally. If the engineering environment already establishes how dependencies are injected, persistent state is accessed, or retries are performed, a component should ordinarily follow that decision. Allowing each component to choose independently would increase the number of mechanisms that engineers must understand and the number of interactions the system must accommodate.

Design can also establish that an apparent local choice must be escalated. Suppose no plausible processing mechanism can remain below the worker's memory limit while satisfying the service's other obligations. Alternatively, suppose the only viable mechanism requires moving authoritative state across a boundary established by the architecture. In either case, design has produced evidence that the inherited constraints do not admit a satisfactory mechanism.

In the building analogy, the plumbing engineer may discover that every pipe capable of supplying the required flow is too large for the service space the architecture provides.

Software Design can expose the same problem. If no satisfactory mechanism fits within the inherited responsibilities, boundaries, or interactions, the problem cannot be resolved as a local Design choice. The inherited decision must be reconsidered.

The correct response is not to force a clever workaround into the implementation; it is to take the discovery to the decision whose scope actually contains the consequence. Escalation can therefore have several destinations:

- **Engineering environment:** the same decision recurs across components and should become a shared convention, abstraction, mechanism, or check.
- **Architecture:** the conflict concerns responsibilities, boundaries, interactions, system-wide budgets, or another property of consequential organization.
- **Specification:** detailed work reveals a missing, contradictory, or infeasible obligation.

Escalation does more than repair the current design. It places what Design has learned at the scope where future engineering decisions can inherit it.

The distinction matters because a local workaround can make an earlier engineering model false.

When a local workaround makes the architecture false

Suppose the architecture establishes A → Port → B because engineers need A to remain independent of B's internals. During design, an engineer discovers that one feature would be easier if A reached directly into B. The feature works and its local tests pass, but the architectural dependency model is now false.

The problem is deeper than untidy code. The architecture previously supported an engineering inference: B can be replaced without changing A. Once the hidden dependency

exists, that inference is no longer trustworthy. A local design decision has compromised the predictive value of the system's engineering knowledge.

Architectural degradation is therefore partly an epistemic failure. Models are useful because engineers can reason from them without repeatedly reconstructing the implementation. When local choices silently violate those models, apparently sound engineering reasoning can produce incorrect conclusions.

A consequential workaround should therefore become visible. Remove it if it was a mistake. Represent it as an explicit exception if it is justified. Change the architectural rule if repeated exceptions show that the rule itself is wrong. Local implementation should not silently redefine the system engineers believe they have.

Cheap implementation changes the evidence

Implementation can itself serve as a design probe. When implementation is expensive, engineers often must choose among mechanisms using models, prior experience, or small prototypes. As implementation becomes cheaper, they can sometimes build multiple plausible mechanisms far enough to observe the consequences that matter. Two internal representations can be implemented and measured for memory use; two batching strategies can be exercised under representative load; an immediate and a deferred workflow can be compared against realistic failure conditions. The resulting code need not survive. Its purpose is to produce evidence for the design decision.

This changes the relationship between Design and Implementation. Implementation does not merely follow a completed design decision; it can be one of the instruments used to make that decision. The engineer still determines what alternatives deserve comparison, which consequences matter, what evidence would distinguish them, and when the evidence is sufficient to choose. Cheap implementation reduces the cost of obtaining some kinds of evidence. It does not remove the judgment required to interpret that evidence.

Cheaper implementation can also change what engineers are able to notice. Suppose three related failures arise weeks apart. Each may appear to be an isolated local problem and be repaired independently. If implementation and change occur quickly enough that the same failures arise in dense succession, their relationship may become visible. What appeared to be several incidents can become evidence of one structural cause.

High velocity does not guarantee better design. It changes the evidence available to the engineer. Temporal compression can make recurring structure easier to recognize, while also producing more changes and failures than a person can inspect individually. Human attention can therefore become the limiting resource.

At that point, the design problem shifts again. Engineers must decide which observations are local, which indicate a shared mechanism or architectural weakness, and which recurring lessons should become durable engineering knowledge. Cheap code can create a firehose of evidence; engineering judgment determines what that evidence means.

The Cheap Code, Costly Judgment case study provides one example. Failures appearing in dense succession revealed that several local incidents shared an architectural cause. Rather than repair each incident independently, the engineers changed the engineering

environment: they represented the underlying knowledge explicitly, derived checks from it, and prevented a class of related failures.

CODA

Implementation in the GenAI Era

This handbook does not devote a chapter to implementation. That differs from many software-engineering texts. Sommerville, for example, treats design and implementation together. Historically, software engineering never developed a widely accepted division between engineers who make consequential decisions and technicians who realize those decisions in code. Programmers have ordinarily done both.

There was good reason for this arrangement. Producing software required substantial human effort, and the skilled programmer doing that work was also capable of resolving many Design questions encountered along the way. It was often economical to leave such choices until implementation rather than represent every decision beforehand. Software's changeability made this arrangement still more practical: if a choice proved poor, the implementation could often be revised. Much design therefore happened during implementation. But encountering a decision while programming does not make it an implementation decision. If choosing a representation materially affects a memory obligation, or a concurrency mechanism determines whether a correctness property can hold, that choice is Design.

Generative AI changes this arrangement by separating cheap implementation from engineering judgment. An agent can follow a sufficiently determined plan and produce the corresponding code at very low cost. I therefore expect implementation to cease being a distinct engineering activity: increasingly, agents will perform it. This handbook consequently does not devote a chapter to the now-artisanal craft of implementation itself. The engineering task is instead to resolve or constrain the consequential Design decisions well enough that what remains can be delegated reliably and audited afterward.

When consequential choices remain open, however, the agent must also make those Design decisions. Current agents can make choices that are locally plausible but wrong for the system, and even a sound choice remains subject to engineering control: engineers must be able to understand the decision, examine its consequences and evidence, and accept responsibility for it. Consequential Design decisions therefore need to be made explicit rather than left to the agent during implementation. Specification, Architecture, Design, and the engineering environment progressively constrain the work until what remains can safely be delegated.

This does not make implementation unskilled or make engineering a rigid sequence. Programming still requires knowledge of languages, libraries, tools, and execution environments, and implementation will continue to reveal bad assumptions and better alternatives. That evidence should flow back to Design, Architecture, Specification, or the engineering environment. The goal is to make consequential engineering decisions where their alternatives and consequences can be examined, then use implementation to express

and test those decisions reliably and auditably. *Model-Based Agentic Software Engineering* develops the practices needed to do this with capable agents.

Summary

Design chooses mechanisms by which parts fulfill their responsibilities within inherited constraints. Specification determines properties that must remain true; Architecture establishes responsibilities, boundaries, and interactions; the engineering environment resolves recurring decisions that should not be made independently. Design operates over the degrees of freedom that remain: follow decisions already made, choose among genuinely local mechanisms, and escalate when the consequences escape the part.

Choosing requires evidence about consequential differences among alternatives. Models, analyses, prototypes, measurements, design documents, reviews, and implementations can all make that reasoning inspectable. Systems thinking extends the judgment beyond individual choices: locally sound mechanisms must compose without exhausting shared budgets, leaving obligations uncovered, violating assumptions, or compromising the properties the architecture sought to control.

Engineering therefore proceeds downward through constraints and upward through evidence. Detailed design can establish that the inherited architecture works, but it can also expose a missing shared mechanism, an architectural problem, or an obligation that must be reconsidered. Cheap implementation increases the evidence available to make these judgments. It does not make the judgments for us.

READ FURTHER

Meyer, Bertrand. *Object-Oriented Software Construction*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1997. Develops criteria for judging whether a decomposition makes understanding, change, composition, and failure sufficiently local. Focus on the "Modularity" chapter.

Parnas, David L., and Paul C. Clements. "A Rational Design Process: How and Why to Fake It." *IEEE Transactions on Software Engineering* SE-12, no. 2 (1986): 251–57. The classic explanation of why rational design descriptions remain useful even though real design is iterative.

"Design Docs at Google." A practical account of making consequential design reasoning inspectable before it disappears into implementation.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994. Patterns package accumulated design experience, expanding the candidate set without determining which mechanism fits a particular system. Read the introductory and concluding chapters, plus the "Facade" and "Command" patterns.

CHAPTER 9

Validation

Premise. *Validation asks what evidence is sufficient to deliver a software system into the world.*

Engineering decisions flow from purpose through requirements, specification, architecture, design, and implementation. At each point, engineers make choices for which they are responsible. Eventually those choices produce software that can be delivered. The engineer must decide whether there is enough reason to proceed.

This makes validation broader than testing. Tests are one way to obtain evidence about a system, but the engineering question comes first: What do we need to know before we are willing to deliver it? A review may provide useful evidence about a design. Static analysis may provide evidence about possible program behaviors. Measurement may establish whether a performance objective has been met. A proof may establish a property under stated assumptions. Different claims require different evidence.

Nor does validation necessarily seek certainty. Software is unusually updateable. Software Engineering showed how an engineer can change one codebase and propagate the result across an enormous deployed population. The same property that spreads mistakes also makes repairs cheap to distribute. Process drew one consequence: when a decision is cheap to reverse, engineers can experiment rather than spend more avoiding an error than the error would cost to correct.

The same reasoning applies to delivery. Sometimes it is better to deliver software with known uncertainty, observe what happens, and revise it. A game can ship with an occasional graphical defect. An internal tool can be useful despite awkward workflows. A consumer application may need to reach users before its engineers can learn which capabilities people actually value. In these cases, delaying delivery until every known uncertainty has been resolved can cost more than learning from use.

In that limited sense, *move fast and break things* describes a real engineering strategy. It is not always irresponsible to deliver software that might fail. The engineering question is whether discovering the failure after delivery is an acceptable way to learn.

That qualification matters because software's updateability repairs the software, not necessarily the consequences of its previous behavior. An update can correct a graphical defect after players encounter it. It cannot necessarily recover money already lost, make disclosed information private again, or reverse a physical injury. As the consequences of being wrong become more substantial or less reversible, learning through failure becomes more expensive.

Process deliberately left consequence of failure out of its model for organizing engineering work. Consequence does not tell us whether requirements can be known in advance, whether a decision is expensive to reverse, or whether a partial system can be built and validated. It answers a different question: how much assurance should we demand before delivery?

That is the question of this chapter.

Acceptance is a professional judgment

Consider three software defects. A game sometimes draws a character incorrectly. A TODO application occasionally loses a task. A medical device can sometimes deliver an incorrect dose.

These defects differ in more than severity. The game defect may directly annoy a player, but more serious consequences require a longer causal chain. The TODO application destroys information on which a user may depend: the lost task might cause the user to miss an appointment or deadline. The medical device can directly injure its user. The more substantial and direct the possible consequence, the stronger the reason to reduce uncertainty before delivering the system.

Causal distance matters because almost any software behavior can be connected to serious harm through a sufficiently long chain. A broken game may frustrate a player, and frustrated people sometimes behave badly. That possibility does not make an ordinary graphical defect safety-critical. Engineers need to reason about consequences that are sufficiently substantial and sufficiently connected to the software to matter to the delivery decision.

This does not imply that consequential systems must be perfect. Engineering rarely offers certainty, and useful systems can carry residual risk. Nor does consequence mechanically determine how much risk is acceptable. Two people can agree completely about what might happen and disagree about whether the available evidence justifies delivery.

Suppose, for example, that everyone agrees that a particular failure could kill people. The organization building the system may understand that consequence and nevertheless judge the residual risk acceptable because of the benefit the system provides. A regulator may permit the system under the same understanding. An engineer may still conclude that the available evidence is not strong enough to justify delivery.

The disagreement is not about what the consequence is. It is about what that consequence requires.

At least three things therefore contribute to the standard applied at delivery. Consequences establish what is at stake if the engineering judgment is wrong. Stakeholders judge what outcomes, costs, and risks they are willing to accept. Engineers exercise an independent professional judgment about what the available evidence allows them to stand behind. These judgments often agree, but none can simply be substituted for the others.

The last judgment takes us to professionalism. Engineers do not act only as employees carrying out the preferences of whoever controls a project. They also act as members of a profession.

DEFINITION · Professionalism

Professionalism is the exercise of specialized capability under obligations that are not reducible to the wishes of whoever asks you to use it.

A customer may be willing to accept a structural risk that an engineer judges inadequately understood. The customer's willingness does not require the engineer to certify the structure. The same distinction applies to software: another party may have authority

> to accept a risk without thereby determining what an engineer can responsibly claim about the system or requiring the engineer to participate in delivering it.

A profession exists partly because specialized decisions cannot always be made well by the people who want the resulting system. Society relies on physicians to exercise medical judgment rather than merely supply requested treatments, and on engineers to exercise engineering judgment rather than merely realize requested artifacts. Professional communities accumulate standards, methods, experience, and expectations about what their members should be prepared to stand behind. Individual engineers draw upon those accumulated judgments when they exercise professional authority.

Professional judgment is therefore different from personal preference. Suppose a customer needs a disposable internal tool by Friday. Its failures have minor consequences, and the customer can tolerate occasional errors. An engineer might prefer to spend another six months making the software extraordinarily reliable. That preference does not make the additional work good engineering. Time and money spent eliminating inconsequential uncertainty cannot be spent elsewhere.

The opposite disagreement is more serious. Suppose a system can kill people when it fails. The customer wants to proceed, but the engineer believes the available evidence does not justify delivery. The customer's willingness to bear the risk does not end the engineer's responsibility. The engineer may need to obtain stronger evidence, change the system, leave the decision to someone with the necessary competence, or refuse the work.

These cases expose two different pathologies. An engineer can demand substantially more assurance than the consequences and stakeholder needs warrant, producing perfection at a cost that does not justify it. Or an engineer can demand less assurance than the consequences warrant and become willing to deliver work that the engineer should not stand behind. Good engineering lies in neither direction. It seeks a defensible standard for the decision actually being made.

The relationship can be summarized as follows:

	Engineer accepts a lower standard	Engineer demands a higher standard
Consequences are substantial	Professional failure. The engineer's standard is inadequate to what is at stake.	Appropriate rigor, when the additional work materially reduces consequential uncertainty.
Consequences are minor	Often appropriate. Good enough may actually be good enough.	Overengineering, when additional assurance consumes resources without commensurate value.

This table intentionally simplifies the stakeholder's role. Stakeholders help determine what the software needs to accomplish and which ordinary failures they are willing to tolerate.

They may also participate in broader decisions about acceptable risk. The table isolates a different comparison: whether the engineer's own standard is appropriate to the consequences the engineer understands.

Professional authority matters most when those judgments diverge. A customer, employer, regulator, or government may be willing to accept a risk that an engineer is not willing to accept. Their willingness does not erase the engineer's responsibility for the engineering judgment. The engineer may refuse to certify the result and may ultimately refuse to use their professional capabilities to produce or deliver it.

This does not put engineers above society. Engineers do not acquire general authority over what purposes other people may pursue merely because those purposes involve technology. But political, organizational, and economic authority do not replace local engineering judgment either. Other people can make decisions within their authority. They cannot make an engineer possess evidence the engineer does not have, nor can they require the engineer to stand behind a judgment the engineer cannot defend.

Professional authority therefore includes the ability to say no.

Validation makes this responsibility especially visible because delivery requires a decision. Professional responsibility does not begin there. An engineer who recognizes a dangerous omission in the requirements cannot knowingly preserve it on the theory that validation will catch the problem later. The same is true of an indefensible specification, architectural decision, or design. Each engineering activity carries responsibility for the decisions made within it.

Validation is nevertheless special because earlier decisions meet evidence there. Requirements establish what the engineering effort promised. Specification bounds what would satisfy those promises. Architecture and design determine how one realization is organized and how its parts fulfill their responsibilities. Implementation gives engineers a realization they can inspect, exercise, measure, analyze, and sometimes deliver provisionally to learn from use. Evidence can reveal an implementation defect, but it can also expose an architectural weakness, a missing distinction in the specification, or a requirement that does not serve its purpose. This continues the book's existing principle that decisions flow downward while engineering learning moves upward (Design).

The distinctive question at validation is whether the evidence is enough to deliver.

Consequence changes the amount of evidence we should demand

A useful starting point is what could happen if the delivery decision is wrong. Suppose a TODO application loses one task in every thousand. Whether that behavior warrants delaying delivery depends partly on what the application is for. A personal scratchpad may tolerate the loss. A task system used to coordinate emergency maintenance may not. The same observed failure rate can justify different decisions because the consequences attached to the lost information differ.

DEFINITION · Consequence

> A consequence is an effect in the world that can result from an engineering decision or system behavior.
>
> An incorrect dose produced by a medical device can directly injure a patient. A lost TODO item can matter because a user later acts without information they expected the system to preserve. A graphical defect in a game may do little beyond frustrating the player. The effects differ in severity and in how directly the software produces them.

Consequences give engineers a reason to demand more or less assurance, but they still do not determine a unique threshold for delivery. Someone must judge what evidence is sufficient given what is at stake.

That judgment also changes as engineers learn. Before delivery, a harmful outcome may be a possibility supported by stronger or weaker evidence. Delivery itself can then produce information. Engineers can observe failures, measure outcomes, receive reports from users, and discover consequences they did not predict. This is one reason delivering under uncertainty can be valuable when the possible consequences are modest: use generates knowledge that might have been expensive or impossible to obtain beforehand.

But new evidence changes the engineering decision. Suppose engineers initially believe that a deployed feature is unlikely to cause substantial harm. Later evidence indicates that it is causing such harm. The justification for the original delivery cannot simply be reused. Continuing to deliver the same behavior is now a new decision made with different knowledge.

This distinction is especially important for software because updating is often possible. Engineers can respond to evidence by changing behavior across a deployed population quickly. That capability makes learning after delivery more practical, but it also removes one excuse for ignoring what has been learned. If engineers know that deployed behavior is causing substantial harm and can change it, continued operation requires a justification under that new state of knowledge.

Validation therefore does not end with the first delivery. Evidence continues to flow upward from the behavior of the deployed system, and consequential evidence should cause engineering decisions to be reconsidered.

The question remains the same even as the evidence changes: Do we know enough to justify what we are about to deliver?

The next step is to make *know enough* precise. Before choosing tests, analyses, reviews, or other validation techniques, engineers need to identify what they are actually trying to establish.

Claims and evidence

Before choosing a validation technique, identify the claim that needs support. Saying that a system has “been tested” tells us little until we know what the tests were intended to establish.

A payment service might need to support several different claims. A payment submitted once should not be charged twice. An unauthorized user should not be able to initiate a

payment. A completed payment should appear in the account history. A normal request should complete within an acceptable time. Each claim concerns the same system, but evidence that supports one may say little about another.

This is why validation techniques should not be taught as a catalog from which engineers select a sufficiently impressive collection. The engineering problem runs in the other direction. Engineers first identify what they need to believe, then consider how the system could violate that claim, and finally seek evidence capable of distinguishing the acceptable behaviors from the unacceptable ones.

If the claim concerns duplicate charging, engineers need evidence capable of exposing duplicate execution under the conditions in which it might occur. If the claim concerns latency, they need measurements under relevant workloads and environments. If the claim concerns behavior over all possible values of some bounded input, exhaustive analysis or proof may provide evidence that ordinary examples cannot.

The method follows from what engineers need to know.

A claim is distinct both from the artifact it concerns and from the technique that supports it. The same artifact can carry many claims, and the same technique can support claims of quite different kinds. Requirements and specification supply the obvious claims: requirements state what the engineering effort promised, and specification bounds the behaviors that would count as keeping those promises. But architecture and design create claims of their own. An architecture that promises failure isolation is claiming that a fault in one part cannot corrupt another. A design that defers work to a queue rests on the claim that the same operation can safely execute twice (Design). These claims rarely appear in any requirements document, yet the system depends on them, and some of the most expensive failures violate an assumption that nobody thought to state as a claim worth checking.

Evidence, in turn, is always evidence *for* something, under assumptions. The assumptions should be made explicit, because they bound what the evidence can mean. A proof can strongly establish a stated property relative to its model while saying nothing about a requirement omitted from that model. A load test establishes latency under the workload it generated, not under the workload users will generate. Treating "we have strong evidence" as a property of the system, rather than of particular claims under particular assumptions, is how validation programs mislead the people who rely on them.

The world, machine, and domain-assumption distinction from Specification returns here with new force. Most validation evidence attaches to the machine: tests, analyses, and proofs establish claims about behavior at the machine's boundary. But requirements live in the world, and the connection between them runs through domain assumptions (Zave and Jackson 1997). A system can satisfy its specification perfectly while its requirement fails because a domain assumption was false. No amount of machine-side evidence can expose that failure, because the machine is behaving exactly as specified. Evidence about the world — observation of the deployed system operating in its actual environment — can. This is one reason operational evidence, taken up at the end of this chapter, is a constituent of validation rather than an afterthought.

Choosing among sources of evidence

Once the claim is explicit, validation techniques become alternatives for producing evidence about it. The useful question about each is its mechanism: what kind of uncertainty can it reduce, which failures can it expose or exclude, and what remains outside its reach.

- **Review** brings another engineer's knowledge and judgment to an artifact. It can expose reasoning errors the author cannot see and consequences the author did not consider. Nothing in the mechanism forces either person to consider a behavior neither imagined.
- **Example-based testing** exercises selected behaviors against expected results. It is direct and cheap when the important cases and their expected outcomes are known, and silent about every case not selected.
- **Property-based testing** states a property that should hold across a class of inputs and searches that class for counterexamples. The engineer trades hand-picking examples for the harder work of stating what should be true in general.
- **Differential testing** compares independently developed implementations on the same inputs. Disagreement flags something worth investigating, which makes it valuable exactly when the correct answer is expensive to state case by case.
- **Fuzzing** explores large or unusual input spaces to find behaviors the engineer did not anticipate. It needs only a weak notion of failure — a crash, a hang — so it exposes robustness failures cheaply while saying little about functional correctness.
- **Static analysis and model checking** reason about possible behaviors without producing each one through execution. They can *exclude* whole classes of failure — something no finite set of executions can do — but only over the model or abstraction they analyze.
- **Measurement and experimentation** establish quantitative claims: whether a latency budget holds under a workload, whether users complete a task, whether one variant outperforms another. The evidence is only as good as the workload's or experiment's resemblance to reality.
- **Operational observation** watches the deployed system itself, in the one environment no earlier technique can fully reproduce. Its evidence arrives after the consequences have begun.

Several of these techniques answer the same underlying difficulty, the *oracle problem*: engineers can often generate inputs far more cheaply than they can state the correct output for each one. Property-based testing responds by stating the expected answer as a property. Differential testing lets an independent implementation flag the suspicious case. Model checking states the property over an explicit behavioral model and searches the state space for a violation. Recognizing the shared problem matters more than memorizing the techniques, because the next technique an engineer encounters will be answering it too.

The strength of evidence

No technique supplies confidence by its name alone. A model checker can exhaustively verify a property of the model while leaving an incorrect model untouched. Thousands of generated tests can explore an input space while sharing the same mistaken oracle. A code review can bring independent judgment while still missing behavior that neither reviewer

considered. Evidence must be evaluated relative to the claim it supports and the assumptions on which it depends.

Evaluating a body of evidence is itself an engineering judgment, and a few questions structure it. What portion of the claim's space did the evidence actually exercise, and what portion did it never touch? Do the exercised conditions resemble the conditions of delivery, or a convenient laboratory version of them? Do the pieces of evidence rest on distinct assumptions, or do they all inherit the same one? Where a technique claims soundness or completeness, what exactly do its verdicts guarantee, and over what model? What uncertainty remains after all of it, and is that residual acceptable for this decision? Strong validation for consequential claims tends toward triangulation: multiple forms of evidence whose failure modes are uncorrelated, so that a mistaken assumption in one is caught by another.

Independence deserves particular attention, because volume imitates it well.

> **KEY-IDEA · Evidence multiplies; independence does not**
> A thousand tests generated from the same mistaken interpretation of a requirement are not a thousand independent reasons to believe the interpretation is correct. They are one reason, repeated. The strength of a body of evidence grows with the diversity of the assumptions it rests on, not with its count.

This point has become more important, not less, as generation has become cheap. When tests were expensive to write, each one embodied a deliberate act of engineering attention, and a large suite loosely signaled substantial scrutiny. When a tool can generate implementations and thousands of passing tests from the same prompt, the tests and the implementation can share a single mistaken interpretation, and the suite's size signals nothing about it. Software Engineering argued that as implementation becomes abundant, the judgments surrounding it become relatively more important. The same shift applies inside validation: cheap evidence generation makes *judging* evidence — its coverage, its representativeness, its independence, its assumptions — the scarce engineering work.

Deciding when to stop

Additional evidence always has a cost: the effort of producing it, the attention of evaluating it, and the delay it imposes on whatever value delivery would create. Process observed that delivering an increment can itself generate information. Withholding delivery to buy more evidence therefore has a price on both sides of the ledger, and the mismatch table from earlier in this chapter applies to it directly. For a low-consequence system, delivering with unresolved uncertainty can be the defensible choice, and demanding near-certainty is the pathology: overengineering, resources consumed without commensurate value. For a high-consequence system, the same unresolved uncertainty may justify expensive and diverse evidence, or it may mean that delivery cannot currently be defended at all.

The decision at the end of validation is also not binary. Evidence that fails to justify delivery does not merely say "stop"; it usually says something about where the problem lies, and the response should go there.

DECISION · The delivery decision

Deliver, when the evidence supports the claims that matter at a standard defensible for the consequences. Gather more evidence, when the shortfall is knowledge and the evidence is worth its cost. Change the system, when the evidence has exposed behavior that should not be delivered. Revisit an upstream decision, when the evidence indicts a requirement, specification, architecture, or design rather than the implementation. Refuse, when the consequences demand a standard the available evidence cannot meet and no party can make that gap disappear by accepting it.

The last option needs no new argument; it is the professional authority of this chapter's opening applied at the moment it matters. Throughout, the three inputs remain distinct. Consequence establishes what is at stake. Stakeholders judge which outcomes, costs, and risks they will accept. The engineer judges what the evidence allows them to stand behind. A delivery decision is defensible when it can answer to all three, and the disagreements worth having are usually about the third: everyone can agree on what a failure would do and still disagree about what evidence its possibility demands.

Evidence after delivery

Delivery moves the system into the one environment no earlier validation could fully reproduce, and the environment repays the favor with evidence. Monitoring reveals behavior under real workloads. Incidents expose failures no one selected as a test case. Field measurements and experiments quantify what analysis could only estimate. User reports carry consequences engineers did not predict. And the world itself changes: a domain assumption that was true at delivery can quietly become false as usage, populations, and conditions shift. Operational observation is therefore not a courtesy that follows validation. It is validation continuing under better evidence.

The crucial judgment is that the decision must update when the evidence updates. It is one thing to deliver amid uncertainty about a possible harm; the uncertainty was weighed, and the decision may have been sound. It is a different thing to continue delivering the same behavior after evidence of substantial harm has accumulated. The original justification cannot be reused, because it was a justification under a state of knowledge that no longer exists.

Recent litigation against social-media companies illustrates the distinction, if it is read carefully. Plaintiffs — including states and school districts — have alleged that platform design features harmed young users, and that the companies continued delivering those features after internal evidence of harm had accumulated. The careful reading matters because legal artifacts establish different things: an allegation is a claim awaiting test, a verdict resolves a legal question under a legal standard, and a settlement may reflect a risk calculation rather than a factual concession. None of them is scientific proof of causation, and this chapter takes no position on the underlying social-science debate. The engineering lesson does not depend on it. Whatever the eventual adjudication, the shape of the accusa-

tion is exactly the failure this section names: not *you delivered under uncertainty*, but *your evidence changed and your decision did not*.

Software's medium gives this duty its particular force. Software Engineering observed that copyability and updateability cut both ways: the same mechanism that propagates a repair across millions of instances propagates a defect across them too. Validation supplies the corresponding judgment. Updateability is what makes learning after delivery a legitimate strategy when failures are tolerable — the repair really can reach everyone cheaply. And updateability is what makes inaction hard to defend when failures are not tolerable, because engineers of deployed software hold an unusual power: once evidence shows that deployed behavior should change, they can change it, everywhere, quickly. An engineer who could respond and does not has made a decision, and it is a decision that must be justified under the new state of knowledge.

Summary

Validation asks what evidence is sufficient to deliver a software system into the world. The question is broader than testing, and it does not demand certainty: software's updateability makes delivering under known uncertainty and learning from use a legitimate strategy, but only when discovering a failure after delivery is an acceptable way to learn, because an update repairs the software, not the consequences of its previous behavior.

Three inputs shape the standard applied at delivery, and none substitutes for the others. Consequences establish what is at stake, weighed by severity and by causal distance. Stakeholders judge which outcomes, costs, and risks they will accept. Engineers exercise an independent professional judgment about what the available evidence allows them to stand behind — a judgment that can fail in either direction, by demanding more assurance than minor consequences warrant or less than substantial ones require. People can agree entirely about a consequence and still disagree about what evidence it demands.

The evidence itself starts from claims, not techniques. Engineers identify what they need to believe — including the claims and assumptions created by architecture and design, not only those written in requirements — then choose among sources of evidence by mechanism: what uncertainty each can reduce and what remains outside its reach. The strength of the result is judged by coverage, representativeness, independence, and assumptions rather than volume; a thousand tests sharing one mistaken interpretation are one reason, repeated, and cheap evidence generation has made judging evidence the scarce work. The decision that follows is not binary — deliver, gather more evidence, change the system, revisit an upstream decision, or refuse — and it recurs after delivery, because operational evidence keeps arriving and a justification made under an old state of knowledge cannot be reused under a new one.

READ FURTHER

Winters, Titus, Tom Manshreck, and Hyrum Wright, eds. *Software Engineering at Google: Lessons Learned from Programming Over Time*. Sebastopol, CA: O'Reilly Media, 2020. An account of how engineers obtain evidence at different scopes, from review through unit tests to larger-scale testing. Read the "Code Review" chapter (chap. 9) and the testing chapters (chaps. 11–14).

Cockx, Jesper. "An Introduction to Property-Based Testing with QuickCheck" (2020). Introduces property-based testing: state properties that should hold across a class of inputs, then generate inputs in search of counterexamples. The Haskell syntax is incidental; the validation strategy is not. For the technique's origin, see Claessen and Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs" (ICFP 2000); for a contemporary application in an agentic setting, see Anthropic, "Finding bugs across the Python ecosystem with Claude and property-based testing" (2026).

McKeeman, William M. "Differential Testing for Software." *Digital Technical Journal* 10, no. 1 (1998): 100–107. Introduces differential testing: independently developed implementations serve as partial oracles for one another.

Palshikar, Girish Keshav. "An Introduction to Model Checking." *Embedded Systems Programming*, February 2004. An accessible introduction to model checking and the evidence obtained by exhaustively checking a model.

CHAPTER 10

Research and Development

Premise. *Research and development invests scarce engineering effort in uncertain opportunities for asymmetric benefit.*

DEFINITION · Research and development

Research and development (R&D) is systematic work undertaken to increase knowledge and develop new applications of knowledge (OECD 2015). The knowledge sought may concern what is possible, how something works, or how an existing limitation can be overcome.

R&D begins with a constraint: there are more worthwhile problems than there is time to pursue them. A researcher may see opportunities to improve performance, remove a limitation, support a new use, exploit a new capability, or understand a phenomenon more deeply. An industrial R&D group faces the same abundance of possibilities. Pursuing one means not pursuing another.

The unusual feature of R&D is that the outcomes can be asymmetric. An investigation may fail and produce little beyond what was learned, while a successful new idea can enable capabilities, improvements, or understanding far beyond the effort invested. Most research projects can fail without doing much harm; a successful one can change what becomes possible.

The preceding chapters followed a purpose into a system: engineers decide what to promise (Requirements), make those promises precise (Specification), organize a realization (Architecture) and resolve the choices inside its parts (Design), and decide what evidence justifies delivery (Validation). R&D asks where engineering effort should go when we do not yet know what is possible. Before asking how to solve a problem, we must decide whether the problem deserves the effort. The answer depends partly on what success could accomplish and partly on how likely success appears. A modest improvement with an obvious solution may be nearly certain. A larger advance may require an idea we do not yet have.

There is no formula that resolves this tradeoff. R&D exists precisely because important facts are unknown. But four questions organize the judgment.

KEY-IDEA · The four questions of R&D

- **What is worth pursuing?** Decide which uncertain opportunity deserves scarce engineering effort: seek asymmetric upside under opportunity cost.
- **Does it require a new idea?** Determine whether the limitation in what exists is essential to the approach or merely a property of its current embodiment.
- **How large is the advance?** Judge what would be different if the work succeeds: an increment, an advance, or occasionally a transformation.
- **What would establish it?** Determine what evidence would support the claimed advance.

The first two concern choosing and perceiving which opportunity deserves your attention and whether it actually requires a new idea. The latter two are judgments about understanding what you have actually accomplished. The chapter takes them in turn.

What is worth pursuing?

A useful R&D problem offers the possibility of an outcome worth the effort required to investigate it. You must weigh two factors:

- **Potential impact.** If this works, how large is the upside? An incremental improvement may be useful; an advance may change what can be done at all.
- **Likelihood of success.** How plausible is it that the investigation will produce something useful?

Cost, timing, available expertise, competition, and dependencies also matter. The point is not to assign each factor a score and select the largest number. It is to compare opportunities explicitly rather than allowing the easiest visible project to consume the time available.

R&D is easily distorted by its visible outputs. A patent, paper, prototype, or product can become the objective rather than evidence that useful knowledge was created. Organizations face the same temptation when they subordinate research to immediate product needs. The work becomes easier to justify in the short term, but less able to discover ideas that would change what the organization can do.

The object of R&D is knowledge. Pursue an important question whose answer is not yet known and whose answer could matter. Papers and patents are valuable ways to communicate, protect, and establish what was learned. Products may eventually put that knowledge to work. They are outcomes of a successful R&D agenda, not substitutes for one.

Here, hard does not mean technically difficult for its own sake. It means the most potentially transformative — for the discipline, the organization, or for you; any and all.

My advice is to choose the hardest important problem you can plausibly make progress on. You will sometimes fail. That is part of accepting the asymmetry. You are giving up some probability of producing a result in exchange for the possibility of producing a much larger one. You will also find that difficult problems change what you are capable of doing: they force you to learn more deeply, acquire techniques you would not otherwise need, and encounter questions you could not have seen from the easier problem. When the hard problem succeeds, it has more room to matter. Over a research career, I would rather see you take serious swings than become very efficient at producing results you already knew how to obtain.

This advice deliberately does not maximize the probability of producing a result. Research can tolerate failed attempts in a way ordinary delivery work often cannot. The relevant question is not simply *How likely am I to produce something?* It is *What might become possible if this works, and what will I learn by seriously attempting it?*

Does it require a new idea?

Finding an important problem does not establish that a new approach is needed. Existing systems are imperfect in innumerable ways. They may be slow, support the wrong formats,

expose awkward interfaces, lack useful features, or have implementations that were never optimized for the situation now under consideration.

Any of these limitations can motivate engineering work. They do not necessarily motivate R&D. The useful distinction is between essence and accident.

DEFINITION · Essential and accidental properties

An **essential** property follows from the underlying concept: an embodiment that lacked the property would no longer faithfully realize the same idea. An **accidental** property belongs to a particular embodiment but need not belong to another embodiment of the same idea. The distinction between essential and accidental properties goes back at least to Aristotle; here we apply it to engineered systems and approaches.

Consider a parser. Parsers accept input and determine whether it conforms to some language or format; that role is essential to the concept. A particular parser might accept JSON but not XML, use one parsing algorithm rather than another, or expose a particular API. Those properties may simply characterize that implementation. Another parser can differ in all of them while remaining recognizably a parser.

This distinction matters because a claimed advance often attacks the embodiment when it needs to attack the idea. Suppose an existing system lacks a capability your proposed system provides. If the existing system could acquire that capability through an ordinary extension while retaining the same conceptual approach, you have identified an engineering task, not necessarily an advance. A long list of deficiencies does not change that fact if every deficiency can be repaired without changing the idea.

The stronger question is therefore: *Could a faithful embodiment of the existing idea avoid this limitation?*

If yes, the limitation is probably accidental. Improve the embodiment.

If no — if the limitation follows from an assumption, abstraction, algorithm, representation, or other defining property of the approach — then eliminating it requires a conceptual change. That is where the opportunity for an advance begins.

Can this be solved with elbow grease?

There is a practical way to develop intuition for this distinction. When you believe you have found a limitation worth researching, ask whether it can be solved with **elbow grease**.

Imagine giving the existing system and the limitation to two competent junior engineers for six months. They can write code, optimize components, replace libraries, support additional formats, collect more data, improve deployment, and clean up awkward parts of the implementation. If you expect sustained engineering effort to make the limitation disappear while leaving the underlying idea intact, be suspicious of the claimed advance. You may have found substantial work without finding a substantial advance.

If the limitation survives that thought experiment, look deeper. Perhaps every faithful realization of the approach inherits it. Removing the limitation might require changing an assumption about the problem, introducing a different representation, replacing the governing algorithm, moving a boundary, or otherwise altering something integral to the idea. Now you may have found the opening for R&D.

Elbow grease is a diagnostic, not a definition. An accidental limitation can be expensive to remove, especially in a large or poorly engineered system. Conversely, a deep conceptual change can occasionally be easy to implement once somebody sees it. The definitive question remains whether eliminating the limitation requires changing the idea.

Generative AI also changes the elbow-grease test because it changes the price of engineering effort (Software Engineering). "Two junior engineers for six months" once represented a substantial amount of implementation capacity. Increasingly, some of that work can be accomplished by a much smaller team working with capable agents in days or weeks. Yesterday's impressive implementation effort can become tomorrow's routine engineering task.

This raises rather than lowers the importance of identifying the conceptual advance. As implementation becomes cheaper, difficulty that comes primarily from producing implementation becomes weaker evidence of an advance. The underlying test has not changed: can additional engineering effort remove the limitation without changing the idea? Commodity intelligence simply makes accidental limitations cheaper to eliminate and therefore harder to defend as the basis for R&D.

How large is the advance?

R&D can produce improvements of very different magnitude. An **increment** improves an existing approach while preserving its essential idea. An **advance** changes something essential and thereby makes something meaningfully different possible. Occasionally, an advance is **transformative**: it changes the space of problems people can solve or the way a field approaches them.

The vocabulary follows from the essential–accidental distinction. Work that removes accidental limitations produces increments: a faster implementation, a broader format, a cleaner interface — valuable, often necessary, and conceptually conservative. Work that removes an essential limitation produces an advance, because every faithful embodiment of the old idea inherited the limitation and the new idea does not. Transformation is rarer still, and mostly recognized in retrospect; few projects should be judged by whether they achieve it.

This vocabulary also disciplines the potential-impact judgment made when choosing what to pursue. When you claim an opportunity has a large upside, say which magnitude you mean. An increment can be worth pursuing when the approach it improves is important enough. But the asymmetric upside that justifies a hard, uncertain investigation usually lives at the advance level: the work does not merely improve what exists, it changes what can be done.

What would establish the advance?

Once an opportunity appears worth pursuing and requires a genuine advance, the remaining question is what would establish that advance. Four pieces must hold together:

- **Problem.** Identify the important limitation or opportunity the work addresses.

- **Essential limitation.** Show why additional engineering of the existing approach cannot adequately resolve it.
- **New idea.** State what changes in the underlying concept rather than merely in its embodiment.
- **Evidence.** Demonstrate that the new idea produces the claimed improvement and establish the conditions under which the claim holds.

Weak R&D often leaves one of these links unsupported. An important problem does not establish that a new idea was required. A new idea does not establish that it works. A successful implementation does not establish that the improvement came from the claimed idea. The evidence must support the particular advance being claimed.

R&D is a bet on learning

R&D begins before we know whether the proposed idea will work. That uncertainty is not a defect in the process; it is why the work is R&D.

The judgment is therefore not merely whether an idea sounds promising. You are deciding which uncertainty deserves your scarce time. Potential impact matters because success should justify the investment. Likelihood of success matters because some bets are implausible even when their imagined payoff is enormous. The need for a new idea matters because an apparent R&D problem that can be eliminated through ordinary engineering may not justify inventing a new approach at all.

Then the work itself produces information. A failed prototype may reveal that an assumed opportunity does not exist. An experiment may show that the supposed limitation of prior work was accidental. An attempted solution may expose a deeper problem than the one that motivated it. A failed research direction can therefore be valuable — but only if you recognize what it taught you and allow that evidence to change what you work on next.

You will spend a career making these bets with incomplete information. Choose them deliberately. Ask what could matter, whether the obstacle is real, whether it requires a new idea, and what evidence would tell you that you were right.

And when you have a choice between an easy problem whose answer you can already see and a hard important problem that might defeat you, I recommend the hard one.

Summary

Research and development invests scarce engineering effort in uncertain opportunities for asymmetric benefit. There are more worthwhile problems than time to pursue them, and the outcomes are asymmetric: an investigation may fail and produce little beyond what was learned, while a successful new idea can return far more than the effort invested. Four questions organize the judgment.

What is worth pursuing? weighs potential impact against likelihood of success, comparing opportunities explicitly instead of letting the easiest visible project consume the time — and, for a researcher, argues for the hardest important problem you can plausibly make progress on. *Does it require a new idea?* separates essential limitations, which follow from the underlying concept, from accidental limitations of a particular embodiment. The

elbow-grease thought experiment supplies the diagnostic, and generative AI sharpens it: as implementation becomes cheaper, accidental limitations become weaker evidence of an advance. *How large is the advance?* calibrates the upside: an increment preserves the essential idea, an advance changes something essential, and a transformative advance changes what a field can attempt at all. *What would establish it?* requires four pieces to hold together — problem, essential limitation, new idea, and evidence that supports the particular advance being claimed. Throughout, the object of R&D is knowledge: papers, patents, and products are outcomes of a successful agenda, not substitutes for one.

The bet is ultimately on learning. Even a failed direction pays, if you recognize what it taught you and let that evidence change what you pursue next.

READ FURTHER

Robertson Ishii, Teresa, and Philip Atkins. "Essential vs. Accidental Properties." In *The Stanford Encyclopedia of Philosophy*. The ancient distinction between what belongs to a thing as the kind of thing it is and what merely belongs to one realization of it.

Hamming, Richard W. "You and Your Research." Talk at Bellcore, 1986. A classic talk about choosing important research problems. Some of the advice has aged; the central question has not.

OECD. *Frascati Manual 2015: Guidelines for Collecting and Reporting Data on Research and Experimental Development*. Paris: OECD Publishing, 2015. The formal definition of R&D, from the manual used to decide what counts as research across countries and industries.

Bush, Vannevar. *Science, the Endless Frontier*. Washington, DC: United States Government Printing Office, 1945. The report that argued for funding research whose object is knowledge and trusting applications to follow.

Conclusion

Software engineers make the decisions in this book because software has particular properties as an engineered medium.

Software is unusually easy to change. We can build and test partial systems. We can copy an implementation at almost no cost. We can create abstractions that hide enormous amounts of detail. We can automate tests and analyses. Increasingly, we can ask an agent to produce an implementation in minutes.

These properties give us choices. We can organize work in different ways. We can choose what to specify and what to leave open. We can divide a system along different boundaries. We can choose among designs with different costs and consequences. We can build first to learn, or reason first because changing our minds later will be expensive. Those choices are why software engineering requires judgment.

Making an engineering decision

When you face an engineering task, begin with a simple question: what is required?

Before choosing a solution, understand what the solution must accomplish. Identify the requirements, constraints, assumptions, and existing decisions that bear on the problem. Determine what is fixed and what remains open.

Then ask: what do I need to know to make this decision?

Perhaps you need to understand how users actually work. Perhaps you need to know how expensive an architectural choice will be to reverse. You may need an estimate, a model, a measurement, a prototype, or an experiment. You may need to read the code. You may need to talk to the person who operates the system every day. Gather the information that could change your decision.

But there is a harder problem. You may not know what you need to know.

The requirement you were given may omit an important stakeholder. The strange piece of code you want to remove may exist for a reason. Someone else may understand a failure mode that has never occurred to you. Your experience may simply not cover the situation you now face.

No technique in this handbook makes that problem disappear. You have to look for gaps in your own understanding. Ask questions. Seek out people who know things you do not. Listen when someone disagrees with you. Treat an unexplained constraint as something to investigate before treating it as a mistake. This is part of engineering judgment too.

Eventually, you have to decide. Engineering rarely provides perfect information, and obtaining more information has a cost of its own. Sometimes the right choice is to investigate further. Sometimes it is to build a prototype and learn from it. Sometimes the evidence is already sufficient and further analysis would add little.

The goal is to know enough to make the decision well, while remaining aware of what you do not know.

DECISION · Before you decide

What is required? What must the result accomplish or preserve?

What do I need to know? What information could change the decision?

What might I be missing? Whose knowledge, experience, or perspective could reveal something I have overlooked?

How should I find out? Ask, inspect, model, measure, prototype, experiment, or build.

Then decide. Act when you have enough information, and take responsibility for the choice.

Requirements elicitation, specification, process models, architectural views, design alternatives, prototypes, tests, reviews, and measurements all help you answer these questions. They are not recipes that remove the need for judgment. They help you exercise it.

That is also why this book cannot finish your apprenticeship. It can show you the questions software engineers have learned to ask and the tools they use to answer them. Experience will teach you when to ask which question, how much evidence is enough, when something feels wrong, and when you need to find someone who knows more than you do.

Welcome to software engineering.