

Requirements

The Software Engineering Handbook

A Judgment and Decision-Making Approach

James C. Davis

Chapter 5 of the full book · Edition 1 · 2026

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).

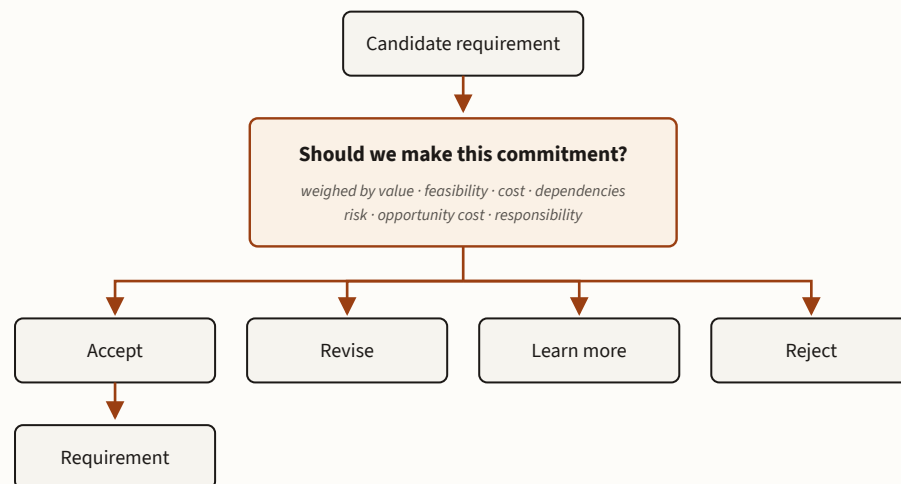


Figure 3. The commitment decision. A candidate requirement is weighed and resolves to one of four peer outcomes; only accept yields a requirement.

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: $\text{Cost} \approx \text{Size} \times \text{Cost per unit} \times \text{Cost drivers}$. 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 $\neq$ budget $\neq$ target $\neq$ 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.

Measurement for decision-making

A requirement is a judgment about an outcome that matters in the world. Before committing to it, engineers should ask what observations would later supply evidence that the promised outcome was actually achieved. An outcome nobody could recognize as achieved is an outcome nobody can be held to.

Some requirements connect readily to observable quantities. Response time, energy consumption, and transaction capacity suggest direct measurements. Others require that a measurement model be built first. "Easy to use" might be investigated through a usability study: recruit participants representative of the intended users, give them representative tasks, and record task-completion times, error rates, and the points at which participants hesitate or ask for help. A standardized usability questionnaire administered afterward supplies a second, partly independent view. "Maintainable" might be investigated through the time and the number of components touched when engineers perform representative maintenance tasks. In each case the measurement does not replace the requirement. It supplies evidence about one aspect of the outcome the requirement represents, obtained under the conditions that particular study created.

Some important requirements resist complete measurement. Safety and security are awkward precisely because observing no failures does not establish that none remain possible. Engineers can count incidents, vulnerabilities found, resistance under a particular

set of attacks, safety margins, hazardous events, or conformance with specified controls. None of those observations measures how safe or how secure the system is in general. Such requirements need several forms of evidence at once, and engineers must stay explicit about what each form establishes and what it leaves open.

Requirements also differ in criticality. Failing to satisfy one may create inconvenience; failing to satisfy another may threaten safety, security, mission success, or substantial economic value. Criticality need not form a simple total ranking. It can depend on operating context, groups of obligations may carry their own internal priorities, and some consequences resist direct comparison. Making these judgments explicit while the commitment is being made lets later engineering allocate design attention and validation evidence according to consequence (Validation).

Thinking about measurement during requirements work therefore serves two purposes. It tests whether the desired outcome has been stated clearly enough to recognize evidence about success. And it makes engineers confront the limits of that evidence before a convenient proxy quietly becomes the requirement itself.

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.” Special issue, *Proceedings of the Conference on the Future of Software Engineering* (New York), 2000, 35–46. <https://doi.org/10.1145/336512.336523>. A concise map of requirements engineering: elicitation, modeling, analysis, negotiation, and evolution.
- Patton, Jeff. “The New User Story Backlog Is a Map.” 2008. <https://www.jpattonassociates.com/the-new-backlog/>. 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 “Proceedings of the 27th IEEE International Requirements Engineering Conference (RE 2019).” Special issue, *Proceedings of the 27th IEEE International Requirements Engineering Conference (RE 2019)*, 2019, 340–50. <https://doi.org/10.1109/RE.2019.00043>. An industrial case study of what a customer’s initial requirements list failed to capture.