

Teamwork

The Software Engineering Handbook

A Judgment and Decision-Making Approach

James C. Davis

Chapter 3 of the full book · Edition 1 · 2026

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.

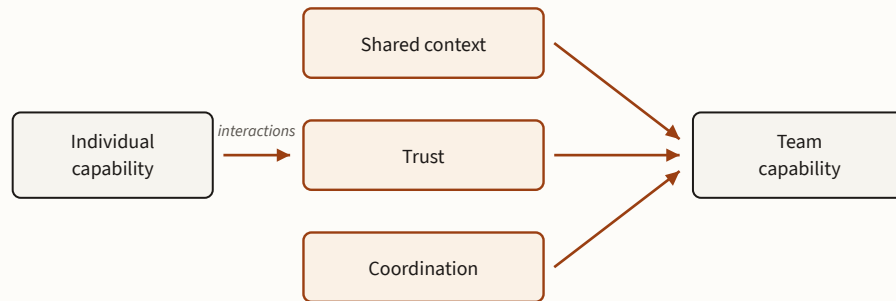


Figure 1. Individual capability becomes team capability only through interactions that build shared context, trust, and coordination. These team-level properties are what a team's engineering output actually flows through.

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.

Table 1. Strategies for engineering coordination.

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

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 organization does not need to recruit, mentor, retain, and develop each new instance of model intelligence. Generative AI therefore introduces something unusual into engineering orga-

nizations: 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 experi-

ence 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.

Measurement for decision-making

A team structure is an engineering hypothesis about how individual capabilities and coordination mechanisms will combine into collective capability. Observation can help engineers determine whether that hypothesis is working.

Coordination leaves evidence in the systems through which engineers work. Email, chat systems such as Teams or Slack, issue trackers, code review, and version-control history record requests, responses, handoffs, dependencies, and decisions. Engineers can use those records to examine how long consequential questions remain unanswered,

how long work sits blocked on another person or team, how frequently work crosses an organizational boundary, where reviews repeatedly wait on the same few people, how often integration exposes incompatible assumptions, and which parts of the system generate unusually broad coordination.

Extracting those relationships at scale once required substantial manual analysis or specialized tooling, which is why most organizations reasoned about coordination from the organization chart instead. Language models make another approach practical: they can classify and connect communication records, issues, reviews, and changes to reconstruct patterns of coordination for engineers to inspect. The result is still a model. A language model can misclassify an interaction, and recorded communication omits the informal coordination that happens in hallways and side conversations. But it makes the coordination the work actually demanded visible in a way an organization chart cannot, because the chart records intended structure while the records show what engineers had to do.

Care is especially important when measurements concern people. Commit counts, lines of code, tickets closed, messages sent, and similar activity measures are poor substitutes for engineering capability or contribution. Once treated as targets, they distort the behavior they purport to measure. The property of interest is usually not individual activity, but whether the team possesses and coordinates the capabilities required to make and realize sound engineering decisions.

Measure the coordination system, therefore, rather than reducing the people within it to scores. The purpose is to discover where the team's model of responsibilities, dependencies, and communication differs from what the work actually requires.

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).” Special issue, *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2015, 700–710. <https://doi.org/10.1109/ICSE.2015.335>. An interview study of why experienced engineers rate judgment, system understanding, and enabling colleagues alongside implementation skill.
- Brooks, Frederick P. *The Mythical Man-Month: Essays on Software Engineering*. Anniversary. 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 Engineering*. 1st ed. 2026. <https://davisjam.github.io/model-based-agentic-software-engineering/book/mage-book/index.html>, Part 7, “The Profession,” §§7.1 and 7.4. Examines where scarce engineering effort moves when implementation is abundant, and how junior engineers will still develop judgment.
- DORA. “Software Delivery Metrics: The Four Keys.” Accessed September 25, 2026. <https://dora.dev/guides/dora-metrics/>. An industry measurement program whose four indicators — deployment frequency, change lead time, change failure rate, and recovery time — characterize how a team’s delivery behaves rather than how much any individual produces.