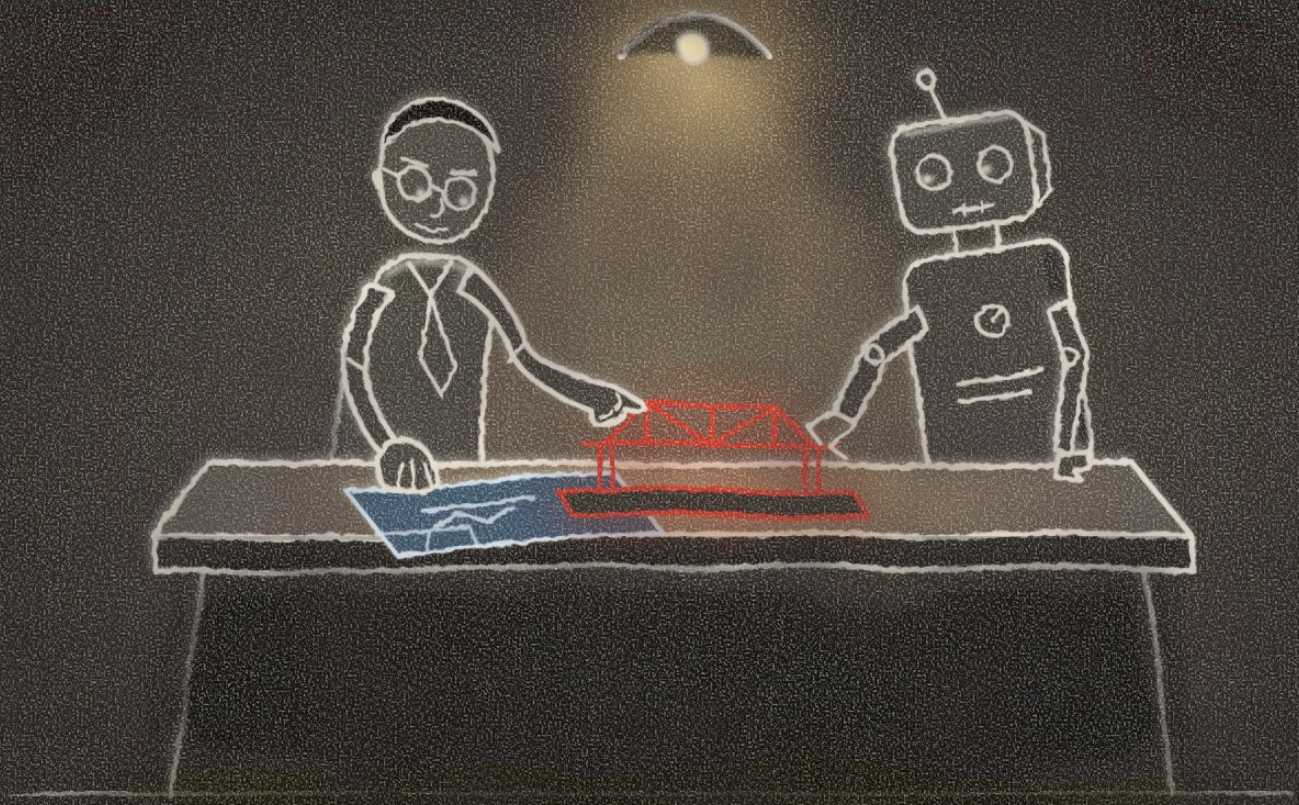
THE MAGE METHOD

Model-Based Agentic Software Engineering

JAMES C. DAVIS



Edition — first published 2026-07-22 · last modified 2026-09-06

Acknowledgments

I thank O. Adeosun, P. Amusuo, B. Çakar, R. Calvo, A. Cha, A. Chlipala, H. Davis, K. Davis, N. Davis, R.M. Davis, R.W. Davis, W. Davis, N. Eliopoulos, S. France, D. Inouye, P. Jajal, K. Kalu, A. Kazerouni, D. Koller, J. Laredo, H. Peng, N. Pond, E. Snible, G.K. Thiruvathukal, and E. Wittern for their contributions, conversations, and help in making this book.

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

Table of Contents

How to Read This Book

This Part is reference material. You do not need to understand it before beginning the book.

First time here? Read the Preface and continue into Part I. You may skip the rest of Part 0 for now; it is designed to become more useful as the terminology and argument become familiar.

Returning to the book? Use *What This Book Argues* for the six central claims, *MAGE on One Page* for the relationship among the major concepts, and the *Glossary* for definitions.

The Preface asks one question: *How do we safely grant autonomy to commodity intelligence?* The book builds toward the answer provided by MAGE. The Conclusion returns to the question and asks its converse: *What cannot be delegated?* Figure 0.1-1 traces that argument.

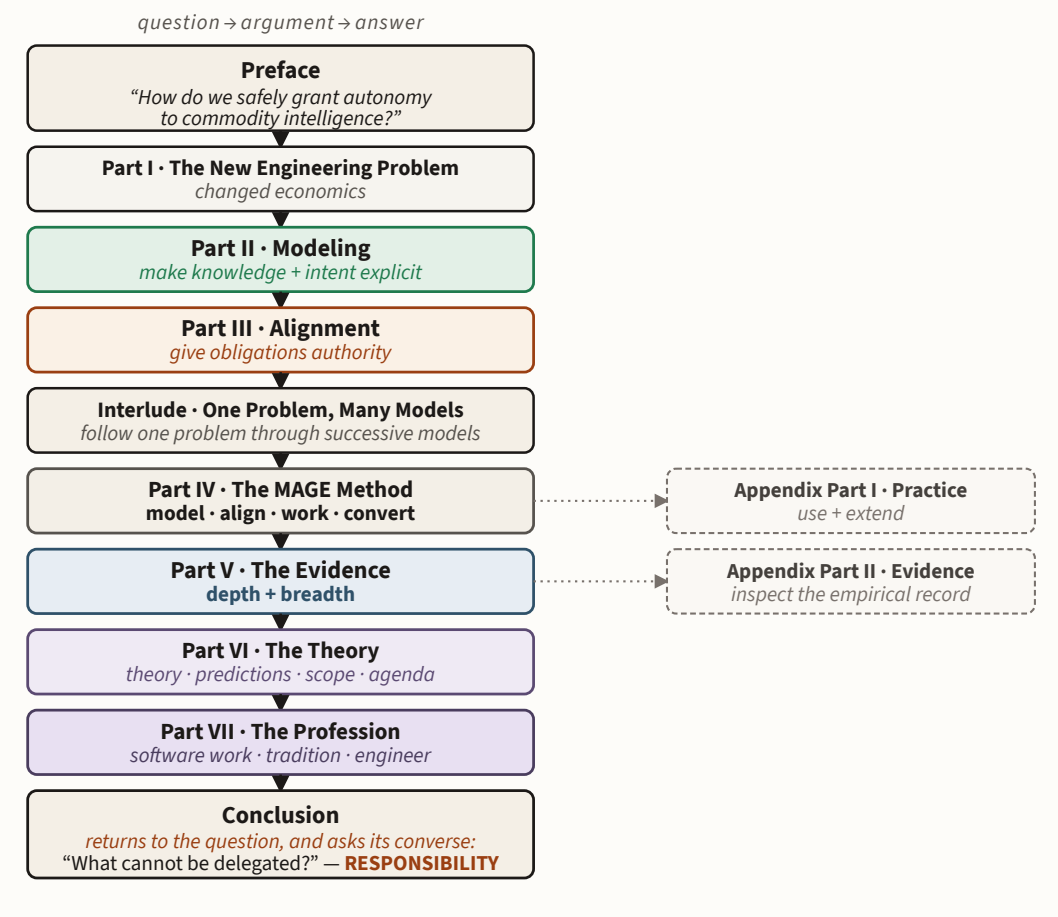


Figure 0.1-1: The book's argument, from the Preface's opening question through the seven Parts to the Conclusion's answer and its converse.

Three shorter routes serve different purposes:

- **Do it Monday.** Part I → Part IV → Appendix Part I: Practice. Start with the premise, move directly to the working method, then use the practice appendices as a reference while applying it.
- **Show me the data.** Part I → Part V → Appendix Part II: Evidence. Read the framing, examine the empirical argument, then inspect its evidentiary basis.
- **I want the theory.** Part I → Part II → Part III → Interlude: One Problem, Many Models → Part VI → Part VII → the Conclusion. Start with the changed engineering substrate, learn Modeling and Alignment, then follow one problem as successive models expose its constraints and design choices. The closing Parts develop the resulting theory and its implications for software engineering.
- **Adopt it in an organization.** Part I → Section 5.5 → Appendix G. Start with the changed engineering problem, see how several organizations are already restructuring agentic work, then use the adoption guide to decide where context, models, evidence, and authority are worth organizational investment. Parts II–IV provide the engineering practices; Part VI provides the underlying theory.

What This Book Argues

This section is a compact statement of the book's argument, intended for orientation and later reference. The Parts that follow develop and support these claims. You do not need their terminology yet.

This book makes six central claims. Together they explain the problem created by scale, what commodity intelligence changes, and how MAGE responds.

1. **Commodity intelligence changes the economics of software engineering.** Implementation capacity is becoming abundant relative to engineering judgment. As implementation gets cheaper, engineering effort shifts toward what remains scarce: deciding what to build, representing the system clearly enough to reason about it, producing evidence, and making important requirements enforceable by the engineering environment.
2. **Scale creates a reasoning problem.** Large software systems already exceed the reasoning horizon of humans; agents inherit the same problem. Software engineering has always answered scale with abstraction. Commodity intelligence does not remove that need. It makes the representations that guide the work more important.
3. **Modeling makes engineering knowledge and intent explicit.** Software engineers have always reasoned about architecture and other system properties through abstraction. What changes is how much of that reasoning can economically remain explicit. Purposeful models capture the knowledge and intent needed for engineering decisions while leaving irrelevant choices open. As commodity intelligence lowers the cost of deriving, maintaining, and using such representations, more engineering knowledge can be carried forward rather than reconstructed from implementation. Models let humans and agents reason about larger properties while deliberately leaving realization choices open where engineering has imposed no obligation.
4. **Alignment gives engineering obligations authority.** When agents perform more of the implementation, deciding what the system must do becomes increasingly separate from writing the code that does it. Important engineering decisions therefore cannot live only in instructions to an agent or in a person's head. Alignment encodes selected obligations into checks and controls that can constrain work or determine what the environment will accept. This distinguishes choices an agent is free to make from decisions engineering has already made. Engineers can then give agents substantial freedom in how they build the system while retaining control over the properties that matter.

5. **Governance conversion turns recurring judgment into durable engineering structure.** When a failure exposes missing knowledge or an unenforced obligation, encode the lesson into a model, procedure, or mechanism that future work can inherit. Durable structure becomes engineering capital when later work keeps benefiting from it.
6. **Engineering work will reorganize around what remains scarce.** As implementation becomes cheaper, more engineering effort will move toward representation, evidence, governance, coordination, and judgment. Agents may perform increasing portions of that work as well. The durable boundary will not be a particular task or level of abstraction, but responsibility for deciding what matters, what evidence is sufficient, what obligations should have authority, and what tradeoffs remain acceptable.

MAGE on One Page

This figure is a map, not a prerequisite. It compresses concepts developed throughout the book onto one page. On a first reading, use it for orientation; return to it as the terms become familiar.

MAGE has two principles and a conversion loop. Figure 0.3-1 shows the whole argument: scale creates the old problem, commodity intelligence changes its economics, and MAGE turns repeated effort into structure that future work can reuse.

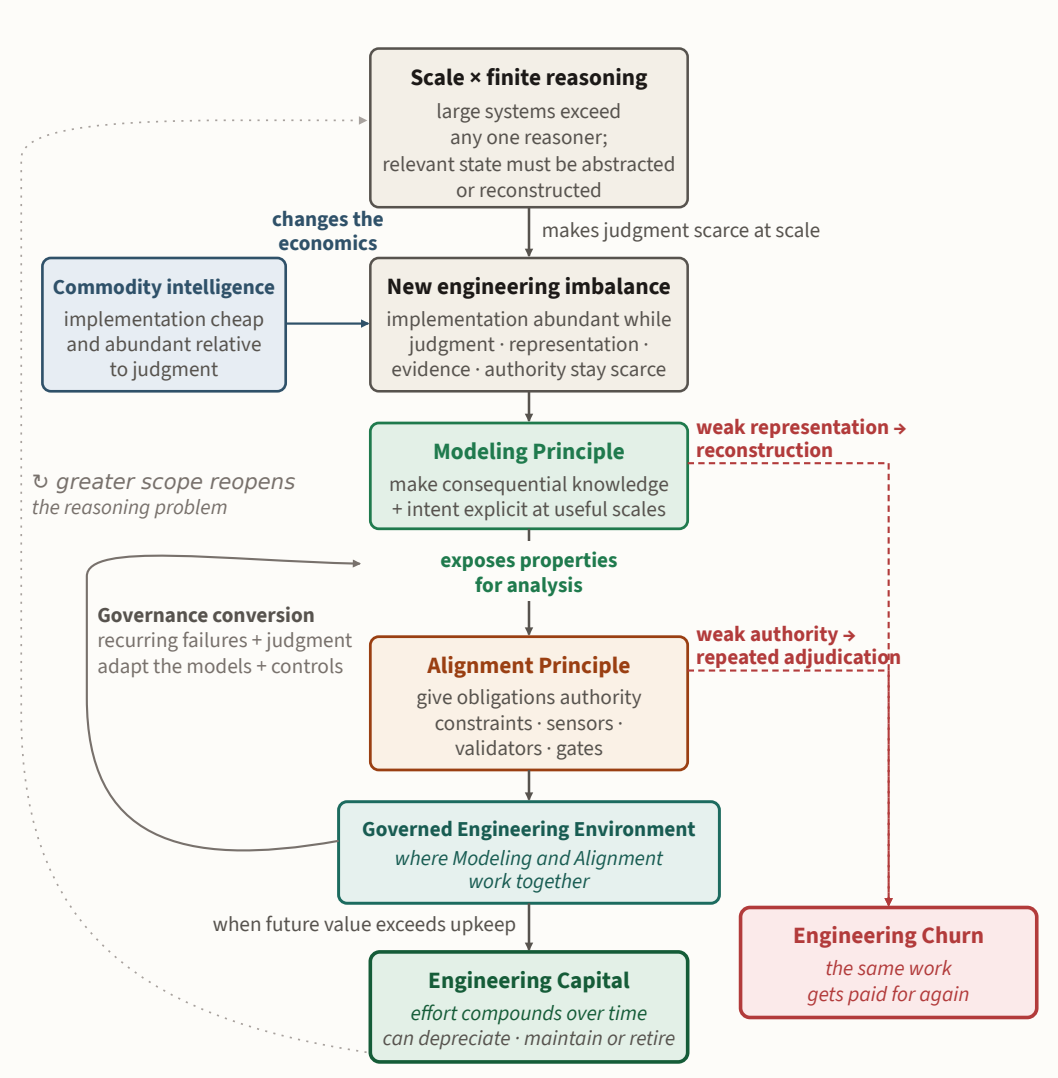


Figure 0.3-1: The MAGE method. Scale creates the enduring reasoning problem; commodity intelligence changes its economics. Modeling makes consequential engineering properties explicit and available to analysis; Alignment gives selected obligations authority. Governance conversion turns recurring failures and judgment into models and controls that future work can reuse, so engineering effort can accumulate instead of being paid for again.

Glossary

Agentic software engineering is moving quickly, and its vocabulary has not settled. This glossary is a reference to the vocabulary used throughout the book. Definitions are collected here for lookup; the main text introduces and motivates the important concepts in context. It has three parts: software engineering vocabulary, software agents, and the governed engineering environment.

Software engineering vocabulary

The book uses a small engineering vocabulary deliberately.

Engineering. Designing under goals, constraints, and trade-offs by reasoning about what a proposed solution may do, what it must do, and what evidence supports those beliefs. Software engineering applies that discipline to software systems.

Engineering obligation. A condition that an implementation or other engineering result must satisfy. Obligations may arise externally—from users, regulations, standards, contracts, or operating environments—or internally from engineering decisions about architecture, behavior, quality, or process. An obligation has authority when the engineered environment can enforce it. Durable mechanisms can give selected obligations that authority by constraining actions or determining whether work is accepted. For example, if a service must respond within 200 milliseconds, a performance gate gives that obligation authority when implementations that exceed the limit are rejected.

Model. A purposeful reduction of a system that preserves the information needed to answer an engineering question while suppressing detail the question does not need. Models make engineering knowledge and intent explicit so engineers and agents can reason about selected system properties without rebuilding that knowledge from lower-level artifacts each time.

Model classes. MAGE uses six recurring classes of models, organized by the engineering questions they help answer (Figure 0.4-1). The list is not exhaustive, and one representation may serve more than one class.

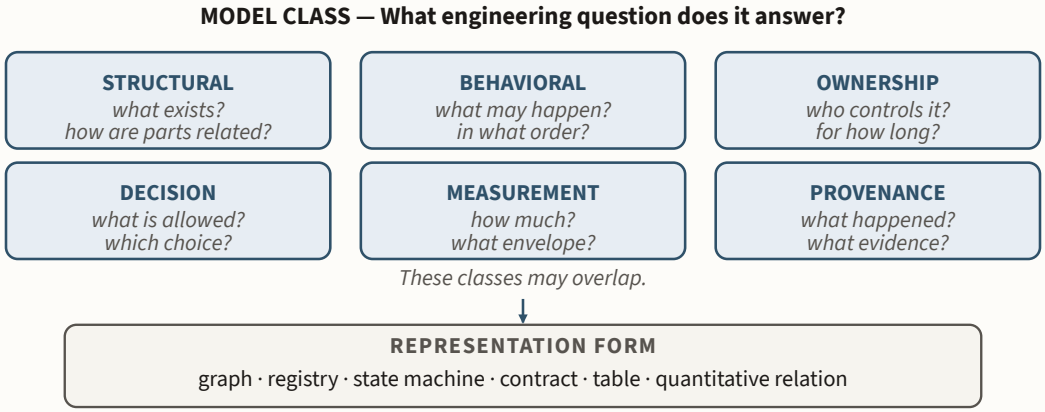


Figure 0.4-1: The six model classes, organized by the engineering questions they answer.

Modeling Principle. Make engineering knowledge and intent explicit in purposeful models so humans and agents can reason about larger questions without reconstructing the needed facts from lower-level detail.

Tolerance and degree of freedom. Engineering obligations need not determine a single acceptable implementation. A tolerance bounds the variation that an obligation permits; a degree of freedom is a choice that engineering leaves open within the applicable obligations. If a service must respond within 200 milliseconds, for example, implementations may vary in response time within that bound, while choices such as programming language may remain degrees of freedom if they do not affect any consequential obligation. An unmodeled choice is simply a choice that the current engineering representations do not make explicit; it is not necessarily free. If some alternatives would be rejected because they violate an engineering obligation, the choice is constrained even if the model does not say so.

Governance. Deciding what autonomous work must know, what it may do, what evidence it must produce, and what mechanisms will hold those decisions over time.

Alignment Principle. Give important engineering obligations authority over autonomous work. Encode them in mechanisms that constrain actions, produce evidence, evaluate evidence against obligations, or control admission. Some obligations can be enforced directly on actions or artifacts; explicit models let the environment state and check richer properties, including acceptable bounds on variation.

Governance conversion. When a recurring failure, repeated judgment, or observed gap reveals something future work should inherit, encode the lesson in durable structure: a model, packaged procedure, or mechanism that enforces an obligation.

Engineering capital. Durable engineering structure—models, packaged procedures, mechanisms, canonical documentation, and other reusable assets—that makes later work cheaper or less uncertain. Governance conversion creates such structure; it becomes capital when later work benefits from it, and depreciates when it no longer fits the system or costs more to maintain than it returns.

Commodity intelligence. General-purpose machine intelligence available cheaply and at enough scale that implementation becomes less scarce relative to engineering judgment. The term describes an economic change, not a claim that intelligence is free, unlimited, or itself sufficient for engineering.

Software agents

The agents, foundation models, harnesses, and operating limits involved in autonomous software work.

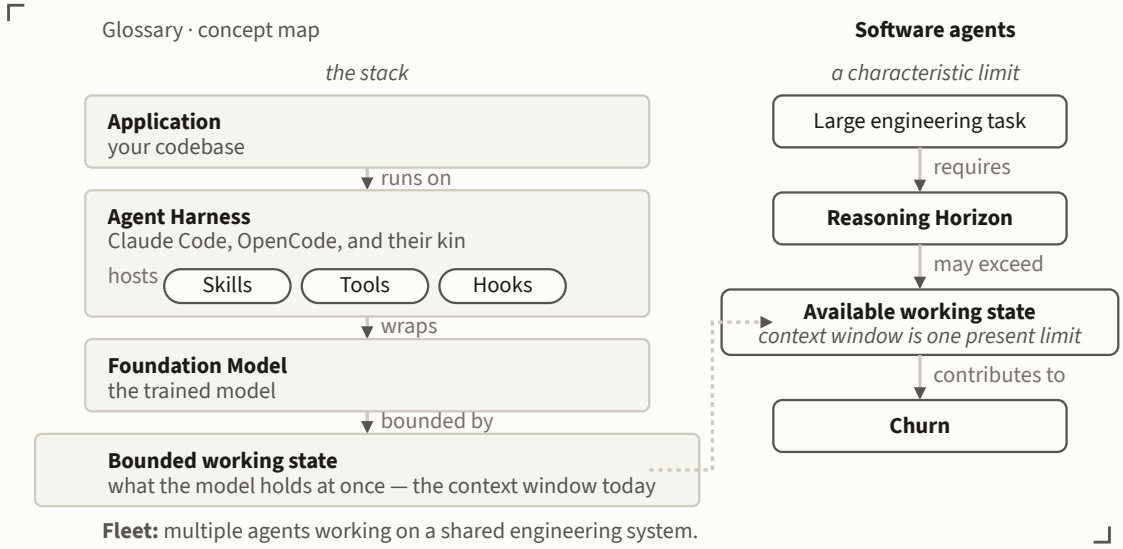


Figure 0.4-2: The agent stack and a characteristic failure mode. When a task's reasoning horizon exceeds the working state the harness can keep active, repeated reconstruction contributes to churn.

Agent. An AI system that pursues an open-ended or underspecified task through multiple steps, exercising judgment about what to do next and using tools in response to intermediate results. In this book, agent means a coding or engineering agent operating over software artifacts, tools, and an execution environment.

Churn. Work spent rediscovering context, undoing recent changes, repairing regressions, or reconciling inconsistencies instead of advancing the system.

Foundation model. The trained model at the bottom of the agent stack. Its capabilities and context capacity set important limits that the harness and engineered environment must work within.

Agentic harness. The runtime wrapped around the foundation model — Claude Code, OpenCode, and their kin — that feeds it prompts, manages the context window, and provides hooks, skills, and tools.

Skills and tools. Capabilities exposed through an agentic harness that extend what an agent can do. A skill provides reusable guidance for a recurring class of work; a tool provides a deterministic capability the agent can invoke rather than approximate through reasoning.

Fleet. The set of coding agents working on a shared codebase or engineering system.

Reasoning horizon. The amount of intermediate reasoning state that must remain available at once to carry a task to a correct result. It is a property of the task, not of who reasons — human, agent, or a later architecture. A *long-horizon task* needs more reasoning state than the available working memory can hold, forcing some state to be compressed and later reconstructed. Agent harnesses call this compaction; today, the context window is one important source of the limit.

The Governed Engineering Environment

The environment that gives engineering obligations durable authority.

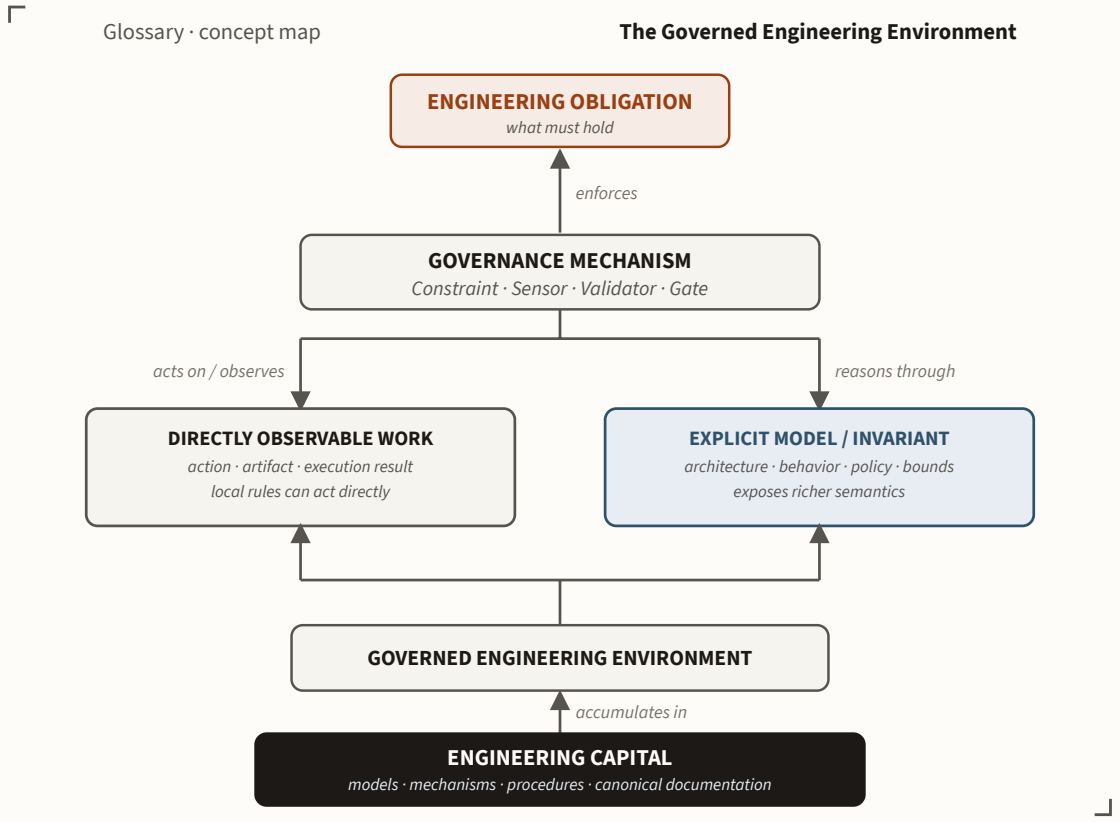


Figure 0.4-3: The governed environment. Governance mechanisms enforce engineering obligations by acting directly on observable work or through explicit models that expose richer semantics. An obligation enforced by the engineered environment is authoritative. Durable models, mechanisms, procedures, and documentation accumulate as engineering capital.

Governed Engineering Environment. An engineered environment where selected engineering obligations have authority through durable mechanisms instead of relying mainly on people to remember or inspect them. Explicit models let those mechanisms govern properties that would otherwise require a person to reconstruct or judge them.

Governance mechanism. A repeatable structure that enforces an engineering obligation in the environment, thereby giving the obligation authority. A mechanism can play four roles: constraints restrict permissible actions; sensors produce evidence; validators check evidence against an obligation; and gates decide whether work may cross a workflow boundary. One mechanism may play more than one role.

Semantic gap. The distance between where an engineering obligation arises and where the environment has enough meaning and evidence to decide whether the obligation is satisfied. The gap explains why some checks must wait until a later boundary, when enough of the system has been assembled to make the obligation decidable.

Model drift. A model and the system, evidence, or other representation to which it is meant to correspond cease to agree.

Traceability. Explicit, machine-readable links from model elements to the artifacts, evidence, obligations, or other model elements they represent, constrain, or correspond to.

Preface

How do we safely grant autonomy to commodity intelligence?

This is a book about engineering: specifically, about what engineering work becomes when reasoning capacity becomes abundant.

Generative AI can produce implementation at extraordinary speed. But software engineering has never been only the production of code. Engineers must decide what a system should do, determine which constraints matter, choose among designs, recognize when assumptions no longer hold, and establish whether the resulting system is trustworthy. Making implementation cheaper does not make these responsibilities disappear; it changes their relative importance. As implementation becomes cheaper, engineering judgment becomes more exposed as the scarce resource.

When agents can produce and modify software faster than people can inspect it, reliability cannot depend on humans repeatedly reviewing every generated decision. The challenge is to determine where judgment is necessary, where decisions can instead be made explicit, and where obligations can be enforced independently of the agent that produced the implementation.

Society has long entrusted consequential work to computers. Software flies aircraft, calculates tax returns, routes telephone calls, and controls power plants. Traditionally, software engineers stood between human intent and machine action: they translated goals, constraints, and judgments into sufficiently precise specifications, programs, and controls, and computers executed what they had made explicit.

Now, the boundary between human intent and machine action is shifting. General-purpose machine intelligence can interpret intent, make judgments, and produce implementation cheaply enough and at enough scale to become an ordinary engineering resource.[1] In this book, I call this increasingly available capability commodity intelligence.[1,2] Software engineering has begun to confront the consequences.

MAGE asks where engineering work is being paid for repeatedly—in reconstruction, review, or human judgment—and what model or mechanism could make that work durable.

Why MAGE? Why Models?

One way to understand MAGE—Model-Based Agentic Software Engineering—is as a classical software-engineering move: **add a level of indirection.**[2]

[1] Here and throughout, terms such as interpret, reason, and judge describe functional capabilities, not claims about cognition or consciousness. Contemporary language models approximate these capabilities through learned statistical computation; they are an imperfect realization of the general-purpose machine intelligence assumed here. Implementation has become cheap enough that agents can produce and modify software at extraordinary speed. The engineering problem therefore shifts: What should we build? What must the agent know about the system? How will we know whether its changes are correct? Which rules must it obey?

[2] Adding a level of indirection is so commonly a solution that it is sometimes jokingly called the *fundamental theorem of software engineering*.

When engineers wrote most implementation themselves, much of their engineering intent could act directly through the code they produced. They made design decisions and realized those decisions in implementation. When agents perform the implementation, that coupling weakens. The engineer increasingly acts through an intermediate structure: an engineered environment that represents what matters, constrains what may vary, requires evidence, and determines what work may be accepted.

MAGE makes that intermediate structure the primary object of engineering. Modeling externalizes consequential knowledge and intent into representations over which humans, agents, and tools can reason. Alignment gives selected engineering obligations authority through constraints, sensors, validators, and gates. Together they create a governed engineering environment through which autonomous implementation proceeds.

The purpose of this indirection is not to specify every implementation decision. Quite the opposite. An engineer should constrain what matters while deliberately leaving other choices open. Within those degrees of freedom, agents can exploit abundant implementation capacity to search, realize, revise, and improve solutions. The governed environment carries the decisions that should persist across those acts of implementation rather than asking each agent—or each human reviewer—to reconstruct them anew.

Software engineering has made this move repeatedly. Interfaces separate what a component provides from how it provides it. Virtual memory separates a program's address space from physical memory. Compilers separate a source-level program from the machine instructions that realize it. MAGE applies the same instinct to autonomous implementation: rather than making the agent itself the repository of engineering authority, put engineering authority in the environment through which the agent works. Software built with agents still deserves an engineer's standard of evidence: **success by observation is not engineering assurance.**

Public calls to make AI-generated software trustworthy have sometimes proposed correspondingly ambitious solutions, including advanced formal methods that mathematically establish properties of generated software.[3] Compared with such proposals, MAGE may look rather ordinary. That is deliberate. Conventional engineering has long separated the engineering of an artifact from its physical realization. The familiar "Designed by Apple in California" inscription is an everyday example: the people who establish a product's architecture, requirements, tolerances, and other engineering models need not be the people—or machines—that realize it. Other engineering disciplines rely on complementary representations of an artifact, use them to reason about different obligations, and establish that the realized artifact corresponds to them. MAGE brings that familiar pattern to software whose realization can increasingly be delegated to machines.

There is a further consequence. The engineering environment itself need not remain shaped around human limitations. Tests, documentation, code review, dashboards, and other familiar mechanisms evolved partly around what human engineers could feasibly write, inspect, remember, and execute. Giving agents access to those mechanisms is useful, but it captures only part of the opportunity. Machine-scale engineering can also make richer

representations economical: explicit behavioral models, large collections of invariants, machine-readable relationships among engineering concerns, and analyses over those representations that would be burdensome for people to maintain or apply manually.

MAGE does not make architecture or abstraction new. Its contribution is economic: more of the knowledge used in architectural and other engineering reasoning can remain explicit and operational.

That economic argument provides one reason for MAGE. Today, explicit representations and independent controls can make autonomous engineering cheaper and more reliable. In the future, better agents may reason both cheaply and well, reducing concerns about cost and fallibility. But MAGE carries a second, and more enduring, commitment. Engineers—and the societies that rely on them—should not cede their authority over engineered systems to the machines that produce them.

MAGE therefore asks not only how to make agents effective participants in an existing software-engineering environment, but how that environment should change when both reasoning and implementation can be performed by machines. Better context can help an agent make a consequential judgment correctly. Better models can sometimes make the relevant property explicit enough that the environment can analyze or check it instead. The first improves the reasoner's chances; the second can reduce how much judgment must be reconstructed and exercised again at all.

Where the theory came from

This book began with one hard engineering problem that emerged over months of building a real production system. Only later did I test its vocabulary against independently described systems from Cloudflare, Docker, Shopify, Spotify, Siemens, Zenseact, Uber, and GitLab. Part V presents both views of the evidence: DocAble in depth, then the industrial reconstructions in breadth. Part VI asks what account explains both. Figure 0.5-1 shows the evidentiary path.

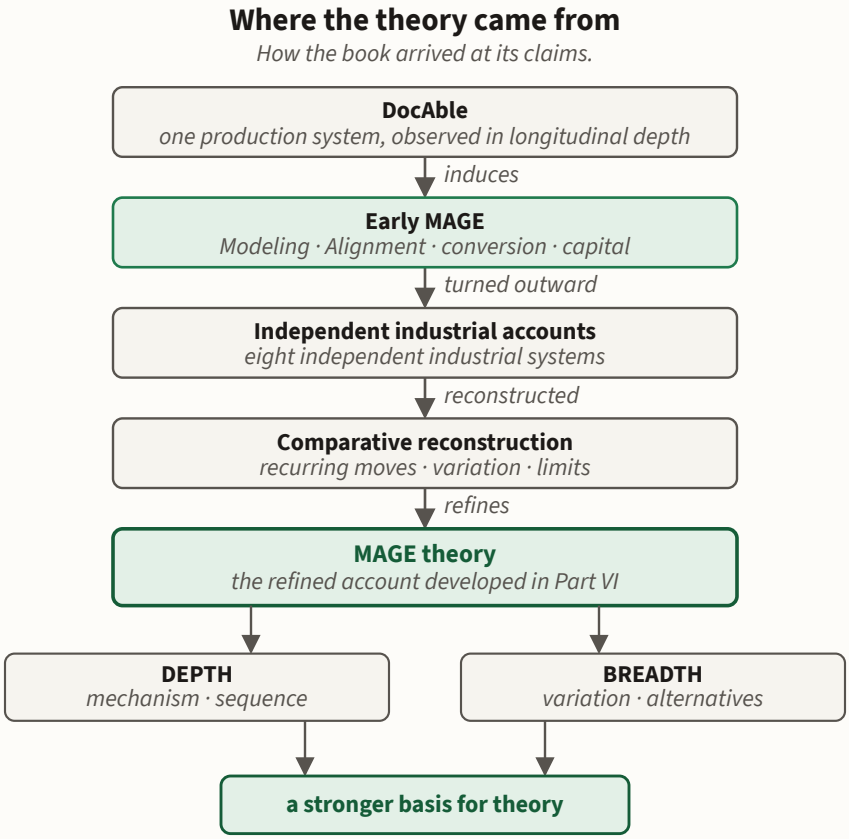


Figure 0.5-1: From case to theory. *MAGE began with one deeply observed production system, then was compared with eight independent industrial systems. The originating case shows mechanism and sequence; the industrial cases show variation and alternative ways to realize the same moves.*

The new bottleneck

Agents can produce changes much faster than the surrounding environment can understand or govern them. At scale, that mismatch becomes a bottleneck.

A fleet can generate, inspect, and modify implementation much faster than a human team. But more implementation capacity does not automatically produce more system-level understanding or better engineering judgment. When the necessary knowledge remains implicit, the fleet repeatedly reconstructs it. When an obligation is locally checkable, a test, sandbox, type, or gate may enforce it without a separately represented model. But when the obligation depends on architecture, ownership, end-to-end behavior, or another model

the environment lacks, somebody must supply the missing meaning. In many organizations that somebody is still a human reviewer. The result can work, but the human sits in the gap between what the environment can check and what the system actually needs guaranteed.

Churn is what happens when too much of that knowledge and judgment must be paid for again: agents rediscover context, undo recent changes, repair regressions, reconcile inconsistencies, or wait for humans to resolve questions the environment cannot settle. MAGE targets the recurring causes. Modeling makes more consequential properties explicit and tractable; Alignment makes selected obligations authoritative; governance conversion turns failures in either into structure the next task inherits.

The originating system

This book began with DocAble, a production service for remediating inaccessible documents. I built it over several months, almost entirely by dispatching AI coding agents rather than writing the implementation myself. By the period studied here, coding agents had written roughly 491,090 lines of production code. The environment governing that code —tests, checks, agent infrastructure, and documentation—had grown about three times larger, to 1,501,907 lines.

Although DocAble is a tidy little demonstration of techniques from natural language processing and compilers, the most interesting thing about DocAble is not DocAble. It is MAGE. MAGE predicts that one engineer can use agents to build and sustain a production system whose implementation would ordinarily require a software team. DocAble shows this in action. It also poses the question that started this book: once agents write almost all of the code, what engineering remains?

DocAble therefore serves as the book's deepest worked example. Parts II and III examine the mature system from the perspectives of Modeling and Alignment. The interlude that follows Part III traces one engineering problem as successive models expose its constraints, support predictions, and change the design. Part V takes the longer view, reconstructing how DocAble's engineering machinery emerged and comparing it with independent industrial practice.

Evidence and scope

MAGE draws on two kinds of evidence. The first is a single longitudinal case: one production system, built by one engineer using one vendor's agent ecosystem over several months and used in production by me and my colleagues at Purdue. Numbers from that case are observations, not measured laws. The second is a set of reconstructions based on independent industrial accounts. Those accounts show less of how each system evolved, but more variation in technical and organizational choices, including systems that rely more heavily on explicit models and systems that rely more heavily on controls and human judgment. Part V presents both sources, states their limits, and ties empirical claims to supporting data or citations.

What the method turned out to be

Two principles organize MAGE:

1. Modeling makes consequential knowledge and intent explicit.
2. Alignment gives important engineering obligations authority.

They solve different problems but interact. Alignment always assumes something about what an acceptable result looks like: what must hold, and sometimes what variation remains acceptable. That understanding may remain tacit or be encoded locally rather than written down as a model. A sandbox can forbid an action using a local rule rather than a separate system model; a test can block a regression by encoding expected behavior; a human reviewer can judge an obligation that no machine-readable model yet captures. But they have a limit: they can govern only what the environment can state, observe, or ask a person to reconstruct. Modeling changes that boundary. By making consequential relationships explicit—structural, policy, behavioral, ownership, measurement, and others—Modeling lets humans, agents, and tools reason over them directly and apply analyses suited to those representations. Where those properties express obligations, Alignment can give them authority through governance mechanisms that consume the available evidence.

MAGE therefore places Modeling first as a methodological choice rather than a logical prerequisite. Relevant knowledge and intent may already be tacit or encoded locally; Modeling makes explicit only what is worth externalizing; it need not eliminate realization choices that engineering has reason to leave open. The method asks the engineer to externalize the abstractions that matter before repeatedly spending human attention on what the environment cannot see. Once those properties are explicit, some judgments that once required reconstruction can be handled by repeatable machinery.

1. **Model the consequential knowledge.** Choose the engineering question first, then externalize the facts and intent needed to answer it without carrying unnecessary detail. The goal is not to model the whole system. It is to make larger properties easier to reason about without reconstructing them from implementation.
2. **Give obligations authority.** Encode important obligations into constraints, sensors, validators, and gates. Some mechanisms can constrain actions or artifacts directly; others evaluate evidence and determine whether work may proceed. Explicit models make richer, system-level obligations available to automated analysis and, where appropriate, enforcement.
3. **Do the governed work.** Let agents operate inside the resulting environment. Models provide reasoning surfaces; governance mechanisms provide evidence and boundaries. Human attention moves toward questions that genuinely require judgment instead of being spent reconstructing facts the environment can check itself.
4. **Convert recurring failures.** When autonomous work exposes missing knowledge, a weak abstraction, or an unenforced obligation, repair the instance and then encode the lesson into durable structure. The environment improves because the next task inherits the model or control rather than the memory of the person who solved the problem.

Neither principle is MAGE by itself. The method is the cycle they create: **model where better representation changes the engineering problem; align what can be made authoritative; do the governed work; convert recurring failures and judgment into durable structure; repeat.** Useful structure becomes engineering capital when later work keeps benefiting from it.

Who this book is for

This book is for software engineers who suspect the ground has moved and want to know where to stand. It is not primarily about the internals of agents; it is about the engineering structures that become important once agents can perform substantial implementation work autonomously. It is also for technical leads, architects, and engineering managers deciding how autonomous software development should work inside an organization. A fleet's models, policies, evidence, infrastructure, and remaining human decisions belong to the engineering organization, not just to an individual's tool.

The book is also intended for advanced undergraduate and graduate students studying software engineering, software architecture, or AI-assisted development. It assumes familiarity with ordinary software-development practice: source control, testing, APIs, build and deployment workflows, and the basic idea of software architecture. No prior experience with coding agents, formal methods, model-based engineering, or machine learning is required. Concepts from those areas are introduced where the argument needs them and illustrated through the systems and examples in the book.

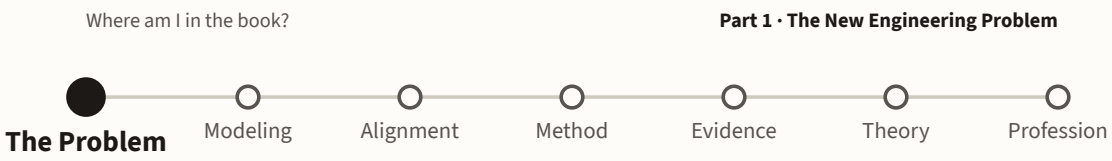
MAGE is neither a general software-engineering textbook nor a textbook on model-based systems engineering (MBSE). It assumes the relevant knowledge of those fields and draws on their representations and techniques when they serve the engineering question at hand. MAGE adopts the MBSE habit of reasoning through explicit representations and applies it to choosing, structuring, and governing models for agentic software engineering; it does not attempt to teach the full range of modeling languages, viewpoints, transformation techniques, systems-engineering methods, or MBSE practice.

The book draws on several other traditions for the same reason. Model-driven software engineering supplies established ideas about metamodels, transformations, and generated artifacts; programming languages and compilers supply types, intermediate representations, and constrained transformation; formal methods supply executable specifications and machine-checked properties. MAGE uses those ideas where they help, without trying to teach the disciplines they come from. Those subjects have their own literatures and textbooks. The same applies to machine learning: the book introduces only the concepts the argument or examples need.

After reading this book, you should be able to recognize when an engineering problem is being paid for repeatedly in reconstruction or human judgment; decide whether a better model, stronger enforcement, or both could remove that recurring cost; and design mechanisms suited to your own system rather than copying another organization's process. One move recurs throughout: **convert recurring judgment into durable engineering**

structure. DevOps learned to make infrastructure, configuration, and policy durable as code. MAGE extends that instinct to models, constraints, evidence, validators, workflows, and the engineering judgment encoded in them.

Part 1: The New Engineering Problem



QUESTION THIS PART ANSWERS

What becomes the engineering problem when implementation becomes abundant?

FOUNDING PREMISE

Commodity intelligence makes implementation abundant relative to engineering judgment.

Engineering effort concentrates around what limits reliable production. As implementation capacity becomes cheaper and more abundant, judgment, representation, evidence, and authority become relatively scarcer. Engineering effort moves with the constraint.

NEW HERE

Commodity intelligence · The Printer · Model · Governed Engineering Environment · Reasoning horizon · Probabilistic surface

Engineering reorganizes when a constraint moves.[3] Steam radically reduced the cost of mechanical power. Integrated circuits did the same for computation. Coding agents are now reducing the cost of software implementation. A scarce factor constrains output, so it draws investment and attention. Implementation was never the only scarce input to software engineering, but for most of the field's history it consumed enough expert effort to limit what teams could attempt and how quickly they could change a system. As its marginal cost falls, other constraints become more visible: deciding what to build, representing a large system well enough to reason about it, producing evidence that a change is acceptable, and giving consequential engineering decisions authority across many changes.

Abundance does not make implementation unimportant; it changes where additional engineering effort earns the greatest return. A factory with unlimited machine capacity and one inspector has not stopped manufacturing. Inspection has become the throughput constraint. In software, agents can now produce changes faster than engineers can specify, understand, validate, and govern them. Models let later work reuse representations and judgments across many acts of implementation; validators and gates can likewise carry selected acceptance judgments forward. Engineering effort therefore moves toward deciding what to represent, what evidence to require, which obligations to give authority, and how the surrounding environment should carry them. This Part asks what follows from that shift: what remains hard, which properties of the new substrate matter, and what engineering problems they leave us to solve.

[3] The intuition is familiar from Amdahl's law: accelerating one part of a computation increases the relative importance of the work that remains[4]. Goldratt's Theory of Constraints states the broader operational version: system performance is governed by a constraint, and improving that constraint eventually moves attention to another[5]. The same constraint logic applies to software-engineering economics.

1.1 The Printer

Coding agents make implementation cheap enough to change the engineering problem. Describe a change and a capable agent can often build it quickly, at a scale that once required substantial human effort. But the agent may not build what you wanted. Reliable production still requires a sound specification, useful representations, and evidence that the result is acceptable.

Think of the agent first as a 3D printer: productive machinery that can realize an unusually broad range of engineering intent. A conventional machine tool arrives with a comparatively settled operation—cut here, drill there, fasten these parts. A 3D printer is more open-ended.[4] The same machine can fabricate a gear, a bracket, a housing, or a prosthetic part. Its productive capacity does not tell the engineer which object to make, what properties that object must satisfy, or which representation is adequate to realizing those properties.

That is the useful resemblance to commodity intelligence. A coding agent supplies broad realization capacity without supplying the engineering discipline for using it. The engineer must still decide which distinctions matter, what must be represented explicitly, which choices may remain open, and which boundaries the productive machinery may not cross. Open-ended capability therefore creates a second design problem: engineers must also design the forms, types, interfaces, constraints, and evidence through which that capability can be used reliably.

A 3D printer also makes clear that intent is not the same thing as fabrication instructions. A photograph can communicate the desired appearance of a part without specifying its internal geometry, material, tolerances, load paths, or allowable joints. The machine needs a representation adequate to the properties the part must have, together with constraints on how realization may proceed. Software agents can infer much more from an incomplete description than a physical printer can, but inference does not remove the distinction. That inferential capacity is precisely what makes the machinery powerful, but inference remains different from an engineered guarantee. The engineering problem is still to make consequential intent available to the machinery in a form that can guide and govern realization.

When agents perform the implementation, engineering leverage moves into the engineered environment. This is the level of indirection introduced in the Preface. Instead of expressing engineering intent primarily by writing the implementation, the engineer designs an environment through which autonomous implementation proceeds. That indirection creates a new engineering surface: representations can carry consequential knowledge, mechanisms can give obligations authority, and agents can remain free to choose among realizations that satisfy them. Figure 1.1-1 shows the shift.

[4] I use *open-ended* here for a productive technology whose useful output classes are not fixed by the machine itself. The capability arrives before the engineering conventions for exploiting it. New artifact types, representations, interfaces, constraints, and operating practices are discovered as people learn what the technology is useful for. Writing and general-purpose computation offer earlier examples; commodity generative intelligence adds a similarly broad realization substrate to engineering.

In the printer metaphor, the governed engineering environment is the engineered instruction set around the productive machinery: the representations that tell it what matters, the boundaries that constrain realization, and the evidence used to decide whether the result is acceptable. Some of that environment helps the agent reason better; some removes selected judgments from the agent's discretion altogether.

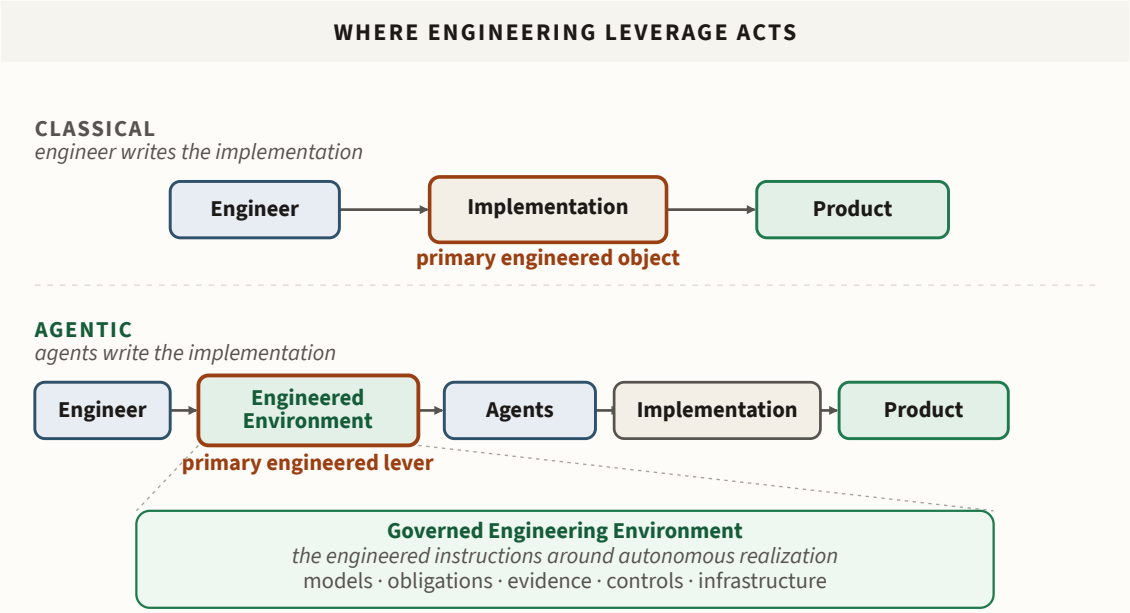


Figure 1.1-1: Where engineering leverage acts. *When engineers write most implementation directly, much of their engineering leverage acts through the code. When agents provide abundant realization capacity, more of that leverage moves into the governed engineering environment—the software analogue of the models, fabrication constraints, process instructions, and acceptance criteria surrounding a 3D printer. The environment represents consequential intent, constrains realization, and evaluates what the fleet produces.*

1.1.1 The Engineering Object

Agent products and foundation models will continue to change.[5] The durable engineering problem begins when an agent works autonomously on a system large enough, important enough, or long-lived enough that "it usually works" is not an acceptable standard.

The object of engineering therefore expands beyond the implementation. Engineers also design the environment through which autonomous work proceeds: what the agent can

[5]Engineering predates machine learning in its use of models as purposeful representations of systems. Machine learning later adopted the same word for a learned approximation of a function—also reasonably a model, but a different kind. This book gives the engineering usage precedence: unqualified model means a purposeful representation of an engineered system or some property of it. I therefore use foundation model for the trained machine-learning system and agent for the acting system built around one through a harness, tools, and working state. Where the distinction matters, I will not ask the reader to infer which "model" I mean.

know, what it may do, what evidence it must produce, and which obligations the environment will enforce.

MAGE is principally about that environment. We will get to foundation models, harnesses, tools, hooks, skills, and workflows because their properties determine what can be engineered around them. But the method lies in how engineers represent what matters, how they give obligations authority, and how they decide which recurring judgments should become durable engineering structure.

In the printer metaphor, Parts II and III develop the two missing halves of the instruction package: Modeling makes consequential properties explicit; Alignment gives selected properties authority over what the printer is allowed to produce.

That environment need not simply reproduce the one built for human developers. Agents can consume and maintain representations at scales that would be impractical for people, and automated analyses can check properties that human review would otherwise have to reconstruct from implementation. The design question is therefore two-sided: what existing engineering machinery should agents inherit, and what stronger machinery becomes practical because the workers are now machines?

MAGE is not a general software-engineering method. It does not attempt to teach engineers which programming language to choose, which architectural or design pattern fits a system, or which verification technique is appropriate. Those questions draw on decades of accumulated software-engineering theory and practice. MAGE asks a different question: when such engineering knowledge matters to autonomous implementation, how should it be represented so that the machinery can use it, and which resulting obligations should be given authority over what the machinery may produce?

Commodity intelligence can also change which traditional engineering choices remain consequential. Consider the choice of a programming language. A human team might once have chosen Java over another suitable language simply because its developers already knew Java. If an agent can implement competently in either language, that constraint largely disappears: some language still has to be chosen, but the particular choice may no longer deserve much engineering attention. Other language choices remain consequential. A deployment target may require a particular toolchain, or a safety requirement may favor a language that provides a needed property. The relevant question is therefore not whether engineers have traditionally made a choice, but what property made the choice consequential.

The same distinction applies at larger scales. Suppose a service needs persistent storage. If either a relational or nonrelational database can satisfy every property that matters, choosing between them may simply be a degree of freedom left to realization. But if the system requires transactions across related records, a particular consistency model, or another property that distinguishes the alternatives, the choice becomes consequential. MAGE does not supply the database theory needed to make that judgment. It asks that the consequential property be represented—and, where necessary, given authority—rather than relying on each implementing agent to rediscover why the choice mattered.

Notice the distinction between unmodeled and free. A choice is unmodeled when the engineering representations do not state it; it is free only when engineering permits the alternatives. Suppose the model says nothing about which database to use. If either database really can satisfy every consequential property, the omission represents a genuine degree of freedom. But if one alternative would violate a security, consistency, or performance obligation, the choice was never free. The model simply failed to make the relevant obligation explicit.

That emphasis has important antecedents. AI research has repeatedly improved machine reasoning by changing the representations, memory, tools, and external machinery available to the reasoner. Recent agentic software-engineering systems similarly improve performance through retrieval, program structure, and purpose-built interfaces.[6] MAGE changes the engineering object: the concern is not what a reasoner needs to solve one task, but the engineered environment through which autonomous software work proceeds over time.

1.1.2 Suspect the Setup First

The Printer suggests a useful diagnostic rule: suspect the engineered setup before inferring a machine limit. Observed performance is a property of the whole arrangement, not just the productive machinery.

A 3D printer separates productive capability from the instructions that condition it. A photograph may show what an object should look like, but appearance does not specify internal structure, material, tolerances, load paths, or fabrication constraints. A failed part can therefore reveal several different problems: the machine lacked capability; the model omitted a consequential property; the chosen material could not satisfy it; or the fabrication instructions permitted a realization that should have been ruled out.

Software agents blur this distinction because they can infer missing structure. A capable agent may produce an impressive result from a screenshot, prose request, or incomplete specification. But inferred completion is not an explicit statement of which properties must hold, which tradeoffs are acceptable, or what evidence establishes acceptance.

When an agent produces the wrong result, inspect first what the engineer controlled: the instructions, the representation of the problem, the task boundary, the available tools, and the evidence the work was required to satisfy. Sometimes the foundation model simply lacks the necessary capability. Failed output alone does not establish that diagnosis.

---

[6] On the AI side: LLM+P translates natural-language problems into a classical planner[6]; MemGPT externalizes state into managed memory tiers[7]; and Graph of Thoughts makes graph structure part of the reasoning process[8]. In agentic software engineering: RepoCoder retrieves repository-level context[9]; SWE-agent engineers the agent-computer interface[10]; and AutoCodeRover reasons over program structure rather than treating a project as merely files[11]. The ambition predates foundation models: Rich and Waters's Programmer's Apprentice envisioned an AI system as a new agent in the software process, sharing explicit programming knowledge with the engineer across implementation, design, and requirements[12]. Part VII returns to these traditions and to what commodity intelligence changes.

Abundant implementation raises the stakes. Capable agents can execute a bad direction with extraordinary speed. Velocity amplifies sound engineering and bad engineering alike.[7]

[7] My field notes on this point were less measured: “Claude is the fastest road to hell.”

1.2 MAGE by Example: Summary of the DocAble Case

DocAble provides the book's deepest worked example: a production accessibility system built largely through coding agents. Part V presents the full case study; before examining its models and Alignment mechanisms here, a few facts about the system are in order.

1.2.1 Accessibility and the Engineering Experiment

In 2024, the U.S. Department of Justice gave public universities a deadline to make their digital content accessible. A typical slide deck can fail the standard in dozens of ways: images without descriptions, missing titles, or a reading order that a screen reader cannot follow. I estimated that remediating one course's materials by hand would require roughly 8 full work weeks; a university offers thousands of courses. I personally maintain two courses and had no desire to spend a full semester remediating them—at least not manually. I was, however, perfectly willing to spend a full semester attempting to automate the work.

In early 2026, coding agents had also become capable enough to make a different experiment possible. I wanted to know what software engineering would look like if I stopped writing most of the implementation myself, so I used accessibility remediation as a real production problem on which to find out. The system became DocAble; the resulting method became MAGE. MAGE makes the judgment of one engineer capable of governing the implementation capacity of a software team. Part V returns to the build and examines how that capability emerged.

1.2.2 What the User Sees

From the user's perspective, DocAble is simple: upload an inaccessible document; receive a remediated document and an evidence record. It works across common document formats: Word, PowerPoint, and Excel; their LibreOffice equivalents; PDF; and text-based formats including LaTeX and Typst. Figures are described, reading order is repaired, structural accessibility is restored where possible, and consequential changes are recorded so that the result can be inspected and, where necessary, reversed.

Why not simply give the entire document to a capable multimodal model and ask it to make the document accessible? I tried that too. For documents within their operating range, capable models can produce impressive results. But the approach is expensive because the model must reconstruct properties already available deterministically in the document (the document's title may be right there on the page, but represented as large bold text rather than Word's Title style); large documents can exceed its practical operating range (the model may simply give up before finishing the deck); and support varies across formats (a model that handles PowerPoint may not accept PDF).

More importantly, "the model said it was accessible" is not an adequate compliance case. The governing obligation is not merely to produce a plausible-looking artifact, but to

provide students with a substantially equivalent experience. If a result is challenged, the university needs credible evidence of what the system checked (every image was examined for a usable description), what it changed (this image received this alternative text), what it could establish (the repaired reading order matches the intended sequence), and what limitations remain (this element could not be remediated automatically). DocAble therefore treats accessibility remediation as an engineering problem rather than a single act of inference.

Before looking inside the system, Figure 1.2-1 shows only the engineering result it is supposed to produce.

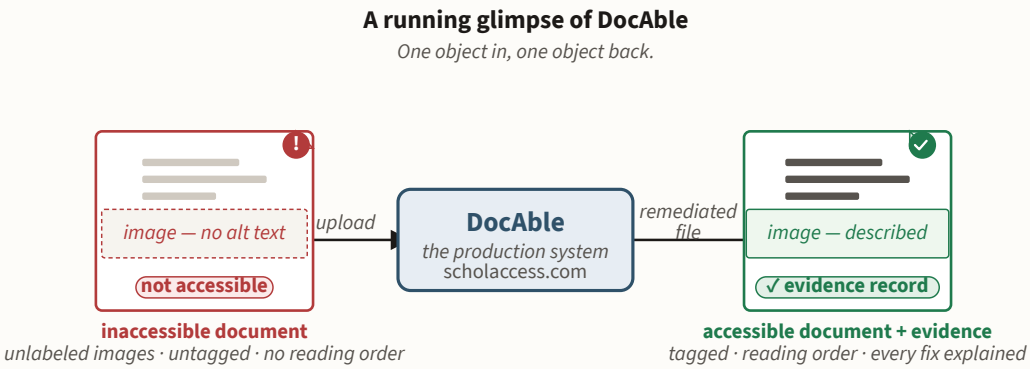


Figure 1.2-1: DocAble from the user's view. *An inaccessible document goes in; a remediated document and evidence of consequential changes come back. Internal structure is omitted because it is not yet relevant.*

That simple interface hides a large production system. At the observation point used in this book, DocAble comprised roughly 491,090 lines of production code, governed by a support apparatus about three times larger (1,501,907 lines, a 3.0× support-to-production ratio). The examples that follow isolate individual engineering questions from that system. They should not be read as the architecture of a small document-processing script.

1.2.3 One System, Several Engineering Questions

Building DocAble well is not one engineering problem. Different questions require different representations.

Ask what the system contains and how its parts relate, and you want a structural view. Ask what happens to a document after upload, and you want behavior: the states a job moves through and the transitions that are legal. Ask which component may call another, and you want a model of permitted relationships. Ask whether the system is fast and cheap enough, and you want measurements and their allowable envelopes. Ask whether the evidence returned with a document faithfully describes what happened, and you want provenance.

The system has not changed between these questions. The representation has. There is no single model of DocAble that is best for all of them. Each model is a purposeful reduction that preserves the facts needed for an engineering question and suppresses detail that question does not need.

That is the modeling move Part II develops.

1.2.4 Following One Job

Before Parts II and III examine particular properties of DocAble, Figure 1.2-2 follows one ordinary job through the system.

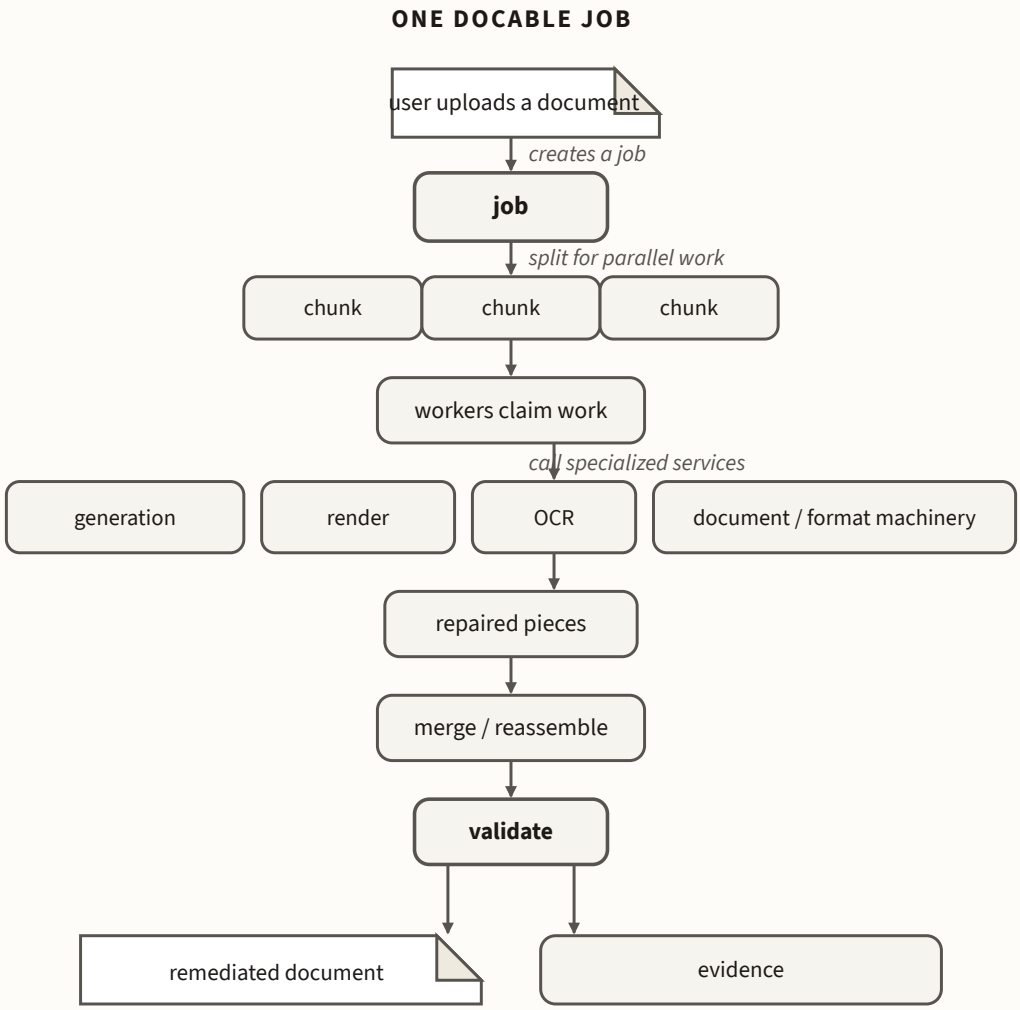


Figure 1.2-2: Processing one document in DocAble. *A user submits a document, creating a job that tracks its processing. The work may be divided into chunks and processed by workers using specialized services for generation, rendering, optical character recognition (OCR), and document-format processing. The resulting changes are reassembled and validated, and the remediated document is returned with evidence.*

A job tracks one document's trip through the system. A chunk is a portion of that work that can be processed independently. Workers claim and process chunks, calling specialized services for capabilities such as image description, rendering, conversion, or OCR. Queues and other infrastructure coordinate those handoffs. The final outputs are the remediated document and evidence describing consequential changes and remaining findings.

The same worker will appear differently in later models. It may be a deployable component, an actor in a lifecycle, an owner of a resource, a subject of an access policy, or a source of measurements. These are not competing descriptions. Each preserves different properties of the same system for a different engineering question: structure, behavior, ownership, decisions, measurements, or provenance.

1.2.5 How the Book Uses DocAble

DocAble is the book's deepest worked example because the system can be examined at the resolution the method requires. Parts II and III use the working system to ask two different kinds of questions. Part II asks which representations make important system properties tractable. Part III asks how engineering intent becomes authoritative over the work.

The interlude after Part III follows one engineering problem in depth. A large presentation first exposes a memory problem, then a representation problem, then an execution problem. Successive models reveal each one and support design decisions before the corresponding implementation is complete.

Part V uses DocAble differently. It examines the build itself: what failed, what those failures revealed, and how recurring judgments became models, controls, and other durable engineering structure. The earlier Parts show the resulting machinery; Part V shows the engineering experience from which much of it emerged.

DocAble provides depth, not the book's entire evidentiary basis. Other engineering organizations reappear throughout the book to show where related structures recur, where they take different forms, and where the evidence supports a more limited interpretation.

We can now examine the machinery that makes autonomous implementation possible and the engineering mechanisms through which MAGE acts on it.

1.3 Agentic Machinery and Engineering Mechanisms

Engineering methods have to fit the substrate they act on. Reinforced concrete works because concrete takes compression and steel takes tension; where the reinforcement goes follows from those properties. Agentic software engineering likewise begins by asking what productive machinery has entered the engineering process and which of its properties matter.

For MAGE, the relevant machinery is not a foundation model alone. A foundation model operates through an agentic harness inside an engineered environment. Together, they determine what autonomous work can do and where engineering can act on it.

1.3.1 Foundation Model and Agentic Harness

A foundation model supplies broad semantic reasoning at low marginal cost. Training a frontier model can require billions of dollars in capital, data, and computation; using an already-trained model for one additional reasoning task is comparatively cheap. An agentic harness—the runtime around it—turns that reasoning into action, manages the working context available to it, and exposes points where the resulting work can be observed or constrained. The useful engineering unit is therefore not the foundation model alone, but the foundation model and harness operating inside an engineered environment.

Four properties of the foundation model matter to the engineering argument:

- **Broad semantic reasoning.** A foundation model can interpret, synthesize, and search across open-ended domains. Give it an abstraction—a schema, state table, or policy—and it can reason about that abstraction rather than merely execute a fixed script.
- **Probabilistic output.** A foundation model can resolve consequential engineering questions incorrectly, and its confidence in its own result is not independent evidence that the result is correct. The producing run cannot establish engineering authority merely by affirming its own output.
- **Bounded working state.** Active reasoning state is finite. Durable system knowledge must therefore persist outside the producing run and be supplied, retrieved, or reconstructed when later work needs it.
- **Cheap repeated reasoning.** Search, generation, repair, and reconciliation—reasoning work that once consumed substantial engineering labor—can now be repeated at previously impractical scale.

The harness contributes three complementary capabilities:

- **Action.** Tools turn model outputs into consequential operations: editing a file, running a command, calling an API, or changing some other part of the engineering system.
- **Context management.** The harness assembles and maintains the working state available to the reasoner through mechanisms such as retrieval, memory, compaction, and task state.

- **Interposition.** Because consequential actions pass through identifiable interfaces, the harness can expose points where the environment observes, constrains, redirects, or denies them.

None of these properties is wholly alien to human software engineering. Humans also reason with bounded state, make uncertain judgments, act through tools, and rely on evidence outside themselves. What changes is their combination with broad, inexpensive semantic reasoning and a machine-mediated environment around its actions. Recent work on harness engineering similarly treats autonomous capability as a property of the foundation-model–harness–environment system rather than the foundation model alone[13–15]. Figure 1.3-1 shows the resulting substrate. This combination changes the economics of familiar engineering problems and exposes new places to act on them.

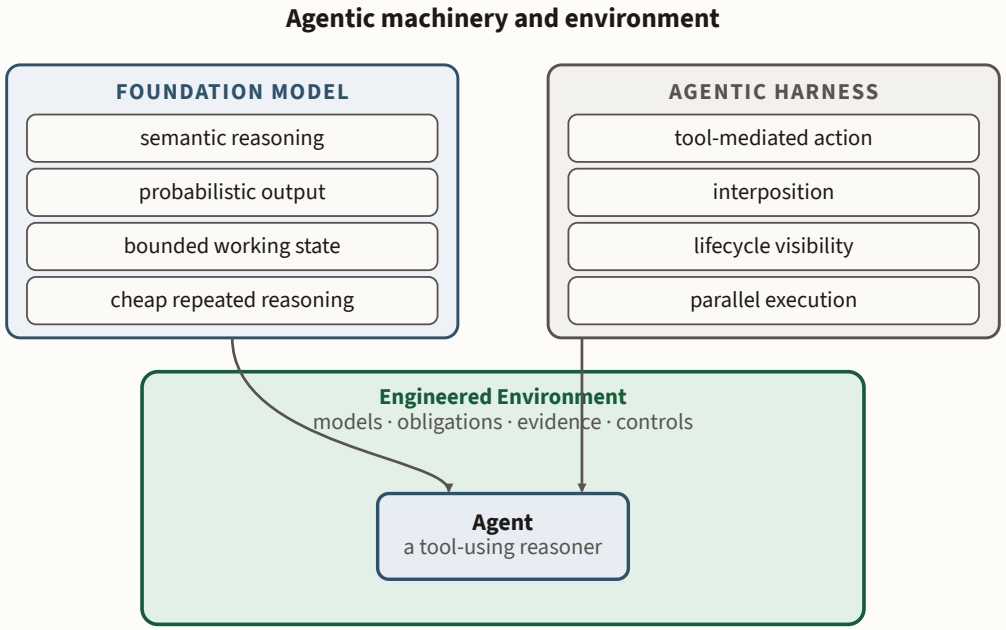


Figure 1.3-1: Agentic machinery and environment. *The foundation model supplies inexpensive semantic reasoning under finite, probabilistic working conditions; the harness turns that reasoning into action, manages its context, and exposes points where the environment can observe or constrain it. These are the properties of the productive substrate the engineered environment must work with.*

1.3.2 The Reasoning Horizon

Scale creates a reasoning problem before an agent ever enters the picture. A sufficiently large software system exceeds what one engineer can keep actively in mind; agentic work inherits the same condition. A long engineering task is not a prompt and an answer but a sequence of related decisions: read the system, form a plan, make a change, inspect the result, reconcile what changed, and carry consequential earlier decisions into later work. The state required by that work grows while the active state available to the reasoner remains finite.

Once a task exceeds that active state, some consequential information must live elsewhere and be recovered when needed. If the reasoner repeatedly reconstructs its picture from raw implementation, loose notes, or lossy summaries, relevant decisions compete with irrelevant detail. Solved questions are re-solved, earlier constraints are contradicted, and system structure has to be rediscovered. The problem is therefore not merely how much information can be stored or retrieved, but how much question-relevant state must be reconstructed for the next engineering judgment.

Agent harnesses can move this boundary through larger contexts, retrieval, persistent memory, and compaction. These mechanisms make more information available, but availability is not the same as a useful representation. Empirical work on long-context language models shows both limits. Usable context can fall well short of nominal capacity, and information already present in the context is not uniformly accessible.[16,17] Greater model capability can move these boundaries; it does not make finite working state disappear.

Repository-scale studies report the same problem operationally: as the relevant state spreads across a larger codebase, navigation and reconstruction consume an increasing share of the work.[18] The precise boundary depends on the model, harness, repository, and task. The important point is the shape of the problem, not a fixed repository-size threshold.

This is a locality problem for reasoning state. Systems have long dealt with finite working sets by keeping useful state close to the computation that consumes it rather than making all state equally immediate. Purposeful representations apply the same principle semantically. A dependency graph can answer a question that thousands of retrieved source lines merely make available; a state machine can expose a behavioral property that additional context leaves implicit. Better representations reduce the amount of state that must be reconstructed for a particular engineering question.

The **reasoning horizon** is the boundary beyond which consequential state can no longer be carried effectively in active reasoning and must instead be recovered or reconstructed. Larger context windows, stronger reasoners, retrieval, and memory can move that boundary. Modeling can move it differently, by changing the representation over which reasoning occurs. It does not make the reasoner unbounded; it makes selected engineering questions require less reconstructed state. Figure 1.3-2 shows the mechanism.

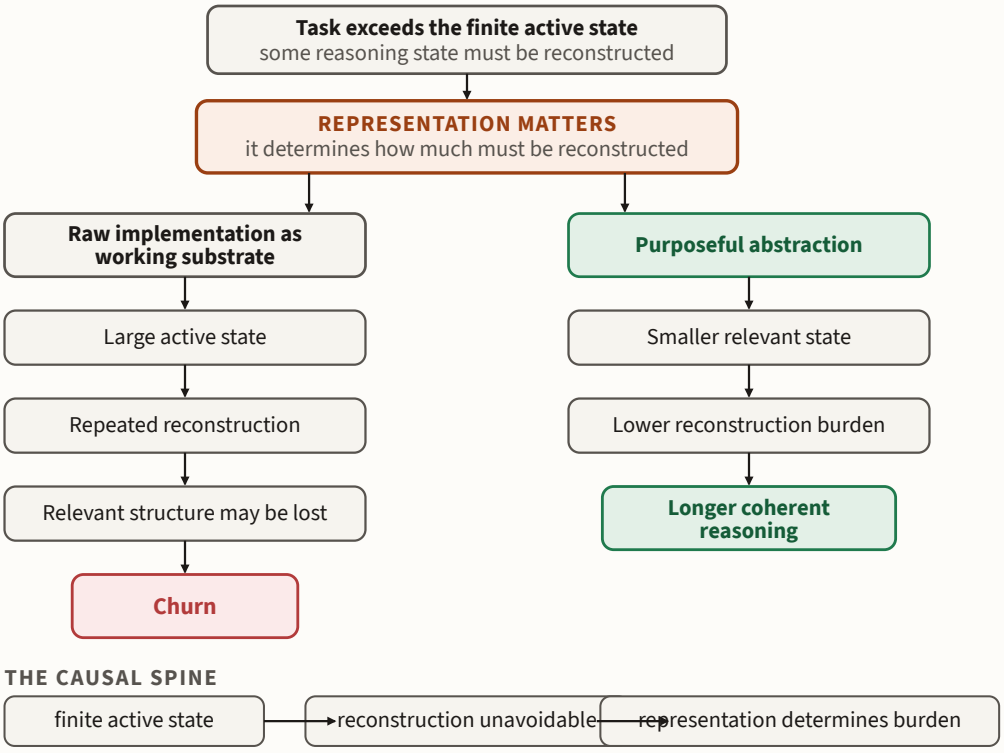


Figure 1.3-2: The reasoning horizon. *At sufficient scale, a task requires more state than a finite reasoner can keep active, so some state must be reconstructed. Representation determines the cost: raw implementation carries irrelevant detail, while purposeful abstraction can keep question-relevant state smaller and cheaper to recover.*

1.3.3 Probabilistic Work Needs External Authority

The second problem is not that agents make mistakes while human programmers do not. Software engineering has always separated authorship from assurance. We compile code instead of trusting the programmer's prediction that it compiles; we run tests instead of accepting an explanation that the program should work; consequential systems add review, static analysis, formal methods, operational evidence, or independent approval because the producer's confidence is not enough.

Cheap, fast, repeatable agentic production makes that separation more consequential. An agent can propose a change, inspect it, explain it, and often repair it, but another probabilistic judgment from the same productive substrate does not automatically create engineering

authority. If an obligation matters, the environment must establish what evidence counts and what follows when the obligation is not met.

Commodity intelligence is probabilistic in an engineering sense. Contemporary foundation models are learned systems whose outputs do not ordinarily carry guarantees that the engineering judgments embodied in those outputs are correct. Their present implementations may add uncertainty at several levels: generation can vary across samples; an underspecified request may admit several interpretations; intermediate reasoning and tool choices may differ; and an agentic task may compose many such judgments before producing a result. Harnesses can retry, decompose, vote, retrieve additional context, or ask another model to evaluate the first. All of these can improve reliability. None by itself turns an inferred engineering judgment into an engineering guarantee.

The details matter when evaluating a particular system, but they are too contingent to serve as the foundation of a general engineering method. Decoding can be deterministic. Errors can be correlated. One model or harness can be much better than another. Future foundation models may operate differently again. We therefore use a deliberately simple abstraction.

Suppose a realization depends on consequential judgments $J_1, ..., J_n$, and let p_i denote the probability that judgment J_i is resolved acceptably. Under the simplifying assumption that these events are independent, the probability that all of them are acceptable is the product

$$P(\text{all judgments acceptable}) = \prod_{i=1}^{n} p_i$$

The effect is small when each judgment is highly reliable and there are only a few of them. If each judgment has a 99% probability of being acceptable, for example, three such judgments give an overall probability of $0.99^3 \approx 97\ \%$. But the chance of success falls as the number of judgments grows: $0.99^{10} \approx 90\ \%$, and $0.99^{100} \approx 37\ \%$. It falls faster when individual judgments are less reliable: ten judgments that are each 95% reliable give $0.95^{10} \approx 60\ \%$. Even under this deliberately favorable model, many individually reliable judgments do not compose into a comparably reliable whole.

Many techniques can improve these probabilities, most notably running agents in iterative loops that critique, test, and revise their own work. Such techniques can make individual judgments substantially more reliable. The equation is illustrative, not a model of an actual agent: real judgments are not generally independent, their probabilities may be unknown, and iterative techniques can change those probabilities. But this simple model survives these complications. A consequential judgment entrusted to fallible inference retains residual risk, and adding more such judgments compounds that risk.

The consequential judgments that still depend on fallible per-instance inference form the system's **probabilistic surface**. Better models, prompts, context, tools, decomposition, and agent harnesses can make those judgments more reliable. Iterative loops can give the agent additional opportunities to detect and repair mistakes. These are valuable improvements, but they leave the obligation dependent on probabilistic judgment each time it arises.

This is the immediate reliability argument for moving consequential judgments out of repeated inference, but it is not the only argument for doing so. Future agents may make many such judgments cheaply and reliably. That will make for an exciting future, whether it takes one year or one hundred. But even granting that future, engineering disciplines require an independent basis for understanding what a system is intended to do, deciding which obligations govern it, and establishing why its behavior should be trusted. Part VII returns to that problem as one of engineering oversight and authority.

1.3.4 Authority Outside the Producing Run

Software engineering offers another move. For selected obligations, the environment can make the required property explicit and enforce it independently of the producing run. Instead of asking an agent to remember an architectural restriction every time it edits the system, for example, the environment may be able to represent that restriction and reject changes that violate it. The judgment that established the obligation can then be reused across later acts of implementation.

Agentic work provides places for such authority to act. Consequential work reaches the engineering system through identifiable interfaces: inputs, tool calls, edits, outputs, and lifecycle transitions. A sandbox can deny network access, a hook can reject a command, and a test can block a regression without requiring a rich system model. Explicit representations can extend the semantic reach of those same control points, allowing an action to be judged against architectural, behavioral, policy, ownership, or other system-level obligations.

The harness therefore supplies control points; representation determines what those control points can know. MAGE develops principled ways to use both while leaving the rest of realization open.

1.4 Two Problems the Method Must Solve

The properties of agentic machinery leave two engineering problems. A finite reasoner needs representations that keep consequential system state tractable as systems and tasks grow. Fallible autonomous work needs authority outside the producing run so that important obligations do not depend on the producer resolving every consequential judgment correctly.

These problems are related but distinct:

1. **The Representation Problem.** Large systems contain far more detail than any one engineering question needs, while the active state available to a reasoner is finite. How should a large, evolving system be represented so that the properties relevant to an engineering question can be reasoned about without continually reconstructing them from implementation? Part II develops MAGE's answer: Modeling.
2. **The Authority Problem.** Autonomous work can be constrained and checked at the interfaces through which it acts, but the producing run cannot be the final authority for its own consequential judgments. How should engineering intent become authoritative over autonomous work, so that satisfying important obligations does not depend on the producing agent remembering, agreeing, or self-certifying each time? Part III develops MAGE's answer: Alignment.

The two activities complement one another. Modeling makes selected system properties explicit enough to reason about; Alignment gives selected obligations authority over realization. Neither requires the other in every case: useful representations need not become constraints, and some local controls can act without a rich system model. Together, however, they let engineering knowledge and judgment persist outside any one producing run. Figure 1.4-1 traces that derivation.

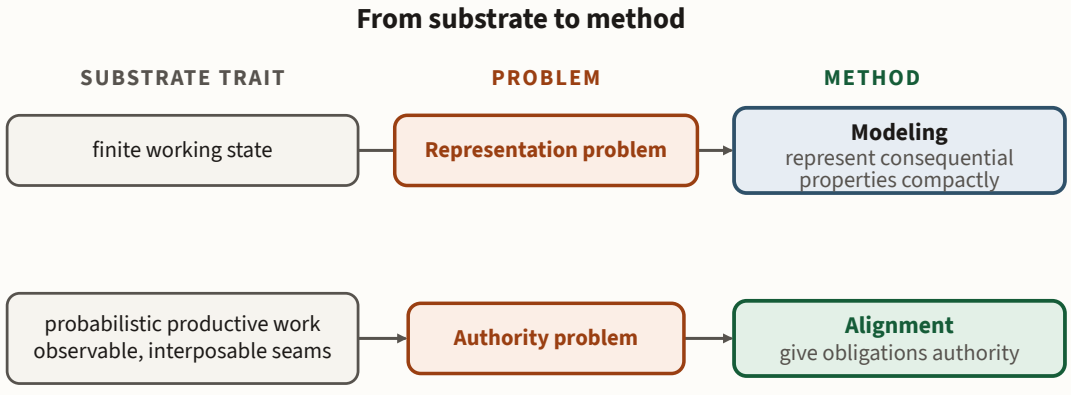


Figure 1.4-1: From substrate to method. *Finite reasoning state makes representation consequential; probabilistic autonomous work makes external authority consequential. Parts II and III develop MAGE's corresponding answers: Modeling and Alignment.*

Summary

Commodity intelligence changes the software-engineering problem because implementation capacity can grow faster than the engineering judgment required to direct and evaluate it. The relevant unit is therefore not the agent alone, but the larger engineered system through which autonomous work is represented, constrained, observed, evaluated, and accepted.

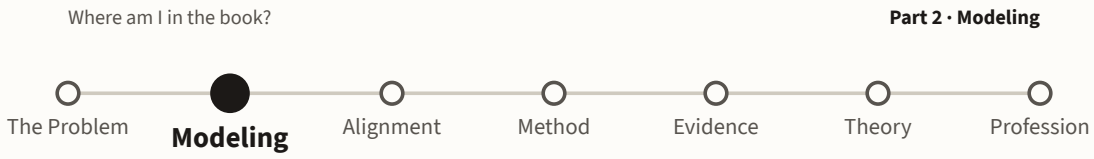
Two problems follow. The reasoning-horizon problem concerns what the reasoner can know and reconstruct while performing a task. Important system knowledge left implicit in implementation must repeatedly be recovered through finite and fallible reasoning. The probabilistic problem concerns consequential judgments left to probabilistic realization. Even a highly reliable decision can eventually fail when it is repeated often enough.

MAGE addresses these problems through two complementary moves. Modeling externalizes selected engineering knowledge and intent into representations suited to the questions engineers and agents must answer. Alignment gives selected obligations authority through mechanisms that can constrain actions, produce and evaluate evidence, and control consequences.

Neither move eliminates engineering judgment. Models preserve selected distinctions rather than representing everything, and not every obligation can or should become mechanically authoritative. The engineering task is to decide what should be explicit, what should bind realization, what should remain open, and what evidence is sufficient for the consequence at stake.

Parts II and III develop those two problems separately. Part II asks what should be modeled. Part III asks how selected engineering obligations acquire authority.

Part 2: Modeling



QUESTION THIS PART ANSWERS

How do I identify useful models?

MODELING PRINCIPLE

Externalize engineering knowledge and intent into explicit, structured models that both engineers and agents can reason through.

Appropriate representations make broader engineering questions tractable and make additional properties available for Alignment.

CARRYING FORWARD

Commodity intelligence · The Printer · Model · Reasoning horizon

NEW HERE

Modeling Principle · Traceability · Invariant · Model drift · Drift Gate

Software engineering has always relied on abstraction. Large systems exceed what any engineer can reason about directly, so engineers understand them through purposeful views. A dependency graph exposes relationships among components; a state machine exposes legal behavior; a schema exposes permitted structure; a quantitative model exposes a resource bound. Each representation preserves what an engineering question needs and suppresses what it does not.

Explicit models have nevertheless remained secondary in much code-centric software practice because another representation costs work to create, maintain, and reconcile with a changing system. Commodity intelligence changes that economics. Agents can increasingly derive, regenerate, reconcile, and query structured representations cheaply. At the same time, rapid autonomous implementation increases their value: engineering knowledge that once needed to be reconstructed occasionally may otherwise be reconstructed across many tasks and fresh reasoning states.

MAGE therefore treats Modeling as an engineering activity in its own right:

What should I model, and what will the model let me know?

Which model is useful depends on what the engineer needs to know. A concurrency question may require ownership and lifecycle; an architectural-boundary question may require components and permitted communication edges. Different questions about the same system therefore call for different reductions.

This Part develops six broad classes of engineering question:

- **Structural** — what parts exist, and which may depend on which.
- **Behavioral** — what states a thing occupies, and how it moves between them.
- **Ownership** — who controls a unit of work, and for how long.
- **Decision** — what is allowed, or which alternative should be selected.
- **Measurement** — what quantities the system holds, and against what bound.
- **Provenance** — what happened to an artifact, and what evidence records it.

These classes organize recurring engineering questions rather than partitioning systems. They overlap, and they are not exhaustive. The same worker may appear as a component in one model, an actor in a lifecycle in another, and the owner of work in a third. Architectural reasoning may traverse several such views at once.

DocAble is the running example—the production accessibility service introduced in Part I. A document enters, remediation is distributed across workers and services, the result is validated, and a corrected document returns with a record of what changed. The real system is far more complicated than any representation ahead. That is the point. Each model keeps only the relationships its question needs.

Each chapter begins with a familiar engineering representation, the properties it makes expressible, and the analyses it supports, then specializes that representation to DocAble. Four terms stay distinct throughout the Part. A **property** is a claim that can be expressed over a model. An **invariant** is a property required to hold over a declared domain. An **analy-**

sis or check produces evidence about a property. Whether the engineered environment enforces the resulting obligation is a separate question of **authority**, taken up in Part III.

> **Four questions for every model**
>
> Every model section ahead opens on the same four lines:
>
> - **Engineering question** — what am I trying to know or decide?
> - **Model** — what representation makes the question tractable?
> - **Property** — what can I now state precisely?
> - **Quality attribute** — what engineering concern does that property serve?

Part III adds a fifth question: **what gives the property authority?** Modeling makes properties explicit; Alignment makes selected obligations enforceable.

The final chapter, System Knowledge, shows how the six models connect without becoming a seventh model.

2.1 From Context to Engineering Models

A foundation model knows programming languages, frameworks, databases, distributed systems, cloud infrastructure, testing practices, security concepts, and common architectural patterns. It does not arrive knowing the particular system in front of it: its boundaries, history, conventions, ownership, or current engineering state. That knowledge is distributed across source, configuration, documentation, registries, history, and the people who maintain the system. An agent can reconstruct what a task needs from those artifacts, but reconstruction consumes reasoning and is easily repeated across tasks and reasoning states.

The first **Modeling move** is therefore to make reusable engineering knowledge explicit. Instead of asking each reasoner to recover the relevant structure from raw artifacts, preserve the entities and relationships that recurring engineering questions depend on. A task can then retrieve the connected slice it needs rather than reconstructing that slice anew.

The Modeling Principle: externalize engineering knowledge and intent into explicit, structured models that both engineers and agents can reason through. *In other words:* once a system exceeds what a reasoner can usefully hold in view, give recurring engineering questions purposeful representations.

Software engineering has long treated the externalization of intent as an engineering problem. Requirements engineering, for example, distinguishes stakeholder goals and environmental assumptions from specifications used to constrain a realization, and develops representations through which those claims can be analyzed, refined, and traced.[19–21] Important requirements may remain qualitative, incomplete, or dependent on assumptions about the environment. The inheritance MAGE takes from this tradition is the more basic move: make consequential engineering knowledge into an explicit object through which engineering can reason. Part III takes up the separate question of which obligations the engineered environment should enforce.

This Part begins with externalized context and proceeds toward representations with richer semantics, explicit properties, and maintained correspondence to the system. This chapter establishes the underlying modeling move: what counts as a model, what makes one useful, and how an engineering question determines the representation worth carrying.

2.1.1 Modeling Is a Classical Engineering Move

Modeling is not new. Engineering disciplines have long used representations that suppress irrelevant detail so that particular properties of a system can be reasoned about directly. Software engineering inherited and developed the same practice. Architectural descriptions expose components and relations; schemas expose permitted data structures; state machines expose states and transitions; interface specifications expose contracts across boundaries; formal models expose properties precisely enough for mathematical or mechanical reasoning. Model-based systems engineering makes this commitment particularly

explicit: engineering knowledge is organized around models that support analysis, communication, design, and assurance across the system lifecycle.[22,23]

The representations differ substantially in purpose and rigor, but they share a basic engineering move: choose a form in which the property of interest becomes easier to express, inspect, or analyze than it is in the realized artifact itself. The value lies in the reduction. A useful model preserves the distinctions required by its question and leaves the rest behind.

MAGE applies this classical move under different economic conditions. Commodity intelligence makes structured representations cheaper to construct, maintain, translate, and consume. Generative implementation simultaneously increases their leverage. As realization accelerates, engineering knowledge about what should be built, how the system is organized, and which properties matter must remain available across an increasing volume of autonomous work.

For readers familiar with model-based systems and software engineering (MBSE), much of the repertoire in this Part will therefore be familiar. MAGE's concern is what happens when these established representations become part of the engineered environment surrounding autonomous realization: agents can reason through them, engineering knowledge can persist across reasoning episodes, and selected properties can later become objects of Alignment.[8]

Models also participate in architectural reasoning. They expose relationships and make selected consequences available for analysis; engineers decide the decomposition, boundaries, interactions, allocations, and tradeoffs the system should embody. A model may record an architectural decision already made or reveal that the architecture should change.

Inset — Models, views, and software architecture SOFTWARE ARCHITECTURE

MAGE follows Fairbanks's definition of software architecture as the set of structures needed to reason about a system, comprising software elements, relations among them, and properties of both.[24] Engineers therefore reason about one system through multiple views: a component view may expose decomposition, a behavioral view legal transitions, a deployment view allocation, and a quantitative view a resource bound. No one view need contain everything another does.

Architecture consequently cuts across the model classes in MAGE. An architectural decision may depend on structural, behavioral, ownership, decision, measurement, or provenance relationships together. The useful views follow from the engineering questions that must be answered.

[8] Readers with a background in model-based systems engineering may find portions of Part II review. The distinctive concerns for MAGE are the use of models to bound consequential properties while deliberately preserving realization degrees of freedom; the separate Alignment problem created when realization is delegated to a probabilistic reasoner that does not itself bear engineering responsibility; and the economic question of when explicit representations, their maintained correspondences, and the mechanisms built over them repay their carrying cost. Parts III and VI develop the latter two concerns in detail.

> Software makes the relationship between model and realization unusually fluid because both are made of information. A schema can generate code; a state machine can drive behavior; a policy model can generate configuration; a dependency graph can become an execution plan. Such a representation can participate directly in the system while remaining selective about what it represents.
>
> When structured views are cheap to derive, maintain, connect, query, analyze, and sometimes execute, more of the knowledge required for architectural reasoning can remain explicit rather than being repeatedly reconstructed from implementation. Modeling names that MAGE activity.

A useful model need not be elaborate. Sometimes it is three states and two legal transitions; sometimes a list of permitted service edges; sometimes a schema joined on stable identifiers. The question is never *how much can we model?* It is *what representation makes this recurring engineering question easier to answer?*

2.1.2 A Model Is a Purposeful Reduction

A **model** is a purposeful reduction of a system. It preserves the information one engineering question needs and suppresses the detail that question does not. Its value is not completeness but selective omission. A reduction an engineer or agent can reason through is more useful than a faithful copy too large to hold in view.

Purposeful reduction is therefore not only compression. It can change the difficulty and character of reasoning. A property that requires semantic reconstruction over thousands of implementation details may become a relation over a small graph, a transition over a finite state space, or a bound over a measured quantity. The representation does not make the answer true. It can make the question tractable. Nor does it make the design decision: it makes selected consequences of that decision easier to reason about.

This is the connection Modeling shares with representation learning: what a reasoner can infer efficiently depends partly on the form in which the problem is presented. MAGE begins from representations engineering has already developed for consequential system questions rather than asking a learned reasoner to rediscover those abstractions on every task.

Tolerance and degrees of freedom

The suppressed detail has a realization-side consequence. Engineering obligations rarely determine a single acceptable realization. They constrain some variation and leave other choices open. This book uses two engineering terms for that distinction. A tolerance bounds the variation that an obligation permits; a degree of freedom is a realization choice left open because the governing obligations do not distinguish among its acceptable alternatives. Purposeful reduction therefore does two things at once: it represents the distinctions the engineering question requires while declining to settle choices that do not matter to that

question. MAGE does not seek to close those degrees of freedom merely because they could be specified.

In software engineering, a tolerance can be numeric but is often qualitative. A latency budget may set a numerical bound: a response must complete within 200 ms. Coupling admits an intermediate form: an architecture may tolerate dependencies up to a measured threshold while also distinguishing kinds of coupling that are acceptable or prohibited. Other architectural tolerances are primarily qualitative: one component may depend on another through a declared interface but not reach directly into its implementation. The difference partly follows from the medium. Physical engineering often concerns continuous quantities such as length, voltage, pressure, or temperature, so tolerances naturally take numerical form. Software structures and behaviors are largely discrete—types, sets, relations, graphs, states, and transitions—so tolerances often define sets of acceptable realizations instead.

Software's medium compounds this: because programs are constrained less by physical material than by the information they must accept, produce, and preserve, different implementations can satisfy the same obligations, so the space of acceptable realizations is often large.[25] Engineering need not choose one point in that space merely because it could. It can instead specify the properties that distinguish acceptable from unacceptable realizations and leave the remaining choices open. Commodity intelligence changes the economics of exercising that freedom: once realization becomes cheap, an autonomous reasoner can construct and revise implementations inside a space that would previously have been expensive for human engineers to explore. Modeling identifies the consequential distinctions; Alignment can enforce selected boundaries. The rest remains available to realization.

Figure 2.1-1 draws the relationship: obligations bound a region without selecting a point inside it.

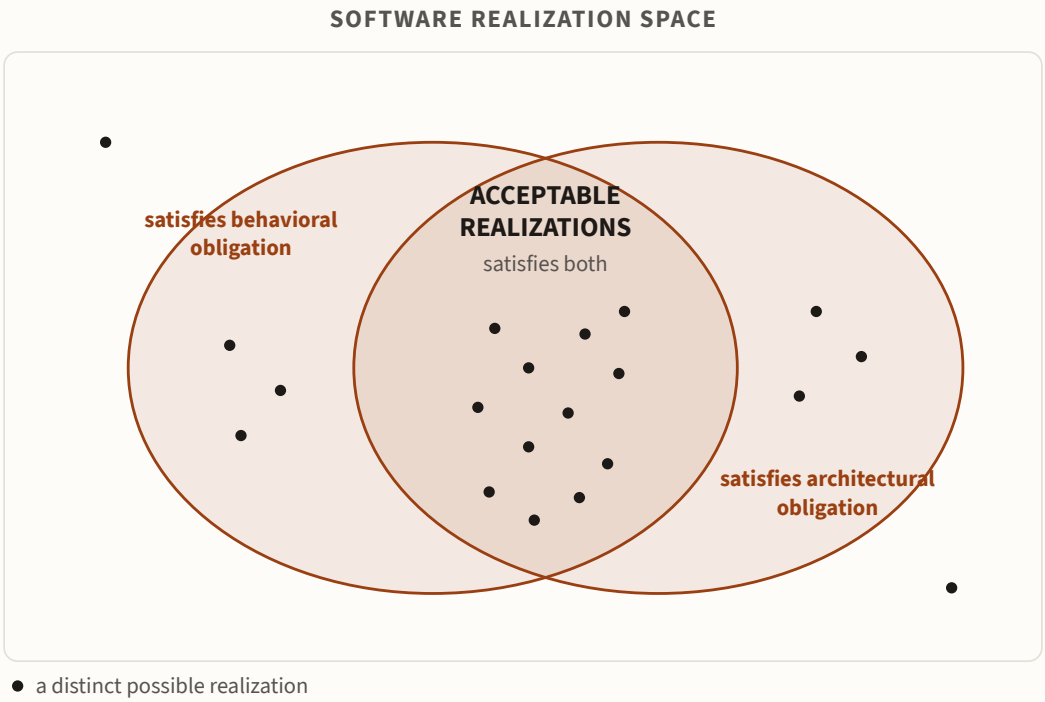


Figure 2.1-1: Obligations bound a realization space without selecting one realization. *Each region contains implementations satisfying one engineering obligation; their intersection contains implementations satisfying both. The points within that intersection may differ substantially in structure and implementation while remaining acceptable under the represented obligations. Those remaining choices are realization degrees of freedom.*

Not every unmodeled choice is free

This also distinguishes tacit obligations from genuinely free choices. An unmodeled choice can look open even when engineers would reject some realizations of it. Then the choice was never wholly free. Human judgment was already imposing a boundary. What was missing was an explicit representation of the obligation or its tolerance. A genuinely free choice is different. Once the governing obligations are accounted for, engineering has no reason to distinguish among the remaining alternatives. A useful diagnostic follows: if a realization would provoke *not that*, some consequential boundary remains, whether or not anyone has written it down.

An unmodeled choice can therefore mean several things. The consequence may not yet be understood; an obligation may remain tacit; the obligation may be explicit but not represented; or the choice may genuinely be free. These states can change as engineers learn from the system. A region initially treated as free may expose recurring costs or interactions that reveal an obligation; another may be investigated and turn out to contain no consequential distinction at all.

Inset — Uncertainty and unknown unknowns ENGINEERING

Engineering distinguishes a choice deliberately left open from one whose consequences are not yet understood. The first is a degree of freedom. The second is uncertainty. In the harder case—sometimes called an unknown unknown—engineering has not yet recognized the consequential property, interaction, or failure well enough to ask the right question about it. An unmodeled region therefore should not be assumed to be free merely because no obligation has been stated.

A model cannot directly represent what engineering has not yet recognized. It can, however, make uncertainty more inspectable. By reducing a complicated system to selected entities, relations, states, quantities, or alternatives, a model can make variation and interaction easier to compare and trace. Unexpected behavior can then expose distinctions that were difficult to see in the realized system directly. A degree of freedom is known freedom; uncertainty is not.

This is one reason modeling can precede specification. Sometimes engineers construct a model because they know which property must hold. Sometimes they construct one to discover which properties matter.

Modeling does not require complete specification. Engineers can model what is known while leaving uncertainty visible and genuine freedom unconstrained. As understanding improves, the model can change with it.

The question is the design input. Ask whether two workers can process the same job at once, and the color of the web interface falls away; you need who can own work, how ownership is acquired, when it expires, and what happens after a crash. Ask instead whether one service may bypass another, and the ownership details fall away; now you need components, communication edges, and the seams a call must pass through. The system did not change. The question did. The useful reduction is not always obvious at first. Sometimes you have to look at the problem differently to see which relationships matter.

Hold the system constant. Change the engineering question. Watch the model change.

There is no single "model of the system," only reductions built to expose particular properties. A model earns its place by the question it settles, not by how much of the system it represents.

This idea also has a long history in artificial intelligence. Knowledge-representation research treats a representation as more than stored information: its form embodies commitments about what distinctions matter and affects what kinds of inference are convenient or possible.[26] MAGE applies the same principle to engineering representations. A dependency graph, state machine, contract, or measurement model is useful not because it records more

of the system, but because its chosen structure makes a particular engineering question cheaper to answer.

A representation remains a purposeful reduction even when it is executable. Some models in this Part participate directly in execution or analysis: machines query them, derive artifacts from them, generate from them, or check properties over them. Software can therefore make the distance between architectural representation and realization unusually small. Executability changes what a model can do; it does not make the model a complete representation of the system. That is why the examples ahead are drawn as reductions before their implementations are discussed: the abstraction should remain visible apart from its encoding.

2.1.3 Structure Makes Representations Machine-Operable

Externalizing knowledge saves reconstruction only if a machine can read it back. Machines can read externalized knowledge more reliably when its important entities and relations are **structured** — given a declared shape — rather than recoverable only from prose. Schemas, registries, graphs, and typed records expose entities and relations that tools can traverse, validate, and join reliably.

A common useful form is a graph of engineering entities and relations. Its nodes are engineering entities: services, modules, endpoints, queues, databases, requirements, tests, owners. Its edges name the relations: calls, depends on, deployed to, owned by, verified by. The graph need not live in a graph database. YAML, registries, and schemas joined on stable identifiers can represent the same structure. The word *graph* names the structure, not the storage. Relevance then follows relationships: start at the entity a task names, and its neighbors come with it, so the agent reasons over a connected slice instead of a document dump.

Many of these representations can be interpreted as graphs of entities and relations. Their shape is not what distinguishes them; their semantics are. A dependency edge, a transition, an ownership claim, and a permission can all be drawn as arrows while answering different engineering questions. The representation becomes useful through what those entities and relations mean.

Externalizing knowledge this way is the lightest form of Modeling, and much of current practice already stands on it. Spotify's public account describes a retrievable service catalog. Cloudflare's describes standards externalized under stable identifiers. Shopify's describes a different form, in which agent sessions are mined into reusable skills. DocAble assembles its graph from registries, schemas, and YAML joined on stable identifiers. Each makes selected engineering knowledge durable outside a single reasoning episode, allowing later agents to retrieve rather than reconstruct it.

2.1.4 Representations Support Accumulating Capabilities

Structured representations support different capabilities: externalized knowledge, typed relationships, explicit semantics, stated properties, derivation, generation, traceability, and checking. A representation may acquire additional capabilities as engineers ask more of it, but these are not a required sequence or maturity model.

The practical rule is selective modeling. Model facts that are repeatedly reconstructed and properties that are repeatedly adjudicated. A service catalogue may end repeated ownership discovery; an invariant may retire a recurring class of judgment. Add structure when the resulting reduction in recurring engineering work justifies carrying it.

Authority remains separate. A rich model may remain advisory, while permissions, sandboxes, compilers, or tests may enforce obligations with little explicit system-level modeling. Modeling expands the set of properties that can be stated and evaluated; Part III asks which the engineered environment should enforce (Figure 2.1-2).

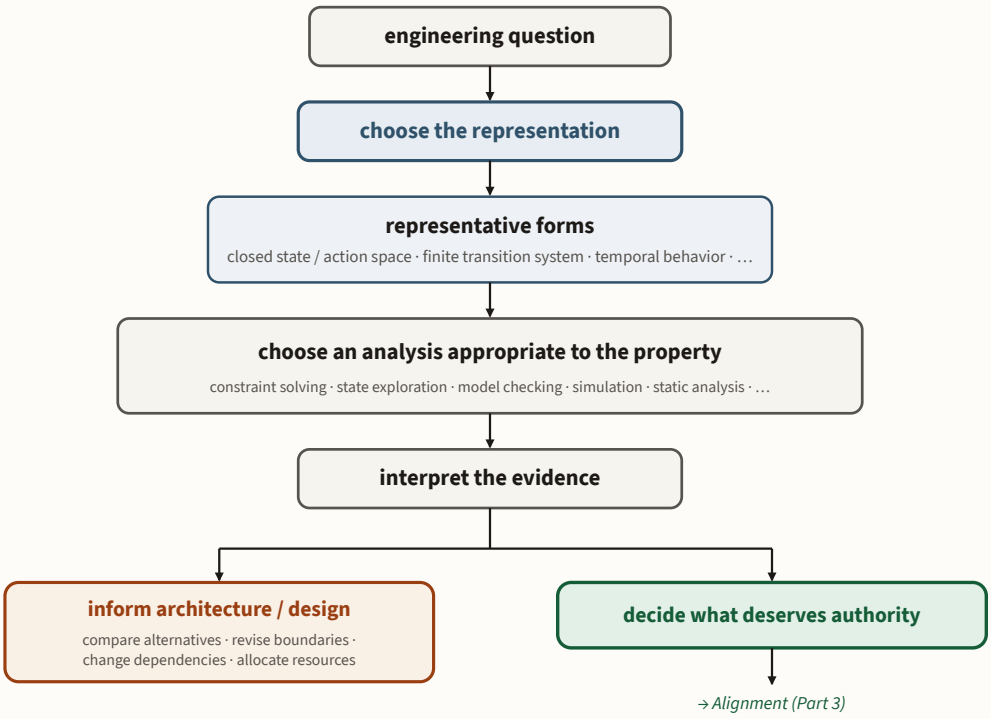


Figure 2.1-2: From engineering question to engineering consequence. *Hold the system fixed and change the question, and the useful representation changes with it. The representation determines which analyses become tractable; analysis produces evidence that can inform architecture and design, expose a property for checking, or both. Part III takes up the separate question of which obligations should be enforced.*

2.1.5 Six Classes of Engineering Question

> MAGE uses six model classes as a working classification. The classes are organized by the engineering question a model helps answer, not by notation or implementation form. They are useful rather than exhaustive, and they can overlap: one represented object may participate in several classes. The classification is a working engineering repertoire, not a claim that these six classes form a minimal basis for engineering reasoning; Part VI reopens that question.

The six classes ahead differ less in shape than in semantics. At an abstract level, each can be understood as a graph: a set of things and relationships among them. A structural model might use nodes for components and edges for dependencies. A behavioral model can use nodes for states and edges for transitions. An ownership model relates work, owners, and claims; a decision model relates alternatives through permitted choices; measurement and provenance models likewise connect quantities or events through meaningful relations. The important difference is not that one is a graph and another is not. It is what the nodes and edges mean, and therefore what questions the representation can answer.

This is why MAGE classifies models by engineering question rather than notation. The same graph-shaped representation can answer different questions depending on its semantics, and one system fact can participate in several views. Architectural reasoning likewise crosses the classes: one design decision may require several views because it affects several engineering concerns.

Figure 2.1-3 sets out a portion of that repertoire: nine familiar representations, read as a set of moves rather than a taxonomy of all modeling. MAGE prescribes no new notation, and the repertoire is larger than the six classes the rest of this Part organizes.

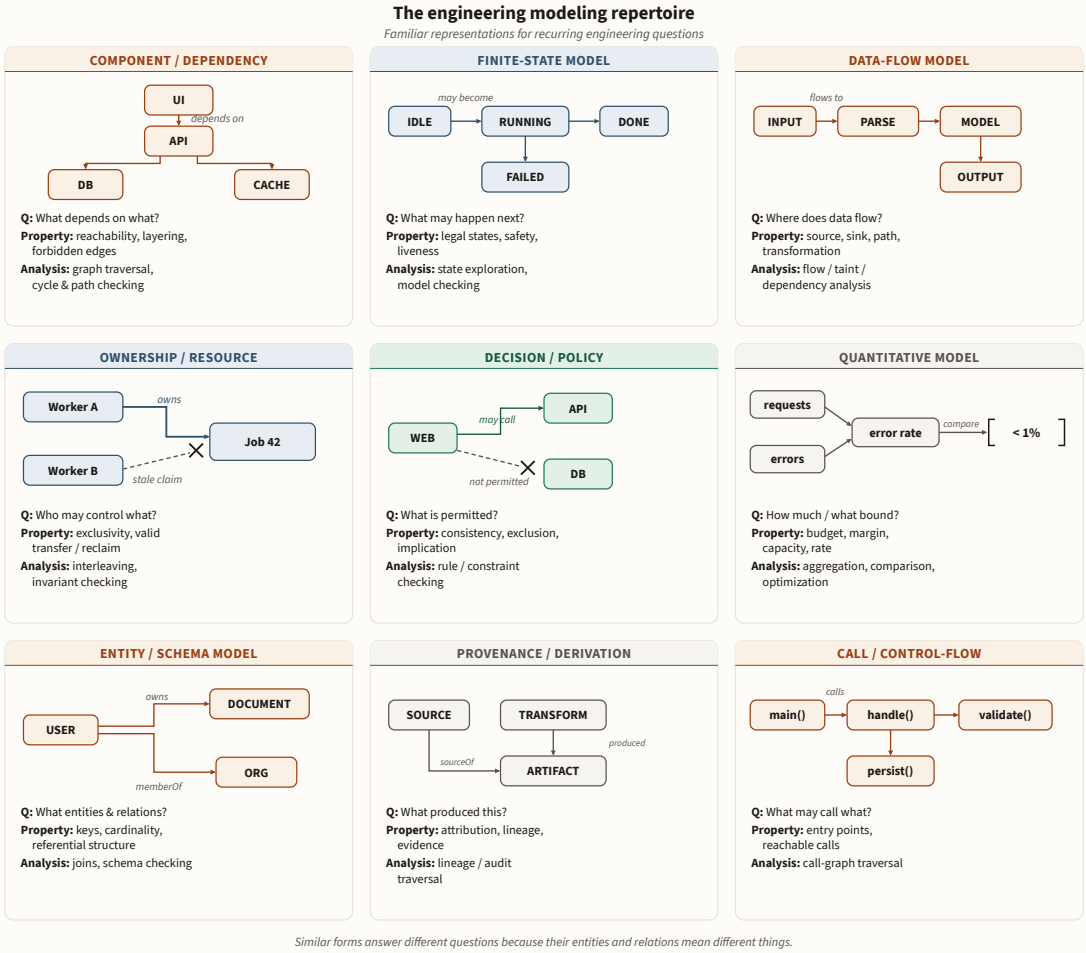


Figure 2.1-3: The engineering modeling repertoire. *Familiar representations preserve different relationships for different engineering questions. Similar graphical forms can carry different semantics: an arrow may represent a dependency, transition, flow, permission, ownership relation, or derivation.*

Inset — Reuse the engineering repertoire

ENGINEERING

This Part does not attempt to enumerate every useful model or every consequential engineering question. Its purpose is to draw attention to the fact that your system has consequential questions, and that models provide a means of articulating those questions in representations that give engineers oversight and agents a succinct form in which to reason about them. The examples ahead illustrate that practice rather than exhaust it.

> But you are an engineer. Take this not as an invitation to invent a new modeling technique for every problem, but as a reminder to study the models, patterns, and analyses your field has already developed. Once the engineering question is known, start with the established repertoire. State machines capture legal transitions; dependency and data-flow graphs support reachability questions; schemas capture entities and relations; queueing and resource models expose capacity and contention; decision tables and constraint systems expose permitted combinations. Each comes with known semantics, applicable analyses, strengths, and tradeoffs.
>
> This is much of what it means to be an engineer. An engineer who needs to sort data does not ordinarily invent a sorting algorithm from first principles. The wise engineer recognizes the problem, knows the established algorithms, and selects among them according to the properties and tradeoffs that matter. Modeling should work the same way. The important judgment is often recognizing the engineering question and selecting a representation that preserves the properties needed to answer it. Occasionally the established repertoire is inadequate and invention is required. Recognizing when that is true, and devising the new abstraction, is precisely the kind of judgment for which engineers are valuable.
>
> There are books full of this accumulated knowledge. Software architecture, model-based systems engineering, formal methods, databases, distributed systems, operations research, performance engineering, and other fields document modeling techniques and the conditions under which they work. MAGE does not reproduce those catalogs. It provides a framework for deciding when such knowledge should become an explicit part of the engineered environment. A governed engineering environment can then make the relevant repertoire—or an approved subset of it—available directly to agents. Appendix E shows one realization: DocAble's self-governance skill packages the MAGE engineering method for agents, including a modeling repertoire and modeling moves. The agent can recognize a situation, choose an appropriate Modeling or Alignment move, act through the environment, and learn from the resulting evidence rather than inventing its engineering method afresh on each task. The repertoire is reusable engineering knowledge; choosing from it remains engineering judgment.

The six classes (Figure 2.1-4) name the questions the rest of the Part is built around:

- **Structural** — what exists, and how the parts relate?
- **Behavioral** — what may happen, and in what order?
- **Ownership** — who controls what, and for how long?
- **Decision** — what is allowed, or which alternative should be selected?
- **Measurement** — how much, and against what envelope?
- **Provenance** — what happened, and what evidence records it?

These are views, not boxes. The in-flight lease we reach in the ownership chapter supports Ownership, Behavioral, and Measurement models at once. The classification tells us which question we are asking, not where the representation belongs. A class suggests the questions to ask and the relationships worth preserving; it does not prescribe one notation, and a useful model may cross class boundaries.

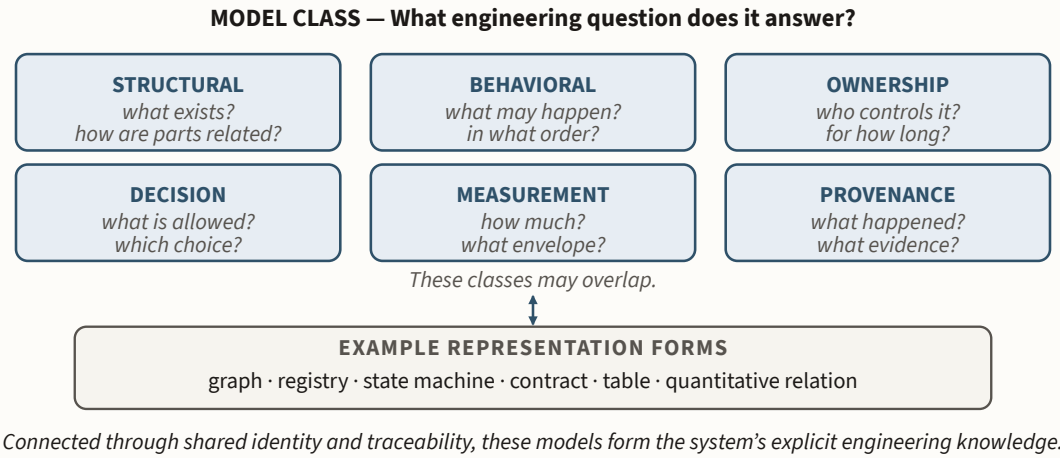


Figure 2.1-4: MAGE's working model ontology. *The six classes distinguish models by the engineering question they answer. Classes may overlap, and each may be encoded using several representation forms. Architecture is not a seventh class; architectural reasoning may traverse several views.*

System knowledge is not a seventh class. It is the connected substrate in which the six meet — where a service named in a structural model is the same service a decision model references, and a work item in an ownership model resolves to the measurements associated with it. Architectural reasoning can traverse that substrate without requiring one universal model of the architecture. The final chapter of the Part treats the substrate directly.

The classes expose different kinds of acceptable variation. A behavioral model can distinguish legal from illegal executions; a decision model can delimit a set of permitted alternatives; a measurement model can place a quantity against an envelope. Structural, ownership, and provenance models can likewise expose obligations that constrain some realizations while leaving others open. Tolerance names the permitted variation under an obligation; it is not a seventh class of model.

The model class names the engineering question; the **representation form** is a separate choice. Structural, behavioral, or ownership questions might all be represented as graphs, tables, schemas, or other structured forms. Purpose comes first; representation follows. The question determines which entities, relations, and properties the model must preserve — so represent shared facts once where practical, then project or join the view the question requires.

The chapters ahead apply this repertoire to six production models from DocAble: its document-processing architecture, recovery ordering, in-flight ownership, service-flow policy,

cost-and-capacity model, and attribution record. The final chapter then asks how such representations connect through shared identity without collapsing into one universal graph.

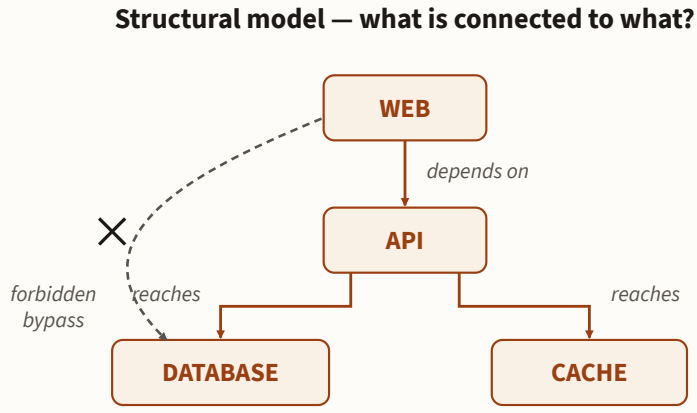
A model does not govern the system merely by existing. Modeling makes an engineering property explicit; Alignment makes selected properties enforceable.

2.2 Structural Models: What Is Connected to What?

Structural models hold a system still. They suppress most behavior and keep the entities, boundaries, containment, dependency, and flow — what parts exist and which rest on which. It is the most familiar reduction in engineering, and the natural place to begin.

2.2.1 The Canonical Move: Structure and Dependency

A component diagram names which services exist; a dependency graph names which modules rest on which; a call graph names which procedures may invoke which; a data-flow diagram names how information moves through transformations; a deployment topology names what is connected across execution locations. The graphical forms differ, but the reduction is the same: keep the entities and the declared relations, discard the behavior (Figure 2.2-1).



Question: What exists, and what is connected to what?
Semantics: node = an engineering entity; edge = a declared structural relation.
Typical analysis: reachability, path / cut analysis, cycle detection.

Figure 2.2-1: The structural move: entities and typed relations. *Nodes are the entities a question needs; edges are the declared relations between them, and the edge label carries the meaning. Its analyses are graph operations — reachability, path, and cycle checks.*

The model selects both the entities and the relations. A graph of service boundaries should not carry every helper function; if the question is about architecture, that detail only makes the model worse. And the edge label is where the engineering meaning lives. “A depends on B,” “A calls B,” and “a record is stored in a database” are different relations; collapsing them

into a generic “related-to” edge throws away exactly the distinction the model was built to preserve.

Once structure is explicit, the claims an engineer wants become graph properties. Some are local: component A may not depend directly on component C. Some concern paths: every route from ordinary logic to a format-specific library must cross a sanctioned seam. Some concern the whole graph: the dependency relation must be acyclic. The analyses follow — reachability, reverse traversal to find what depends on a node, cycle detection, impact analysis over a proposed change, and conformance of an observed graph against a declared one. A question that would otherwise mean reading imports, configuration, and deployment manifests becomes an operation over a declared graph.

Static analysis can sometimes derive the graph from the implementation; in other cases engineers declare the architecture as intent. Those are different claims — one describes what the system appears to do, the other what it is supposed to do — and Part III treats the distinction carefully. For now the move is smaller: choose the structural relation that makes the engineering question explicit. DocAble uses that move at two grains.

2.2.2 DocAble: Document-Mutation Structure

DocAble supports several structured document formats through libraries powerful enough to alter their underlying object structures. Those libraries are necessary, but they sit behind a deliberate architectural boundary: most remediation logic works through DocAble’s structured-document abstractions rather than reaching into a format-specific implementation directly.

Document-mutation model · Structural MODEL CARD

- **Engineering question** — Where is document mutation supposed to occur?
- **Model** — a layered structural relation: remediation logic operates through a structured-document model, realized by format-specific implementations for DOCX, PPTX, XLSX, and PDF.
- **Property** — the intended architecture places format-specific mutation behind one sanctioned seam; ordinary remediation logic is not intended to bypass it.
- **Quality attribute** — architectural integrity and modifiability.

The useful model is small. Figure 2.2-2 draws it: remediation logic operates *through* a structured-document model, which is *realized through* a format-specific implementation, which finally touches the concrete DOCX, PPTX, XLSX, and PDF formats. It says nothing about queues, retries, deployment, user sessions, or the mechanics of individual repairs. Those facts are real but irrelevant to the question; the reduction preserves only the architectural boundary that matters.

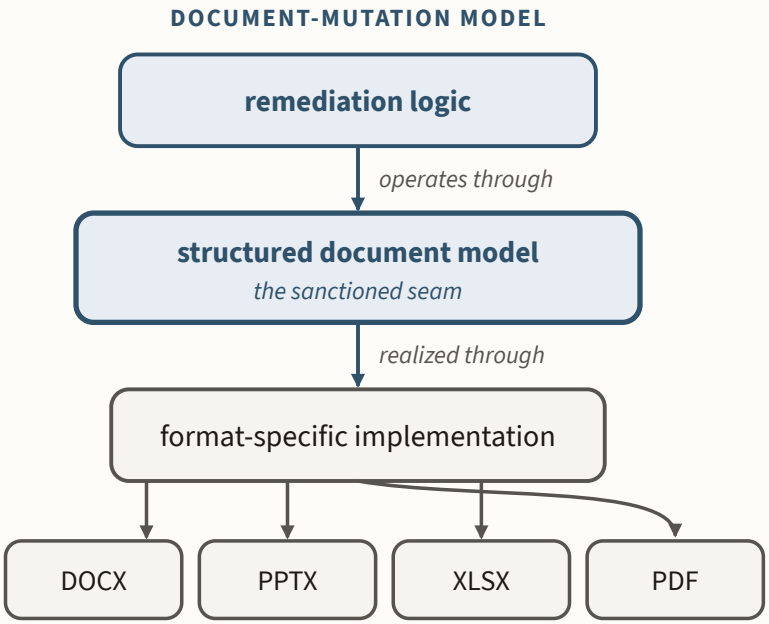


Figure 2.2-2: Document-mutation structural model. *The reduction preserves one architectural relation: ordinary remediation reaches format-specific mutation through a structured-document seam. Enforcement of that relation is a separate Alignment decision.*

The principal property is a path claim: every ordinary route to format-specific mutation crosses the seam, and no route reaches a raw format library directly. The model *states* that relation and exposes it to checking; it does not, by existing, enforce it. Part III returns to the same relation when a static control flags code that bypasses the seam. The model supplies the architectural claim; the mechanism gives that claim authority. This obligation is a **hard** one — a lint refuses the bypass — but the refusal belongs to Part III.

A structural model carries the opposite risk too. The code moves and the drawing does not, so the boundary it shows quietly stops matching the system. That gap between a model and the system it describes is **model drift**. Later sections make the correspondence explicit by deriving, generating, or tracing between model and implementation.

2.2.3 DocAble: Computation and Composition

The document-mutation model asked where one kind of change belongs. A second structural question reaches further inside the system: what computations make up remediation, and how do they depend on one another? DocAble runs a document through a collection of registered remediation passes. A pass may detect a condition, produce information a later pass uses, decide whether remediation should proceed, apply a bounded change to the intermediate representation, or edit a larger region directly. That structure exists in the implementation whether or not a model names it. The first reduction is the obvious one: treat each registered pass as a node, and suppress the statements, library calls, and loops inside it.

Remediation-computation model · Structural MODEL CARD

- **Engineering question** — What consequential computations make up remediation, and how does information move among them?
- **Model** — a static directed graph whose nodes are registered computations and whose typed edges represent declared composition among them.
- **Property** — declared data dependencies and control dependencies are distinguishable and consistent with pipeline ordering; each modeled computation also states how bounded its mutation is — a typed, comparable patch for all or a declared subset of its edits, or a direct in-place edit — and, on a separate axis, which apply seam a patch producer uses.
- **Quality attribute** — analyzability, modifiability, and architectural integrity.

DocAble's remediation-computation model represents registered computations as nodes and distinguishes two kinds of composition. A data-flow edge means one computation produces information another consumes in producing its result. A control edge means one computation produces information another consults only to decide whether it should run. Treating both as a generic dependency would obscure a consequential distinction: information that contributes to a result is different from information that controls whether the computation occurs. Figure 2.2-3 draws the resulting reduction.

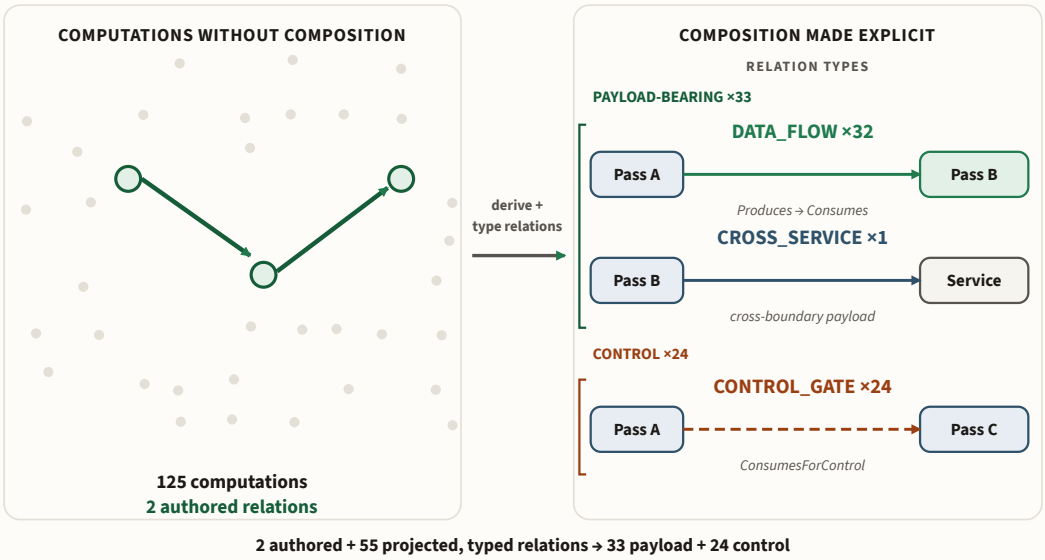


Figure 2.2-3: From computations to composition. *Typing the dependencies among registered computations separates payload-bearing relations — data-flow and cross-service — from control gates that a computation consults only to decide whether it runs.*

The same model also records how bounded each computation's effect is: some produce typed patches, some produce them for part of their work and edit directly for the rest, and most edit document state directly. These distinctions do not declare one form universally better. They make previously implicit differences available for engineering questions about composition and mutation.

The computation model remains static with respect to any one run: it records no per-job invocation history, retries, or model fingerprint. Other views can nevertheless attach observations to its stable computation identities. DocAble now joins staged-run telemetry to those identities, so measured latency and cost can be queried alongside the structural model without turning the structural graph into an execution log. The per-session mutation record remains separate: it records what happened to one artifact, not the execution history of the computation graph.

The same identity spine can support additional engineering views without enlarging the structural schema each time. Properties such as complexity, bounded execution, latency, test coverage, and checker trust can live in separate providers keyed by computation identity. Declared properties state engineering intent; measured properties report observation and may have incomplete coverage. An unmeasured latency is therefore unknown, not zero. The model family grows by joining views around stable identities rather than by turning one graph into a model of everything.

The next chapter adds time: what states a thing occupies, and the orderings it is allowed to move through.

2.3 Behavioral Models: What May Happen?

A structural model holds a system still; a behavioral model lets it move. It suppresses most of what a component *is* and keeps what it may *do* over time — the states a job may occupy, the transitions permitted between them, and the orderings those transitions allow or forbid. Where a dependency graph answers *what is connected to what?*, a behavioral model answers a question that only arises once time is admitted: what may happen, and in what order?

2.3.1 The Canonical Move: States, Transitions, and Executions

Engineering has represented behavior this way for a long time. A finite-state machine names the legal states of a system and the transitions between them; a labeled transition system adds the events that label those transitions; statecharts add hierarchy and concurrency so that large behaviors stay legible[27]; a temporal specification describes a property of whole executions rather than of any single state. These forms differ in expressiveness, but they share one reduction: they keep the states and the transitions, and they discard the code that implements each one.

That reduction is what makes behavior checkable. A job-processing service described only through its handlers, database writes, and retries hides the very question an engineer most needs to ask — can the system reach a state it should never occupy? The same service, reduced to a handful of states and the transitions between them, turns that question into one a person or a tool can actually answer (Figure 2.3-1).

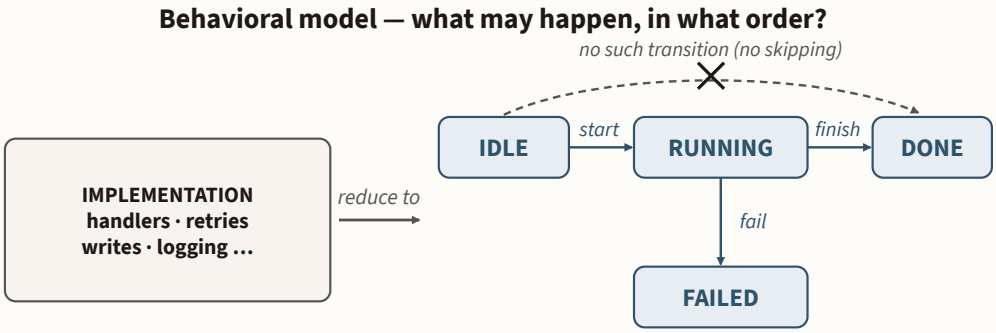


Figure 2.3-1: The behavioral move: states and transitions. *A model keeps the legal states and the permitted transitions and discards the code; the transition it omits — here the skip from IDLE straight to DONE — is the behavior it forbids. Its analyses search for a reachable bad state or a required event that never arrives.*

A node is a legal state; a directed edge is a permitted transition; a path through the graph is an execution the model admits. The key content is often negative: the transitions the model omits are exactly the behaviors it declares illegal. Once behavior is explicit, the questions form a progression from local to global — is this single transition legal? is this state reachable at all? can a legal execution reach a forbidden state? can the system deadlock? must some event eventually occur? The first are **safety** questions, which a single bad execution refutes; the last is a **liveness** question, which no finite good execution can establish, because the failure is the system running forever without the required event. A model can make either expressible, and the choice of check — transition validation, reachability search, invariant checking, or temporal model checking[28] — follows from the shape of the property rather than from a preference for one method.

A behavioral model does not make the system behave. It makes the behavior something one can reason about before it runs.

2.3.2 DocAble: Recovery Ordering

DocAble has one behavioral obligation whose violation is quiet and expensive. When a job falls back to a recovery path, the durable artifact and the database record that points at it must be written in the right order: the record may not name an artifact that has not yet been published. Reduced to its states, that concern is a small model.

Recovery-ordering model · Behavioral MODEL CARD

- **Engineering question** — In what order may recovery publish an artifact and update the record that refers to it?
- **Model** — a transition system over the recovery lifecycle; the decisive fact is a transition that does *not* exist.
- **Property** — the database record may refer to the fallback artifact only after that artifact has been durably published.
- **Quality attribute** — data integrity; crash-safety.

The recovery lifecycle runs through several states, but the obligation lives in two of them. An artifact is first written locally (`FALLBACK_STAGED`) and only later published to durable storage (`FALLBACK_UPLOADED`). The record may advance to `DB_UPDATED` only from the durable state. Figure 2.3-2 draws the ordering, and the fact that matters is the edge that does *not* exist: there is no direct `FALLBACK_STAGED → DB_UPDATED` transition, so the record can never point at a merely-staged, not-yet-durable artifact. Local staging and durable publication are different states, and the whole obligation rides on keeping them apart.

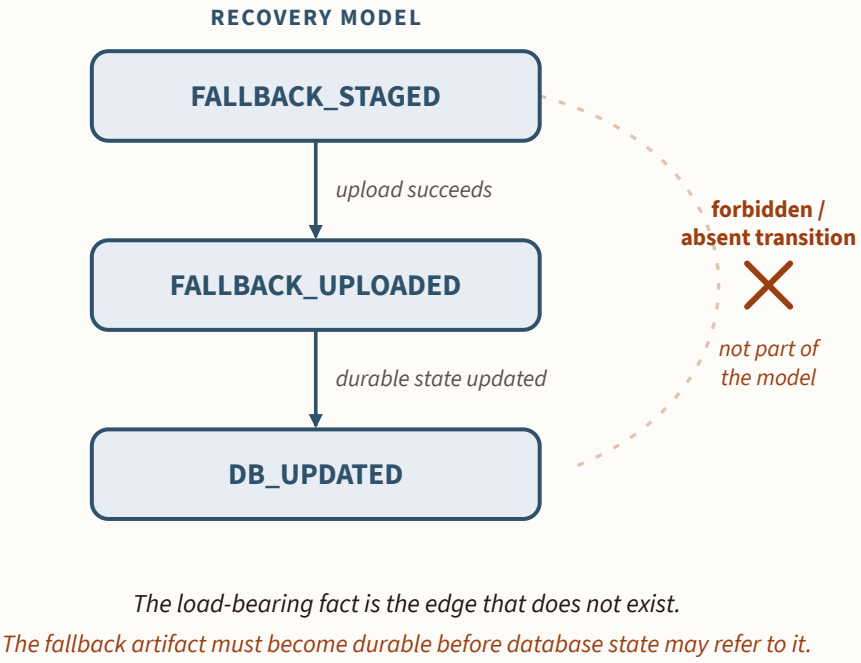


Figure 2.3-2: Recovery-ordering model. *The fallback artifact must become durable before database state may refer to it. The load-bearing fact is the absent transition from FALLBACK_STAGED directly to DB_UPDATED.*

Stated over the model, the property is a reachability claim: `DB_UPDATED` is reachable only through `FALLBACK_UPLOADED`. That is a property the model *exposes* and makes checkable — one can search the transition system for any path into `DB_UPDATED` that skips durable publication. Whether the running recovery path is *held* to the ordering — whether a transition table refuses the short-circuit step — is a question of authority, and it belongs to Part III. The two layers stay separate here: the model exposes the ordering; a mechanism enforces it. This obligation is a **hard** one — the machinery refuses the violation rather than merely reporting it — but the refusal is Part III's subject, not the model's.

2.3.3 What the Model Makes Checkable

The recovery model answers a safety question — can a bad state be reached? — and one counterexample would settle it. Not every behavioral obligation has that shape. "Every job eventually reaches a terminal state" is a liveness claim: no successful run proves the system can never stall, so the obligation ranges over infinite executions rather than reachable states, and it needs a different checker. Figure 2.3-3 places the three classes side by side.

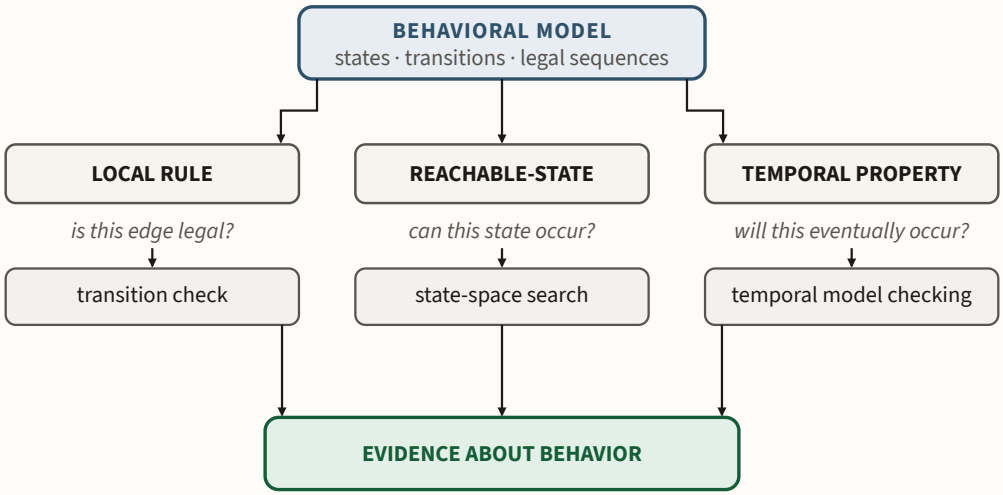


Figure 2.3-3: Behavioral models expose different classes of properties. *A transition relation can support local legality checks, reasoning over reachable states, or temporal claims over executions. The appropriate checker follows from the property; richer machinery is not automatically stronger for every question.*

The point is not that DocAble should apply every one of these techniques. It is that once behavior has an explicit representation, the right form of reasoning becomes available for each kind of property, and the checker follows from the property rather than from habit. The recovery ordering is itself a sharp slice of a larger behavioral model: a DocAble job carries a full lifecycle — queued, leased, remediating, awaiting-merge, completed or failed — with its own legal transitions. The full lifecycle answers broader workflow questions; the three-state recovery ordering is the sharper reduction for crash-safe publication.

The recovery-ordering model makes an ordering obligation explicit and searchable. It does not, by itself, establish that the implementation realizes the model, prove that this is the right ordering for the product, or decide what should happen when the ordering is violated. It gives engineering a precise object to check and a precise thing to align.

Structural models froze time; this chapter let it flow but assumed a single actor moving through states. The next question is what happens when several actors reach for the same work at once.

2.4 Ownership Models: Who Controls What, and When?

With one worker, *who owns this job?* has an obvious answer. Add many workers, a crash, and a sweep that reclaims abandoned work, and the same question acquires time and contention: ownership can be claimed, can expire, and can be reclaimed, and a holder that went away can return to find its claim gone. An ownership model keeps those relationships and discards the rest — who holds what, on what terms, and for how long.

2.4.1 The Canonical Move: Resource Ownership

Software engineering has a long vocabulary for control over a shared resource: locks and leases[29], resource-allocation relationships, capability and claim relationships, ownership transfer, and the lifecycle of a claim from acquisition through expiry to reclamation. Ownership is the family the earlier chapters most obviously touch. A claim relates a holder to a resource, which is structural; the claim is acquired, held, and released over time, which is behavioral. That overlap is why ownership is the natural place to show one representation carrying more than one class of question (Figure 2.4-1).

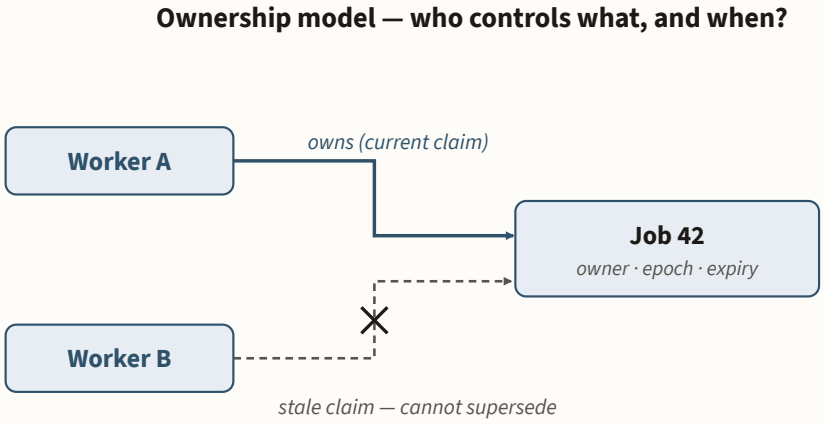


Figure 2.4-1: The ownership move: a claim over time. *A claim binds a holder to a resource and carries the generation and lifetime that tell a current claim from a superseded one; a second, stale claimant is exactly what the generation exists to reject. Its analyses explore concurrent interleavings for an invariant violation.*

A claim binds a holder to a resource and carries the facts that make contention decidable: which holder, which generation of claim, and when it expires. The generation is the fact a

bare “A owns X” edge cannot express — it distinguishes a *current* claim from a *superseded* one. The properties a designer wants to state follow from that: at most one valid holder at a time, a stale holder cannot affect the current one, a release or transfer is valid only from the current claim, and abandoned work is eventually reclaimed.

The hard failures do not live in any single transition. They live in the interleavings: the orders in which concurrent claims, expiries, and reclaims can occur. So the characteristic analysis is state-space exploration over those sequences, with an invariant checked at each reachable configuration. This is the behavioral toolkit of the previous chapter pointed at a concurrency question, which is again why ownership reads as an overlap family.

2.4.2 DocAble: In-Flight Ownership

DocAble leases each in-flight job to one worker. A lease carries an owner, a monotonically increasing generation, and a lifetime; when a lease looks abandoned it can be reclaimed and the job re-leased to another worker at the next generation.

In-flight lease · Ownership MODEL CARD

- **Engineering question** — Who currently owns this in-flight work, and can a returning stale owner disturb the current claim?
- **Model** — a claim relating a worker to a job, stamped with a generation and a lifetime; reclaim advances the generation.
- **Property** — a stale claim cannot clear or supersede the current one.
- **Quality attribute** — correctness under concurrency; crash-safety.

Start with what the model *means*. A job is leased exactly when it has an owner; the two are the same fact seen from two sides, so the model represents ownership as the presence of a current claim rather than as a separate flag that could disagree with it. That is a definition, not a property that could fail. It sets up the invariant that can.

The consequential invariant is about time and contention. Suppose worker A holds the claim at generation 7, appears to stall, and is reclaimed; the job is re-leased to worker B at generation 8. If A now wakes and tries to release using its generation-7 claim, that release must not clear B’s current generation-8 claim. The generation makes the distinction decidable: a release carrying an old generation names a claim that no longer owns, so it is inapplicable to the current one.

Figure 2.4-2 draws the claim and its lifetime; the generation is the small addition that separates a live claim from a stale one.

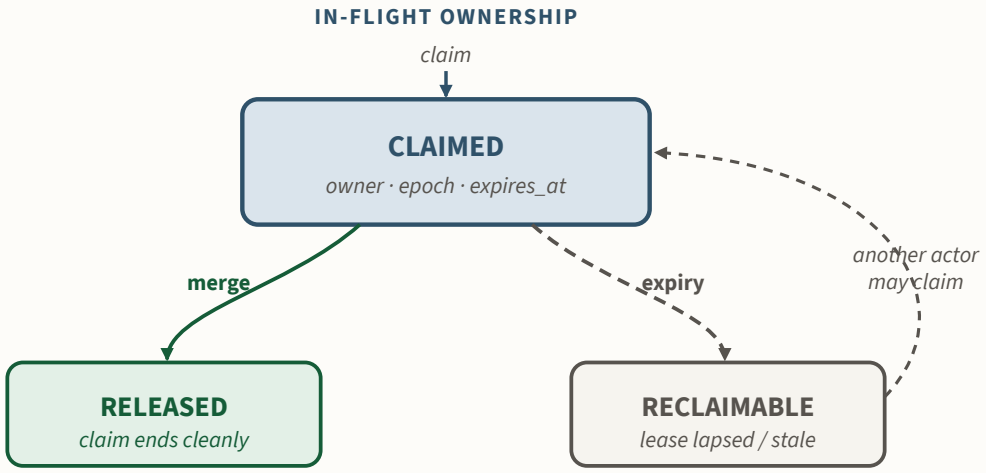


Figure 2.4-2: In-flight ownership model. *A current claim records owner, epoch, and lifetime. Merge releases the claim; expiry makes it reclaimable. The model distinguishes a live claim from a stale one without representing the storage machinery that implements the lease.*

As with the recovery ordering, the two layers stay separate. The generation *exposes* the question — is this release applicable to the current claim? — as one a checker or a release path can decide. Whether the running system *holds* the line — whether the release path actually compares generations and rejects the stale one — is a matter of authority for Part III. That guard is a **hard** one: the mechanism refuses the stale release rather than merely reporting it, and an exhaustive search over the interleavings serves as its verification. Here the model's job ends at making the invariant precise and checkable.

2.4.3 One Representation, Several Questions

The lease is the Part's clearest case of one representation answering several classes of question. Read it as an ownership model and it says who controls a job. Read the same claims as a behavioral model and they trace a lifecycle — claimed, held, expired, reclaimed. Count the active leases and the same representation becomes a measurement: how much work is genuinely in flight right now. Figure 2.4-3 sets the three readings side by side.

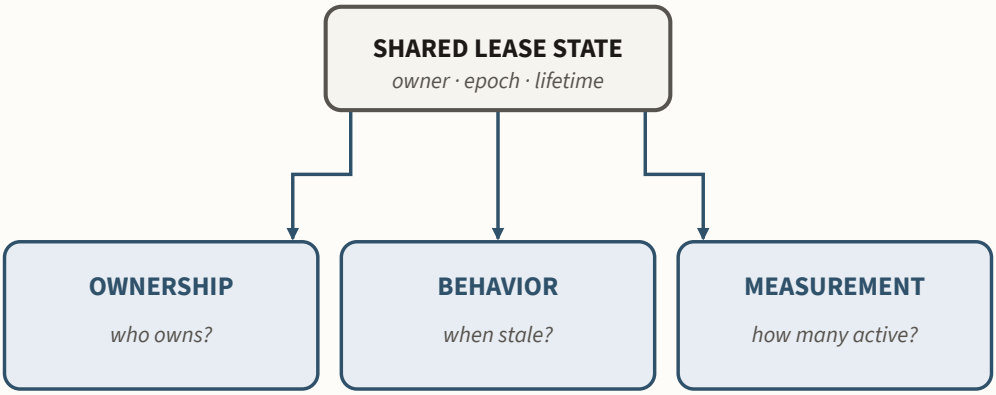


Figure 2.4-3: Model classes overlap. *The same lease state supports ownership, behavioral, and measurement questions. The classes distinguish purpose, not storage objects.*

That a lease supports ownership, behavioral, and measurement questions at once is not a sign the model is under-specified. It is the point the classes have been making since the opening chapter: they sort *questions*, not data structures, and one purposeful representation can answer several when they share the same underlying facts. This is the observation the closing chapter builds on when it connects the models through shared identity.

The lease model makes ownership under contention precise and its key invariant checkable. It does not establish that the implementation obeys the invariant, nor decide what should happen when a stale release is attempted — surface a conflict, drop the release, raise an alarm. Modeling names the claim; Alignment determines whether and how the obligation is enforced.

Ownership introduced permission implicitly: a stale claimant *may not* clear a current lease. The next family makes permission the whole question.

2.5 Decision Models: What Is Allowed?

The models so far have been descriptive: they say what the system *is* or *does*. A decision model says what the system *may* do. It represents alternatives and the rules that permit, require, or forbid them. That single change in mood — from *does* to *may* — is the whole subject of the chapter.

2.5.1 The Canonical Move: Permitted Alternatives

Engineering represents permission in several familiar forms: decision tables that map conditions to allowed actions, rule systems, access-control matrices that record which subject may act on which object[30], policy graphs, feature models that constrain which combinations of options are valid, and constraint systems that declare which assignments are admissible. What unifies them is not their shape but their mood.

This is the sharpest instance of the earlier lesson that topology does not fix meaning. A structural edge and a decision edge can be drawn identically and mean opposite kinds of thing: `A → B` as a structural edge says "A calls B," a fact about what happens; the same arrow as a decision edge says "A *may* call B," a rule about what is permitted. The consequence lives in the absent edge. A missing structural edge means "no such call happens"; a missing decision edge means "no such call is allowed." The gap between those two readings is the reason decision models exist (Figure 2.5-1).

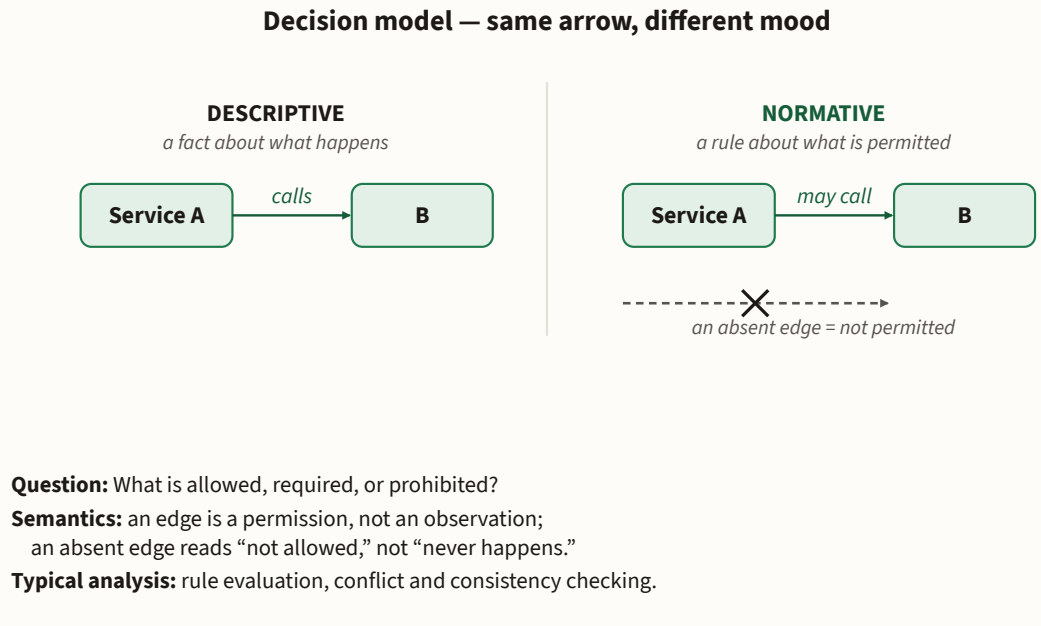


Figure 2.5-1: The decision move: the same arrow, a different mood. *A structural edge states that a call happens; a decision edge states that it is permitted. The absent edge carries the weight — "not allowed," not "does not happen" — and the analyses check a declared policy for consistency and least privilege.*

Once permission is declared, the claims a policy designer wants become expressible: which alternatives are permitted or forbidden, which are mutually exclusive, which imply prerequisites, whether the policy is internally consistent, whether it is complete over the cases that can arise, and whether it achieves least privilege. The analyses follow — evaluating a rule against a case, checking or solving constraints, detecting conflicts between rules, testing completeness and consistency. A declared policy turns "is this allowed?" into an evaluation rather than an argument.

2.5.2 DocAble: Service-Flow Policy

DocAble declares which of its services may communicate with which. The model is a graph whose edges carry the normative reading: an edge means the call is part of the sanctioned policy, and an absent edge means the policy does not permit it.

Service-flow model · Decision MODEL CARD

- **Engineering question** — Which service may call which, and what may be reached from outside?
- **Model** — a declared graph of services and the permitted edges between them; the deployment is wired from the graph rather than the graph inferred from the deployment.
- **Property** — internal communication is permitted only along a declared edge; one declared node is the public entry.
- **Quality attribute** — least privilege; architectural integrity; security.

Figure 2.5-2 draws the permitted calls. One node is declared publicly reachable; every other edge is an authenticated internal call. The model is the single source of truth for those edges, and the deployment is generated from it rather than inferred from it — so a completeness check can require every gated edge to be fully specified before the deployment is wired. That check is real and enforced, but it governs the *declaration's* internal consistency, not the running traffic. The model declares the topology; it is not a runtime firewall.

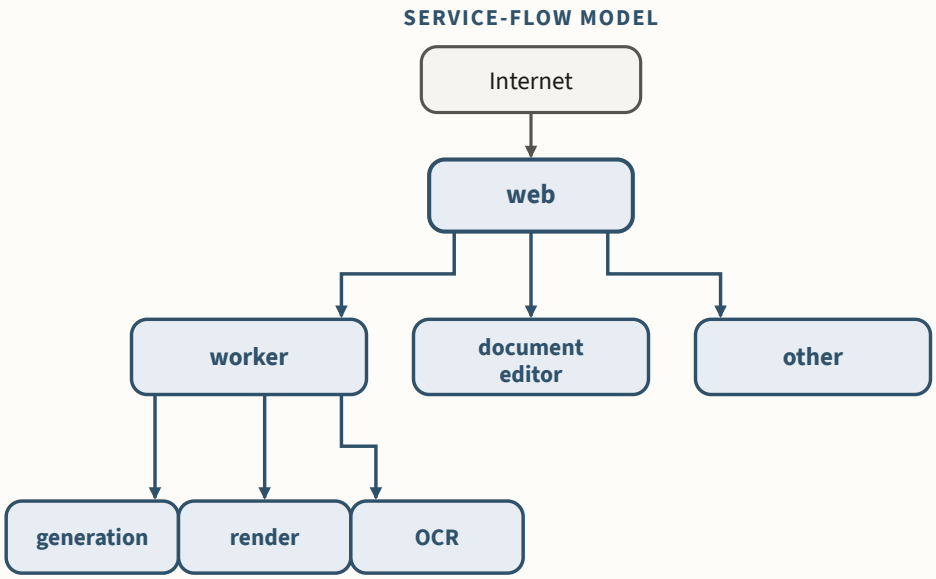
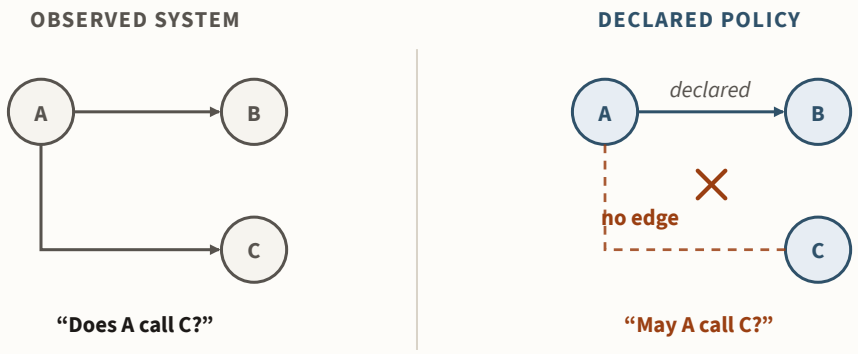


Figure 2.5-2: Service-flow model. *A declared graph of which service may reach which. The public web tier is the only node facing the outside; an absent edge means the relationship is not permitted by the model, and enforcement is separate.*

Declaring the policy exposes a question the running system cannot answer for itself: do the services call only along declared edges? That is a correspondence between an observed structural fact and a declared decision, and the two are different claims. The observed call graph reveals that A calls C; the decision model states whether A *may* call C. Figure 2.5-3 sets them side by side.



Observation reveals what is. The decision model declares what belongs.
Agreement between them establishes correspondence, not correctness.

Figure 2.5-3: Observation and intent answer different questions. *The observed system can reveal that A calls C; the decision model can state whether that relationship is permitted. Agreement establishes correspondence, not correctness.*

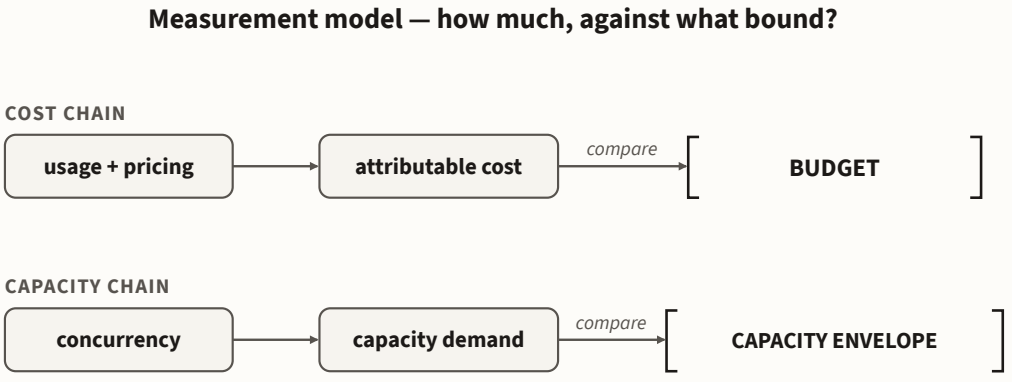
Agreement between observation and policy would establish *correspondence* — the implementation matching the declared intent — not *correctness*: a faithfully-obeyed rule can still be the wrong rule. DocAble generates its deployment from the declared graph, but nothing at runtime reconciles observed traffic back against the declaration. That open correspondence is not a gap to hide. It is the clearest illustration in this Part of a model exposing a checkable property without, by itself, establishing that the implementation satisfies it. Modeling declares what is permitted; whether the code stays on the declared edges, and what a departure should cost, is Alignment's question in Part III.

2.6 Measurement Models: How Much, and Against What Bound?

Every system emits numbers, and most of them are just telemetry: observations with no engineering meaning attached. A measurement model gives selected observations meaning by *relating* them: usage to cost, cost to a budget, concurrency to a capacity. It answers the quantitative question the earlier families set aside — how much, and against what bound?

2.6.1 The Canonical Move: Quantities, Relations, and Bounds

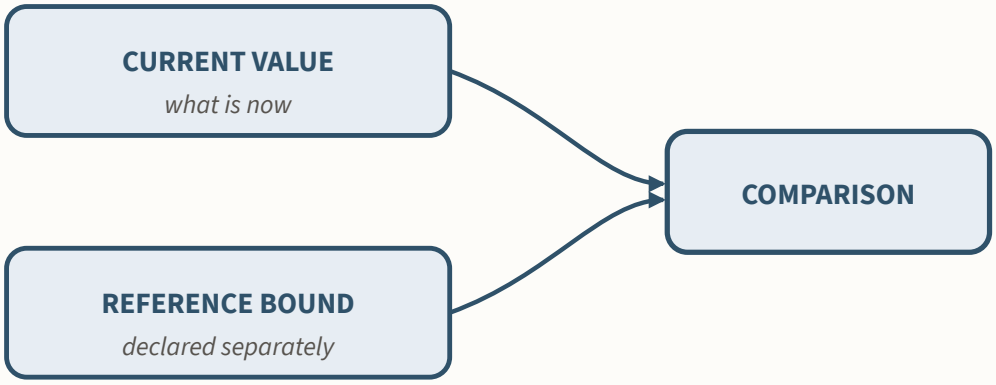
Engineering has many quantitative representations: performance and parametric models, capacity models, cost models, and the budgets and envelopes they are compared against; queueing and resource models appear where contention matters[31]. The move they share is small but decisive. Telemetry is observation; a measurement *model* relates quantities so that a bound becomes statable. A node is a quantity; a relation is an accounting or functional link between quantities — usage times price is cost, concurrent jobs are capacity demand; a bound is the envelope a quantity is compared against (Figure 2.6-1).



Question: How much, and against what bound?
Semantics: relate quantities (usage × price = cost; concurrency = demand), then compare each against a bracketed bound.
Typical analysis: aggregation, comparison against a bound, capacity planning.

Figure 2.6-1: The measurement move: quantities against a bound. *Telemetry becomes a model when quantities are related and a value is read against a declared envelope; the analyses aggregate and compare rather than watch a stream call by call.*

Once quantities are related to a bound, the claims a designer wants become expressible: upper and lower bounds, budgets, capacity constraints, rates, conservation relations, tolerances, and margins. The analyses follow — aggregation, comparison against a bound, sensitivity and capacity analysis, forecasting. Figure 2.6-2 draws the core relation: a current value read against a separately declared reference. The payoff is that "are we within budget?" becomes a comparison against a modeled bound rather than a guess over a dashboard.



The bound is declared apart from the quantity.
The model compares; it does not decide the consequence.

Figure 2.6-2: Core measurement relation. *A current value is interpreted against a separately declared reference bound. The model defines what is compared; it does not determine the consequence of the comparison.*

2.6.2 DocAble: GenAI Cost and Capacity

DocAble's consequential quantities are the cost of its GenAI usage and the capacity its concurrent work demands. The model keeps two chains deliberately separate, because they answer different questions and are compared against different bounds.

GenAI cost and capacity · Measurement MODEL CARD

- **Engineering question** — What does GenAI usage cost against budget, and what capacity does concurrent work demand?
- **Model** — two related chains: usage and pricing determine an attributable cost; concurrency determines a capacity demand.
- **Property** — attributable cost can be compared against a declared budget; capacity demand can be compared against a capacity envelope.
- **Quality attribute** — cost-awareness; capacity planning; operability.

Figure 2.6-3 draws the two chains. Note the deliberate wording of the property: the model says cost *can be compared* against a budget, not that cost *must not exceed* one. The cost chain is observational — a computed ratio, surfaced and reported, that never blocks work as it is spent. The capacity chain is a runtime-tunable cap, adjustable rather than proven. A cost figure is evidence; whether crossing a budget should block work, raise an alarm, or simply be recorded is a separate question of authority.

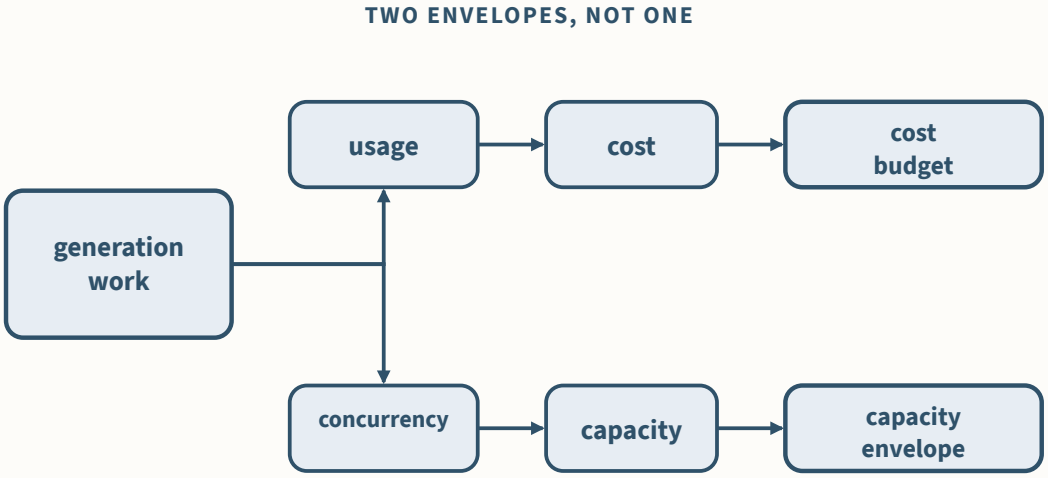


Figure 2.6-3: GenAI cost-and-capacity model. *Generation work fans into usage and concurrency; usage and pricing determine cost against a budget, and concurrency contributes to capacity demand against a capacity envelope. The model names the quantities worth relating — the interpretation placed over raw requests and charges, not the telemetry itself.*

DocAble does not attach a hard invariant to either quantity. Cost is routed to the administrative pane for observation and accounting, while capacity is controlled through a runtime-tunable limit. The model makes both quantities available for engineering use without requiring either to become an enforced obligation.

2.6.3 Tolerance and Margin

A bound is rarely a single line. A budget usually carries a margin; a capacity envelope leaves headroom. Those are the measurement family's version of the tolerance idea from the opening chapter: the model represents not just a target but the band of acceptable variation around it, so that "within tolerance" and "over the line" are both statable. The point is only that a measurement model can carry a margin — not to reopen the degrees-of-freedom discussion the opening chapter already settled.

The cost-and-capacity model makes spending attributable and demand comparable to an envelope. It does not decide the budget, guarantee the numbers are collected correctly, or enforce the bound. Figure 2.6-4 draws that separation: the model defines the quantity and the reference; whether a comparison is merely observed, used to adapt behavior, or enforced by a gate is Alignment's decision. This is the clearest case in the Part of a model that earns its place by exposing a property while leaving every question of authority open.

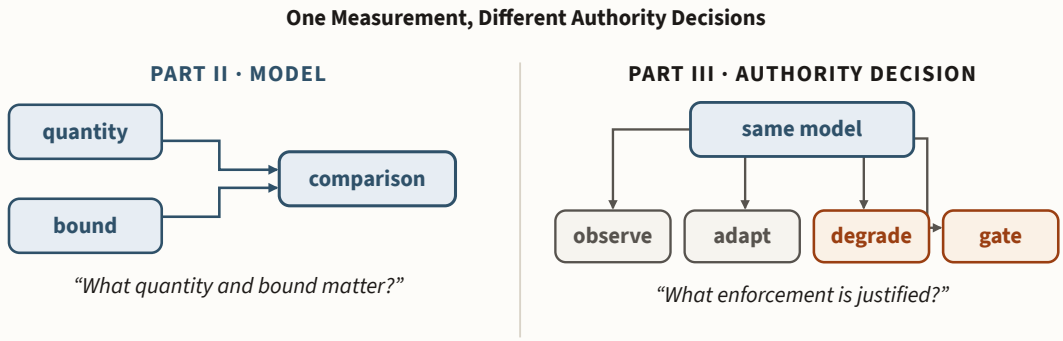


Figure 2.6-4: Modeling and authority are separate decisions. *The measurement model defines the quantity and reference bound; Alignment determines whether the result is observed, used to adapt behavior, used to trigger graceful degradation, or enforced by a gate.*

Measurement counts what is happening now. The last family asks what already happened, and what records it.

2.7 Provenance Models: What Did the System Do, and What Did We Observe?

The preceding model families describe what a system contains, what it may do, who may control something, what choices are permitted, or how much of some quantity is acceptable. Provenance looks backward at a realized execution: what did the system actually do, and what evidence did it retain about what happened?

Software engineers already make running systems observable. A log records selected events. A trace connects events across an execution. Metrics summarize quantities over many events. Audit records preserve actions that may later require explanation. Together, such mechanisms can expose an enormous amount of runtime history.

But observable history is not automatically useful engineering knowledge. A production system may emit millions of events while still making a consequential question difficult to answer: what changed this artifact, which operation changed it, and what evidence supports that account? A provenance model selects and structures the history needed to answer such questions.

2.7.1 The Canonical Move: From Observability to Provenance

The canonical move is to turn selected observations into structured history. An execution exposes far more than an engineering question usually needs: function calls, retries, intermediate values, network requests, model invocations, allocations, and other events. A sufficiently detailed trace may capture much of that execution. A provenance model deliberately keeps less.

The model identifies the consequential entities and operations, gives them stable identities, and records the relationships needed to reconstruct what happened. An artifact may identify the source from which it was derived, the operation that produced it, the component responsible for that operation, and the evidence retained with the change. The exact representation can vary. What makes it provenance is the meaning of the recorded history, not a particular graph notation or data format (Figure 2.7-1).

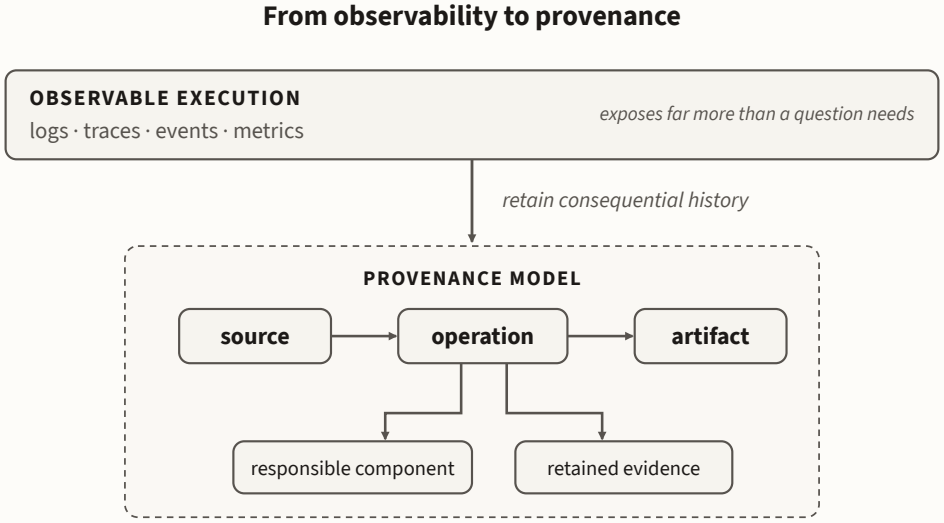


Figure 2.7-1: From observability to provenance. *Logs, traces, and other runtime observations can expose far more of an execution than an engineering question needs. A provenance model retains the consequential entities, operations, relationships, and evidence needed to reconstruct selected history.*

That reduction makes questions about realized history tractable. An engineer can trace an artifact backward to its source, follow an operation forward to the artifacts it affected, determine which operation produced a change, or reconstruct the evidence available during an audit. A question that would otherwise require searching and interpreting logs becomes a query over selected, structured history.

2.7.2 DocAble: Attribution

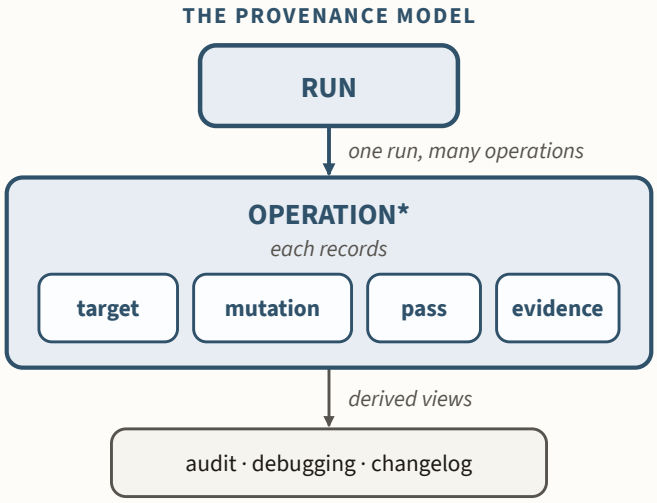
For DocAble, the consequential history is the history of changes to the user's document. A general execution log could record every function call, retry, model request, intermediate value, and service interaction involved in remediation. A distributed trace could connect much of that work across components. Those mechanisms make execution observable, but they do not by themselves answer the question DocAble needs to preserve: what changed in this document, what operation changed it, and what evidence supports that change?

DocAble therefore records consequential mutations as structured operations. Each operation identifies its target, the mutation made, the remediation pass responsible, and the evidence retained about the change. The resulting record is narrower than a general execution trace and more directly useful for explanation, debugging, and audit.

Attribution model · Provenance MODEL CARD

- **Engineering question** — What happened to this artifact, which operation produced the change, and what evidence records it?
- **Model** — a run made of operations, each naming its target, its mutation, the pass that made it, and the evidence it left.
- **Property** — consequential operations have structured attribution linking the action, target, mutation, responsible pass, and retained evidence.
- **Quality attribute** — auditability, debuggability, and consistency between what changed and what is reported.

Figure 2.7-2 draws a reduced provenance model: a run contains operations, and each operation names a target, a mutation, the pass responsible, and the evidence it left. This is not a trace of every function call. That would reproduce execution rather than model it. The model keeps only the events needed for explanation and audit.



The structured record carries the provenance model; audit, debugging, and changelog are derived from it, not maintained apart.

Figure 2.7-2: Attribution provenance model. *A run contains consequential operations; each operation records its target, mutation, responsible pass, and retained evidence. Audit and explanation can be derived from this shared history.*

Human-facing explanations can be derived from the same structured history rather than maintained separately. Record the consequential action once, then derive the audit view,

debugging view, or changelog from it. That reduces the chance that what the system did and what its documentation says will drift apart.

Recording execution is not the same as modeling its history. An execution contains far more activity than an engineer ordinarily wants to preserve as provenance. Too coarse a record collapses distinct consequential changes into an opaque “document modified” event. Too fine a record becomes another execution trace that later reasoning must reconstruct into meaning. Choosing the grain is therefore part of the modeling decision.

For DocAble, the useful grain is the consequential mutation. The provenance record answers the questions for which it exists: what changed, what operation changed it, what evidence was retained, and what later explanation can be derived from that history.

2.7.3 When Provenance Becomes Executable

DocAble’s edit record supports one additional capability: the recorded history can be replayed. During a PDF remediation session, document mutations are recorded as typed edit values in a per-session log — roughly thirty variants covering the changes the editor can make. A replay engine can apply the same sequence deterministically. The same representation can therefore explain what happened and reproduce the resulting mutations.

Replay is not what makes the record a provenance model. It is an additional capability enabled by this particular representation. The record remains a purposeful reduction of execution: it omits the reasoning, analyses, control decisions, retries, and other activity that produced the edits. It records the consequential mutations that resulted.

That separates three engineering questions that are easy to conflate. The computation graph asks what computations may compose. The provenance record says which consequential mutations did occur in this run. Replay asks whether that realized history can be reproduced. No one representation subsumes the others. Figure 2.7-3 sets the static structure and the realized history side by side — the reader's first sight of two heterogeneous models describing the same operation from different sides, which is the work the next chapter takes up.

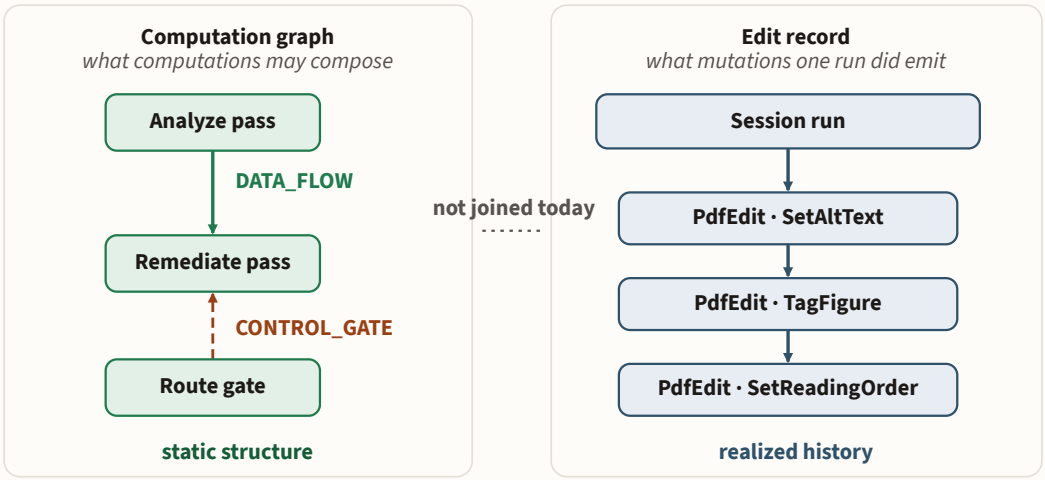


Figure 2.7-3: Computation structure versus realized history. *The computation graph models declared computations and their typed composition; the per-session edit record logs the consequential mutations one run actually produced. They remain separate reductions today. A shared computation identity could join runtime edits to the static model when an engineering question requires that relation, without collapsing the two representations into one.*

One limit closes the chapter. A provenance model does not guarantee that every consequential operation actually leaves the required record. DocAble addresses that separately, by requiring typed mutation verbs to emit attribution and checking that obligation structurally. That check is a **hard** presence guarantee over a **soft** content claim: it can establish that every mutator leaves an attribution record, but not that the semantic account inside that record is correct — that remains the author's. Part II models the evidence an operation should leave; Part III asks what mechanism gives that obligation authority.

The next and final chapter of this Part steps back from any single model to ask how the six fit together as a system.

2.8 System Knowledge: Connecting the Models

The six preceding chapters each drew a different model of the same product. Together, those models expose different structures through which engineers can reason about the product's architecture: decomposition, behavior, ownership, policy, quantitative bounds, and provenance. It would be natural to collapse those views into one large model, but DocAble has no such model, and MAGE requires none. Instead, the system uses a **family of structured models**, each built for a different question. Shared identity and traceability connect them where engineers repeatedly need to relate one view to another. System knowledge is not a seventh class of model. It is the **substrate** that lets those views work as a system.

2.8.1 A Heterogeneous Modeling Substrate

DocAble's models differ in shape on purpose. Some are registries of components and their relations. Some represent state transitions. Some encode contracts or decisions. Some capture quantitative relationships. They share identities, derivations, query surfaces, and traceability where those help, but nothing forces them into a single representation. Read the collection as a **modeling substrate**, not as one system-knowledge graph.

The substrate is concrete, and it is larger than any one chapter. In DocAble it holds on the order of ninety structured model modules plus several data dialects, organized in layers: a component-identity layer names what exists; a per-component metadata layer annotates each one; orthogonal meta-files sit across the top for deployment topology, inter-service edges, data flow, required environment, state machines, and the governance graph itself. In DocAble, these models are represented as data rather than importing the implementation they describe. This keeps model and territory separable enough for correspondence to be checked. When a model is meant to provide independent evidence of correspondence, keep it distinguishable from the artifact it checks. Figure 2.8-1 abstracts this substrate to the six model classes and the shared identities that join them.

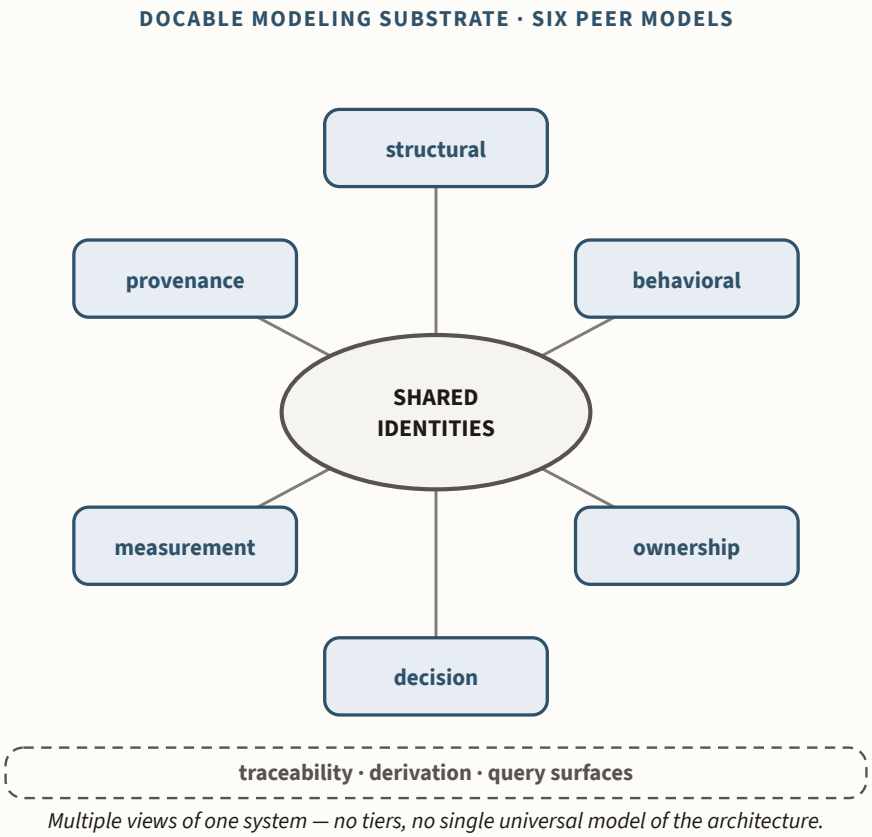


Figure 2.8-1: Heterogeneous models over shared identities. *Structural, behavioral, ownership, decision, measurement, and provenance models remain separate representations while joining through stable identities and explicit correspondence where needed. Their combination supports architectural reasoning without requiring the architecture to be encoded in one representation.*

The unity that matters is semantic, not syntactic. A service named in a structural registry is the same service a flow model references. A work item in an ownership view resolves to the lifecycle and the measurements for that work. A provenance event identifies the operation whose effect it records. Nothing demands that the models share a file format or metamodel; they need only agree on **entity identity**.

That agreement follows one discipline: prefer shared identity and derivation over independently maintained copies of the same fact. Where one model needs a fact owned elsewhere, join through a stable identity or derive a projection. For example, the access policy is generated from the service-flow model rather than maintaining a second endpoint list; a journey model joins to those endpoints by call site rather than copying them. Where duplication is unavoidable, make the correspondence explicit and checkable. This reduces the number of independent truths that must be reconciled by hand.

Static and runtime representations need not collapse into one model either. A computation model can identify remediation work and its declared relations while a provenance record identifies the mutations realized during one run. Shared identity allows observations to join the structural model when a question requires it: DocAble now associates staged-run latency and cost with computation identities. The per-session edit record remains separate because it answers a different question — what happened to this artifact — and is not yet a per-run execution graph over computation nodes.

Shared identity also provides a query entry point. An agent — or an engineer — enters the substrate through the object its task names and traverses outward, following identities across models rather than loading everything. A canonical query surface lets the agent traverse represented relationships rather than rediscover them through repository search: name a component and ask what it owns, what it depends on, which flow reaches it, what its lifecycle looks like. The agent does not need *the whole model.* It needs the connected slice that answers the engineering question in front of it. Figure 2.8-2 draws that entry and outward walk.

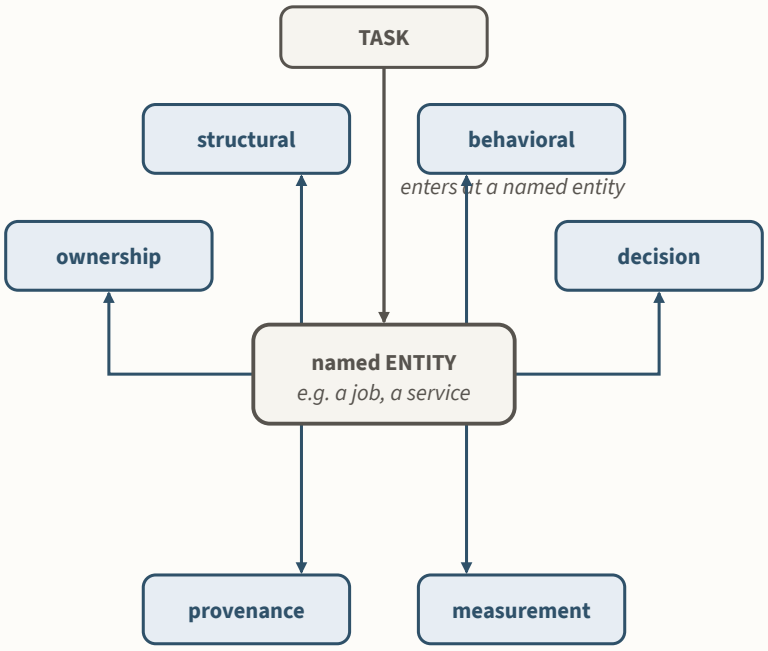


Figure 2.8-2: Task-driven traversal. *A task enters the substrate through a named system entity and follows only the model relations required by its engineering question.*

2.8.2 Maintain Explicit Correspondence

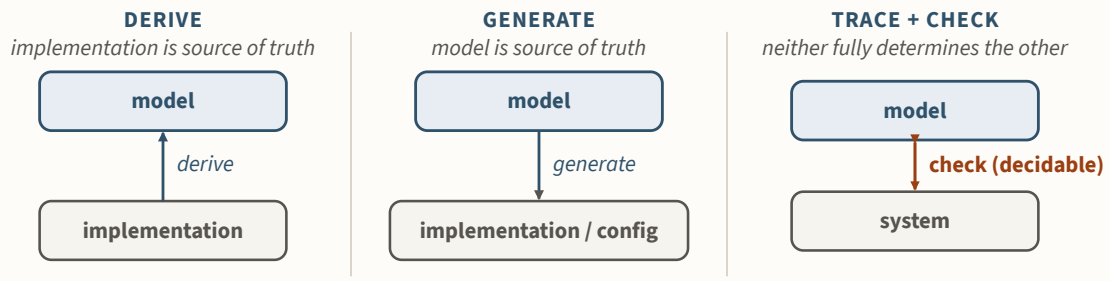
Multiple models create a correspondence problem: represented facts can diverge from the system or from one another. Engineer the relationship between map and territory so divergence is caught rather than assumed away. Which machinery you reach for depends on which side is the source of truth.

Make the direction of correspondence explicit. Where the implementation is the source of truth, derive the model. Where the model is the source of truth, generate the downstream artifact. Where neither fully determines the other, maintain traceability and check the mechanically decidable correspondences. Figure 2.8-3 lays the three cases side by side.

Derive when implementation owns the truth

The model is a projection of the code, reconciled at build time. A component-zone registry reads the real directory tree; a journey's dependency list is induced from its real call sites. The drift check is a **reconciler** — it re-derives the model from the code and fails on divergence. This pattern is common when models are introduced into an existing codebase. ### Generate when the model owns the truth

Code, configuration, or documentation are emitted from the model. The access policy, the service catalog, and the wire-contract types are generated from the service-flow model. The drift check is a **freshness and provenance check** — the generated artifact carries a header, and a hand-edit or a stale regeneration is a finding.



Three correspondence patterns — derive where implementation leads, generate where the model leads, trace and check where neither does.

Figure 2.8-3: Three correspondence patterns. *Derive where implementation is the source of truth; generate where the model is the source of truth; otherwise maintain traceability and check the correspondences that are decidable.*

DocAble's remediation subsystem now illustrates both directions around one source of truth. Its explicit computation model names remediation work and its composition, and the execution machinery consumes that model to determine dependency order and readiness. The model therefore has operational authority over composition: this is the generate case in the consequential sense that realization follows the model rather than reconstructing composition from node implementations.

A separate Python remediation-graph view projects that structure for analysis and governance. For Office formats, its edges derive from the same registry read/write metadata used by scheduling; for PDF, parallel typed declarations are held in correspondence by blocking checks. The view can therefore be queried, joined to other models, and checked for drift without itself becoming the runtime scheduler.

One source of truth can thus support both directions at once: execution consumes the authoritative model, while analytical representations derive from it or from parity-bound declarations around it. The important question is not whether an artifact is called a model, graph, registry, or view. It is which representation owns the fact and which consumers are required to follow it.

The graph is derived, but it remains a view: it suppresses method bodies, internal algorithms, runtime history, timing, and most document state, keeping only the entities and relations its structural questions need. Derivation prevents one kind of drift by construction, since an engineer never authors the edges, but it does not establish that the underlying architecture is good, nor that this view captures every relationship or obligation that matters. Those are separate claims, and Part III separates them.

Trace and check where neither owns the truth

Where neither side can be fully derived from the other, the move is **traceability plus drift checking**. Relate each model element to the implementation that realizes it, then check the correspondences a machine can decide. Keep that relation *live*: resolve the target against current code when the check runs, rather than storing a line number or a symbol name that is itself another snapshot able to drift. A deleted target then surfaces as a finding instead of a stale green edge. In DocAble a traceability gate resolves each model-to-code anchor through a language-aware resolver — one for each source language — and reports every anchor that no longer resolves. Before the gate existed, such renames could silently strand model references.

Some correspondence remains semantic

Mechanical correspondence has a declared surface. Some claims are decidable: an anchor resolves, an observed dependency belongs to the permitted set, or a generated artifact is current. Other claims remain semantic and require judgment. Still others are not modeled at all.

These are different gaps: a semantic miss occurs when correspondence still holds syntactically but meaning has changed; an unmodeled region has no model-based coverage. A recurring miss is a candidate for refining the representation or adding a control. Figure 2.8-4 distinguishes mechanically decidable claims, semantic claims requiring judgment, and properties outside the modeled surface.

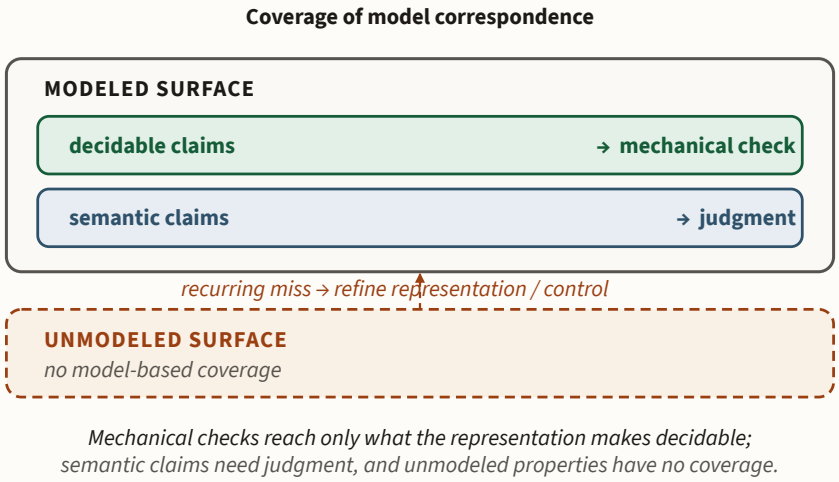


Figure 2.8-4: Coverage of model correspondence. *Mechanical checks cover only claims the representation makes decidable; semantic claims still require judgment, and unmodeled properties have no model-based coverage.*

For invariant-bearing models, one useful executable pattern is a structured schema plus checked predicates over that schema. The schema defines entities, relations, fields, or states; the predicates state properties that a checker evaluates over the declared domain. Other executable models may instead drive generation, simulation, transformation, or analysis. A schema alone does not establish correspondence to a changing system; checks or derivations are needed where that correspondence matters. Figure 2.8-5 draws that relationship.

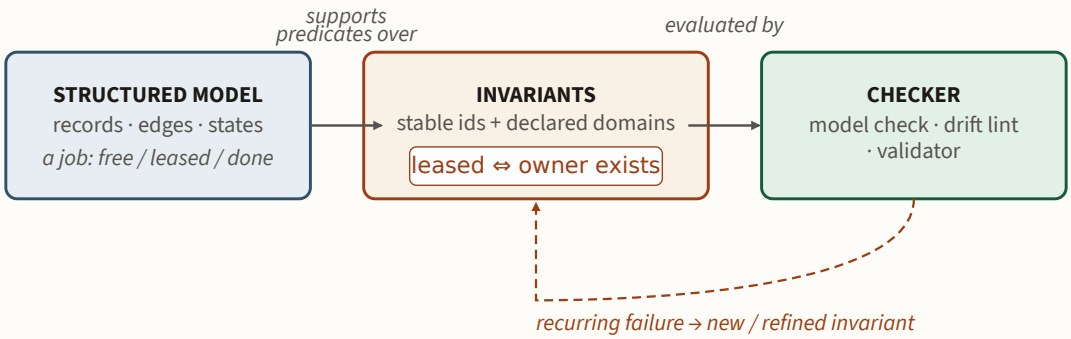


Figure 2.8-5: Checked predicates over a structured model. *A structured model supplies the domain; invariants state properties over that domain; a checker evaluates them at a declared cadence. This is one executable-model pattern, not the definition of executability.*

Inset — What is an invariant? FORMAL METHODS

An **invariant** is a property required to hold over a declared domain of states, structures, or observations. In a behavioral state machine, the invariant may be a predicate that must hold in every reachable state; in a structural model, it may require every observed dependency to belong to a declared set; in a measurement model, it may require a quantity to remain within a hard capacity bound. State both the property *and* the domain the checker claims to establish it over.

Take a behavioral illustration. A job carries two fields at once — a `status` of `free`, `leased`, or `done`, and a `lease_owner` that names the holding worker or is empty. Different steps write the two fields, so they *can* fall out of agreement, which is exactly why you write the invariant down. "A job is leased if and only if it has an owner" is a predicate over the pair: `(status == "leased") == (lease_owner is not None)`. A `free` status with a lingering `lease_owner` is a stale lease; a `leased` status with no owner is a lost lease. Separating the two writes can make either inconsistent state reachable, and both break the predicate. The predicate becomes mechanically meaningful when a checker evaluates it over the declared reachable state space:

```
STATES = product({"free", "leased", "done"}, {None, "worker-1", "worker-2"})
  def invariant(s):                                    # leased iff owned
      return (s.status == "leased") == (s.lease_owner is not None)
  assert all(invariant(s) for s in reachable_states())  # check every reachable state
```

None of this proves the model *right*. Synchronization does not establish that the model is correct. A model can be structurally in sync and still express the wrong architecture. A pointer can resolve while its meaning has changed. A schema and its generated client can agree while a live producer emits something else. The guarantee the machinery offers is narrower, and worth stating plainly:

> **For the surface the model claims, check the correspondence that can actually be decided.**

That boundary returns in Part III, where correspondence and correctness part company. Derive where implementation is the source of truth; generate where the model is the source of truth; trace and check where neither fully determines the other.

2.8.3 Modeling Makes Properties Available to Engineering

A model does not govern the system merely by existing. A structural model can name a forbidden dependency without preventing it; a behavioral model can name an illegal transition without rejecting it; a measurement model can represent a budget without stopping work that exceeds it. Modeling makes these properties explicit. It does not enforce them.

The important change is semantic reach. Local syntactic obligations can often be checked with little explicit modeling. Architectural boundaries, temporal obligations, distributed ownership, provenance requirements, and end-to-end policies require enough representation to expose the relevant system relationships. Modeling makes those relationships available to analysis and, where appropriate, to governance.

They are also available to design. A dependency graph may reveal that an execution boundary is artificial; a behavioral model may expose a recovery protocol that should change; a quantitative model may show that an otherwise plausible decomposition cannot meet its resource envelope. In each case, the model does not merely document the architecture. Analysis of the view can change the architecture before implementation commits to the choice.

Alignment does not therefore require a separately represented model. Some obligations are already observable at an action or artifact boundary and can be constrained directly. Models extend the range of properties the environment can enforce.

The result is not architecture replaced by models, but architecture made available through them. Different questions call for different views; shared identity lets engineers cross those views when a decision spans them; analysis can change the design; and executable repre-

sentations can sometimes realize or constrain that design directly. None of this requires one complete model of the system.

Part III asks which obligations the engineered environment should enforce, where they become decidable, and what evidence and mechanisms are sufficient to enforce them.

Deep dive — Beneath commodity intelligence: why structure helps

AI

MAGE deliberately treats intelligence as a commodity. The method should not depend on a particular neural architecture, training recipe, or context-window size. Still, the language-model systems available in 2026 offer a useful view one level below that abstraction. They suggest why engineering the structure around an intelligent agent can matter as much as improving the intelligence itself.

Access to state is not the same as reasoning over state. Modern language models can accept large contexts, but their ability to use those contexts depends on the reasoning the task requires. OOLONG, for example, was designed to require semantic processing and aggregation across much of a long input rather than retrieval of a few relevant passages. Frontier models degrade substantially on these tasks even when the input fits within their nominal context windows[32]. Recursive Language Models (RLMs) make the distinction especially clear. Instead of placing a long input directly into the model's context, an RLM keeps it in an external environment that the model can inspect programmatically, decompose, and pass in pieces to recursive model calls. On several long-context tasks, this restructuring substantially improves performance, including on inputs far beyond the underlying model's context window[33].

This belongs to a broader shift from treating a language model as a function that produces an answer toward treating it as a reasoner operating within an environment. ReAct interleaves reasoning with actions that obtain new observations from an external environment[34]; CodeAct extends this idea by allowing executable code to serve as an expressive action language[35]. RLMs go further by making context management and decomposition themselves part of the interaction with the environment. The common lesson is not that any particular harness is universally correct. It is that the structure through which intelligence encounters a problem changes what that intelligence can effectively do.

Recent work offers a possible explanation in terms of compositional generalization. Zhang, Li, and Khattab hypothesize that frontier models may already be highly capable on many bounded problems while remaining poorly "managed" on long-horizon ones: a larger task succeeds when it can be decomposed into smaller tasks that individual model calls can reliably solve[36]. Zhang and Khattab subsequently argue that a harness can supply some of this structure by transforming complicated external state into observations that remain locally familiar to the underlying model; their RLM experiments suggest that such structure can improve generalization across both task length and domains[37]. These are developing results rather than a settled theory of language models, but they offer a useful substrate-level interpretation of an engineering observation: more intelligence and more context do not remove the need for structure.

MAGE operates one level above this work. A generic agent harness can manage context, call tools, delegate subtasks, and decide how to decompose computation. Software engineering also provides semantic decompositions of the system itself. An architecture model exposes components and boundaries. A data-flow model exposes consequential paths. An automaton exposes legal state transitions. Ownership and obligation models expose authority and constraints. Bidirectional links among models and implementation let an agent move from code to the relevant model, across related models, and back to the affected code rather than reconstructing all of that structure from a flat collection of source and prose.

This distinction also marks the boundary between Modeling and Alignment. Modeling gives commodity intelligence representations in which consequential engineering relationships are explicit and traversable. Alignment gives selected obligations authority by connecting them to mechanisms in the engineered environment—hooks, validators, tests, gates, runtime checks, and others—that can enforce them. Better learned decomposition may improve how agents perform work inside that environment. It does not decide which customer, organizational, safety, or societal obligations the environment should preserve.

The connection is therefore explanatory, not foundational. Future models may reason over much longer contexts, learn better decompositions, and require different harnesses. MAGE should survive those changes. Its engineering claim is more durable: capable reasoning becomes more useful when consequential system structure is made explicit, and consequential obligations become more reliable when the environment gives them authority. Most importantly, those structures keep engineers in control of what the system is for, what it must preserve, and which tradeoffs are acceptable, even as increasingly capable machines take on more of the work required to realize those decisions.

Models can provide value before enforcement: they support reasoning, analysis, composition, and traceability. Part III asks the separate question of authority—which obligations the environment should enforce, where they become decidable, and what mechanisms should carry that enforcement.

Summary

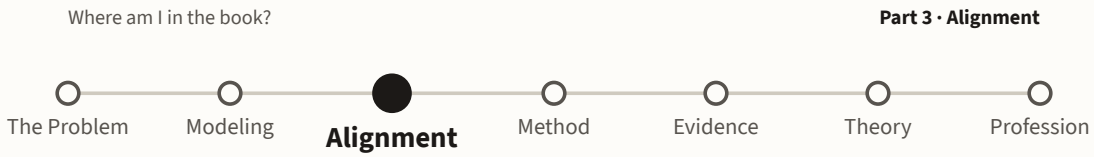
1. **Model for an engineering question.** A model is a purposeful reduction of a system. Preserve the distinctions needed to answer the question; omit detail that does not contribute to it. Build and maintain the model only while the engineering leverage it provides justifies its carrying cost.
2. **Externalize engineering knowledge and intent.** Models can expose architecture, obligations, dependencies, invariants, provenance, costs, and other properties that would otherwise have to be reconstructed from implementation and context. Different questions may require different models of the same system.
3. **Choose degrees of freedom deliberately.** Every representation leaves some choices open. Reduce consequential freedom where doing so improves reasoning or control; preserve freedom where realization can safely remain open.
4. **Maintain the correspondences you depend on.** Multiple representations create correspondence obligations. Derivation, generation, validation, shared identity, and other mechanisms can keep related representations sufficiently consistent for their intended uses.
5. **Distinguish correspondence from authority.** A model may describe another artifact, or realization may be required to follow the model. Decide which representation owns each consequential fact and which consumers must follow it.

MODELING PRINCIPLE

Externalize engineering knowledge and intent into explicit, structured models that both engineers and agents can reason through.

Appropriate representations make broader engineering questions tractable and make additional properties available for Alignment.

Part 3: Alignment



QUESTION THIS PART ANSWERS

How do I give engineering obligations authority in my environment?

ALIGNMENT PRINCIPLE

Give engineering obligations authority by encoding them into mechanisms that constrain actions, produce evidence, evaluate that evidence, and control admission.

An obligation is authoritative when the engineered environment can enforce it.

CARRYING FORWARD

Commodity intelligence · The Printer · Model · Modeling Principle · Invariant · Traceability

NEW HERE

Alignment Principle · Governed Engineering Environment · Governance Mechanism · Constraint · Sensor · Validator · Gate · Governance Conversion · Engineering capital

Engineering systems differ in how they carry their obligations. Some remain in human expertise, documentation, or practice and must be interpreted when the relevant decision arises. Others are encoded into the engineered environment so that compliance can be constrained or evaluated mechanically. Software architecture provides one example: an architecture can state permitted dependencies and interactions, while conformance checks determine whether an implementation respects them.[38–40] System-safety engineering provides a broader precedent, identifying hazards and constraints over the larger system and designing controls, feedback, and operating processes to keep those constraints satisfied.[41] Software engineering uses mechanisms ranging from types and access controls to validators and deployment gates for the same purpose.

MAGE calls this move Alignment and treats it as the companion to Modeling. Modeling makes engineering knowledge and intent explicit. Alignment gives selected obligations authority by connecting them to mechanisms in the engineered environment that can constrain work, evaluate its results, or determine whether those results are accepted. An obligation with no such mechanism may still matter greatly, but satisfying it continues to depend on human or agent judgment at the point of use.

Commodity intelligence makes this longstanding problem newly important. A generative implementer can produce plausible realizations rapidly and explore degrees of freedom that human implementers might never have considered. Instructions alone do not determine which realization will appear. If an architectural relation, behavioral invariant, ownership rule, measurement requirement, or policy matters to the system, relying on every future realization to reconstruct and voluntarily preserve it leaves that obligation on the probabilistic surface.

Selected obligations can instead be encoded into the engineered environment. Some can be carried directly by representations, interfaces, or permissions; others require evidence produced as work proceeds and evaluated before the result is accepted. The appropriate mechanism depends on the obligation and on where the evidence needed to evaluate it becomes available.

Not every preference warrants mechanical enforcement, and not every important property can support it. Some obligations remain dependent on expert judgment; others would cost more to mechanize than doing so is worth. Part II's tolerances and degrees of freedom also distinguish obligations from variation that engineering deliberately leaves open. The chapters ahead develop how to choose among these possibilities.

Figure 3.0-1 traces the progression from engineering intent to a governed environment.

Figure 3.0-1: How intent becomes authoritative. *Enforcement begins with an obligation at a boundary where the environment can act on it. Mechanisms enforce selected obligations; experience reveals additional opportunities for governance; accumulated controls eventually require governance of their own.*

Some mechanisms can be designed before work begins. Others emerge when a failure exposes something the environment did not represent, observe, evaluate, or control. When those lessons become durable engineering structure, later work can inherit them as engineering capital. As the mechanisms accumulate, the control machinery itself becomes an engineering object.

3.1 Where Obligations Can Be Enforced

A foundation model produces probabilistic outputs; a harness turns selected outputs into actions. Once those actions can alter a repository, invoke an API, change infrastructure, or ship an artifact, the engineering question is no longer only how to steer the reasoner. It is where the surrounding environment can intervene between a proposed action and its consequence.

Start with the distinction between guidance and authority. **Guidance influences behavior; authority determines what the environment will permit or accept.** A prompt, brief, example, or playbook can make one action more likely than another, and that influence can be strong. A permission boundary can forbid an action; a validator and gate can refuse a result. Feedback can then return the evidence or explanation to the reasoner—a failing test, an observed call, a measured latency, or a validator's explanation—so that subsequent work can respond. MAGE uses all of these mechanisms, but reserves authority for cases in which the engineered environment can enforce the obligation rather than merely advise the reasoner.

Recent agentic-security guidance draws a similar boundary. Once agents act before commit time, the repository is too late to be the security boundary: the tools, permissions, and execution environment that produce a change join the trusted base, so agentic security must secure the process that creates the artifact, not only the artifact[42].

An obligation can be enforced at several boundaries in the work. The useful question is not which one is "hardest," but **where the relevant obligation is visible and where a mechanism can act on it.** Four representative boundaries recur throughout the book — reasoning input, agent action, the work unit, and the accepted artifact or runtime state. Figure 3.1-1 puts them on one line.

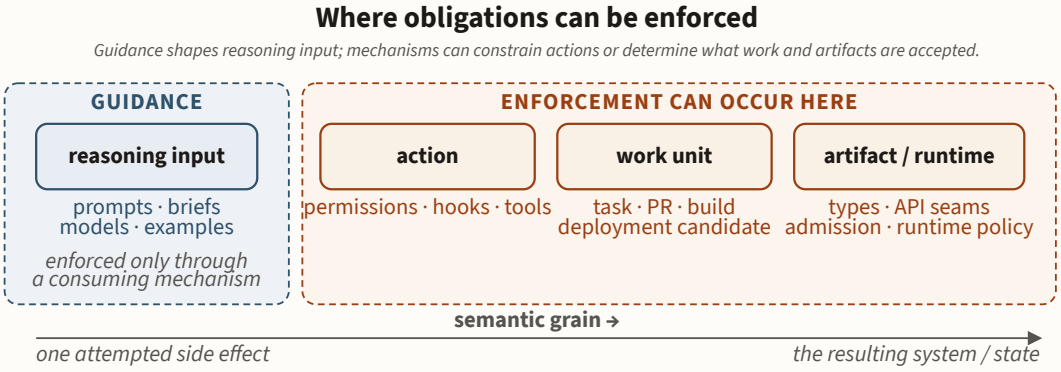


Figure 3.1-1: Intervention boundaries. *Guidance shapes reasoning input; mechanisms can constrain actions or determine whether completed work, artifacts, or runtime states are accepted. Place the mechanism at the earliest boundary where the obligation is legible and enforceable.*

3.1.1 Reasoning Input, Action, Work Unit, and Artifact/Runtime

The first location is **reasoning input**: the context, representations, examples, and instructions put in front of the reasoner. This is where Modeling and context engineering do most of their work: a finite reasoner does better when the relevant system knowledge is already externalized and selected for the task. But placement in context is ordinarily guidance, not authority. The model can misread, forget, or disregard what it was shown.

> **Inset — Context Engineering: Making the Right Knowledge Available** AI
>
> Context engineering concerns what information is made available to a reasoner for a particular episode of work. Instructions, examples, retrieved material, prior decisions, and engineering models can all contribute to that working state. The problem is not fitting more information into a context window. It is selecting the right information, at the right abstraction, when the decision is made.
>
> MAGE gives context engineering durable engineering knowledge to draw from. Modeling externalizes engineering knowledge so that each agent does not have to reconstruct it from source code, documentation, and prior conversations. Context engineering can then select the relevant portions of that knowledge for the work at hand. The distinction is useful: Modeling engineers what is known; context engineering engineers what is presently available to reasoning. Retrieval, memory, compaction, skills, and other mechanisms may change how that selection is implemented, but the underlying engineering problem remains.
>
> One realization is dynamic context injection (DCI): the environment observes properties of the work, selects the engineering knowledge relevant to those properties, and places that knowledge into the agent's reasoning context. The mechanism improves what the agent can see when making a decision; it does not itself make the resulting guidance authoritative. Section 4.5 develops this pattern operationally.

Guidance can make an acceptable realization more likely. A sufficiently sound constraint or admission mechanism can instead exclude a represented class of unacceptable realizations from the surface it governs. The first improves the producer's chances of satisfying the obligation; the second changes what admission must trust.

$$\text{guidance: } p_i \uparrow \qquad \text{authority: selected } J_i \text{ need not be trusted for admission}$$

The first enforcement boundary is the **action boundary** exposed by the harness and its tools. Here, one attempted side effect is in view: a tool call, an API request, a shell command, or a file mutation. Permissions can remove an action from the agent's repertoire; a hook can inspect or reject a proposed action; a tool can expose a deterministic operation instead of asking the model to improvise it. An alert-triage agent permitted to shift a maintenance window but forbidden to create infrastructure has a bounded blast radius because the API denies the forbidden action, not because the prompt discourages it[43].

The second is the **work-unit boundary**, where many actions compose into a meaningful unit: a completed task, pull request, build, migration step, or deployment candidate. Obligations that no single action can express become legible here: that the model and the implementation agree at task completion, that a change carries all its required evidence, that "done" actually holds. A pre-commit check can reject a local defect; a task-completion audit can weigh the whole body of work; a deployment gate can refuse a candidate whose accumulated evidence is inadequate. These boundaries matter because different obligations become legible at different grains of work.

The third enforcement boundary is the **artifact or runtime boundary**: the resulting system or running state. Types, narrow APIs, architectural seams, schemas, runtime permissions, and admission policies can make a forbidden state impossible to express or impossible to accept. DocAble's format seam provides a concrete example: ordinary remediation code reaches the raw format library only through one structured interface, while a ban-lint detects any forbidden direct access. The model states the architecture; the interface and the check give the boundary its authority.

3.1.2 Placement and Strength Are Different Decisions

The four boundaries do not form a monotonic scale from weak to strong. A narrowly scoped API permission may bind more absolutely than a late CI check; a type may rule out one state completely while saying nothing about a system-level obligation a task-completion gate can weigh. **Where a mechanism acts and how strongly it enforces an obligation are different design decisions.**

Richer models can expand the properties the environment is able to state and govern, but substantial enforcement can exist without them. A bounded execution sandbox, for example, can constrain actions without requiring a rich model of the system it contains.

3.1.3 Return Evidence at the Earliest Useful Boundary

The boundaries also decide where feedback can return. A harness can preserve durable state before a lossy compaction; an action hook can explain why a call was refused; CI can hand back a failing invariant while the change is still fresh in the window; a deployment validator can report which evidence failed admission. The earlier useful evidence returns, the less work an agent must reconstruct before repairing the miss.

Controls within the GEE can also interact or conflict; Section 3.5 treats their growing portfolio as a system in its own right.

The Alignment Principle: give engineering obligations authority by encoding them into mechanisms that constrain actions, produce evidence, evaluate that evidence, and control admission. *In other words:* once the environment should enforce an obligation, satisfying it should not depend solely on each future agent remembering it.

Part IV turns these mechanisms into method. For now, their placement is easiest to see in systems already using them.

WORKED EXAMPLES

Docker. Docker's public account places enforcement in the environment, not its instructions: the agent acts only through a constrained tool seam, and the platform — not the prompt — decides what that seam permits. A move outside the boundary is refused, not merely discouraged. The permission is enforced at the action boundary: the agent cannot override it through its instructions. Two industrial fleets place enforcement the same way at coarser grain. Zenseact's public account describes each domain team's agent as little more than a config file and a skill, while a central team owns the runtime, permissions, and context routing; an agent's reach is a property of where it sits in the platform, not of how it was asked. Spotify's public account describes migration agents operating through a restricted API and a narrow Git interface, with every change clearing a merge gate before it lands. The gate, not the agent's confidence, admits the work.

DocAble. DocAble distributes its controls across the boundaries: a skill in the harness, an action hook, a pre-commit check, and — its sharpest — an application-boundary seam for document mutation. At the application boundary, DocAble represents document mutation through a sanctioned structured seam. Static controls reject ordinary code that bypasses that seam for the raw format library. The model states the architecture; the interface and lint give that relation authority.

Takeaway. Place the enforcing mechanism at a boundary where the obligation is legible and the environment can evaluate or constrain the relevant work. The appropriate boundary depends on the obligation.

The next chapter asks what must be true before an obligation deserves authority.

3.2 From Engineering Intent to Enforceable Obligation

Authoritative evaluation requires two things: **an obligation independent of the work being judged and a boundary where the evidence needed to evaluate that obligation is available.** Without the first, a checker may only confirm that the system agrees with itself. Without the second, the mechanism tries to enforce a good obligation before the necessary evidence exists.

Part 2 supplied many ways to make obligations explicit. Some live in models: permitted service edges, legal state transitions, ownership relations, operational bounds. Others live directly in types, permissions, schemas, tests, or policy. Alignment does not require every obligation to originate in the same kind of representation. It requires that the obligation be distinguishable from the thing being judged and that an authoritative mechanism can act on the result.

Software engineering already has a long tradition of connecting engineering intent to realization through requirements, traceability, verification, and acceptance. A requirement may be traced to design elements, implementation, tests, and other evidence so that engineers can ask whether the intended obligation survived realization. Alignment builds on that tradition but asks an additional question: what consequence should follow from the evidence?

Traceability can connect an obligation, its realization, and its evidence without making the obligation authoritative. A requirement may trace cleanly to a test whose failure is merely reported. Conversely, a permission boundary may carry substantial authority without a rich traceability model. Alignment therefore separates the relation among intent, realization, and evidence from the mechanism that acts on the resulting judgment.

Where an obligation permits variation, the acceptable region must likewise come from the engineering intent that governs the work. A measured implementation can establish what latency it achieved or what dependencies it contains; whether those values are acceptable requires an independent engineering obligation.

A description and the implementation it describes can therefore agree while both violate the intended property. Placement matters as well: if an obligation is enforced at a boundary too small to make it legible, the checker may inspect the wrong thing.

3.2.1 Agreement Is Not Correctness

A drift check can establish a declared correspondence between two artifacts. It cannot establish that the correspondence expresses the right intent. Suppose an endpoint is required to be authenticated. If an independently authored policy says so, a check can compare the implementation against that obligation and flag an open endpoint. If instead the representation is derived only from the implementation, it may faithfully record the endpoint as open. Model and code agree; the security obligation is still violated.

The distinction is not **whether the model was derived or authored.** Derived models are useful throughout MAGE. The distinction is **description versus independent obligation.** A model extracted from code can supply the current *is*; an independently authored invariant can supply the governing *ought*. A hand-authored model can be just as wrong when it merely repeats the implementation's assumptions. Authority requires something against which the work can actually be wrong.

Drift checking therefore has a precise job: preserve a declared correspondence between model and implementation. If code and model diverge, surface the disagreement. Correctness requires another claim—the policy, invariant, requirement, or property that must hold. Once that obligation exists, a validator can compare evidence against it and a gate can decide what follows.

Figure 3.2-1 distinguishes the three relations.

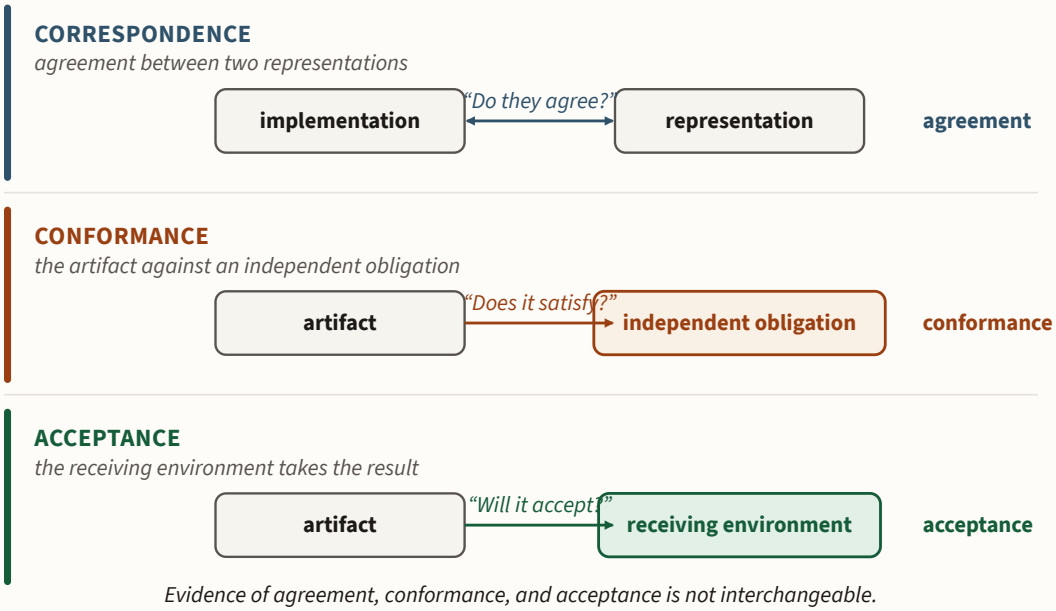


Figure 3.2-1: Three distinct relations. *Correspondence tests agreement between representations; conformance tests an artifact against an independent obligation; acceptance asks whether the receiving environment will take the result. Evidence for one relation does not establish the others.*

WORKED EXAMPLES

Siemens. The Siemens reconstruction describes models, implementation, and engineering analyses kept in correspondence across a digital thread without making every disagreement a build-blocking event. In that account, a human confirms the bill of materials while physics solvers carry parts of the evaluation. The example is useful precisely because correspondence and gating are separate design choices: the coupling can hold without a hard gate deciding the instant the two diverge.

DocAble. DocAble uses all three relations. Wire-contract tests ask whether independently represented sides of an interface correspond; they establish agreement, not external compliance: a synthetic payload can satisfy both sides while the real producer emits something else. PDF output can instead be checked against veraPDF, an independently implemented conformance oracle for the ISO standard. In Office workflows, the acceptance question may be whether Microsoft's own checker accepts the result. Agreement, conformance, and acceptance are useful evidence, but they are not interchangeable.

DocAble's remediation pipeline gives a sharper example of correspondence and authority as separate properties. Its computations and their composition are represented explicitly rather than recovered from the bodies of the remediation passes. The execution machinery consumes that computation model to determine dependency order and readiness. The model therefore has authority over composition: node implementations determine what individual computations do, but they do not independently decide how the pipeline is assembled.[9]

A separate Python remediation-graph view exposes that structure for analysis and governance. The current view contains 125 computations and 57 typed relations: 32 data-flow edges, 24 control gates, and one cross-service relation. Fifty-five relations are projected and two remain authored seeds. For Office formats, the projection derives edges from registry read/write metadata also used by scheduling. For PDF, the analytical relation is projected from typed Produces, Consumes, and ConsumesForControl declarations and held to the executable territory by blocking parity checks.

The two representations therefore carry different authority even where their correspondence is strong. The computation model is consumed by execution. The analytical view is consumed by analyses and checks. Making the view more faithful does not by itself give it authority over execution; authority follows from what the environment requires a representation to control.

The Office checks expose a second limit. Nodes and edges are now projected and their parity checks run green, but those checks remain audit-only while one live pass still runs outside the registration surface the checks can enumerate. Promoting them to blocking authority before that escape is drained would certify a green result over an incomplete observation surface. A validator should not receive more authority than its evidence surface warrants.

[9] The computation model describes the pipeline rather than the history of any one remediation. Runtime telemetry can now be joined to computation identities for aggregate measurements such as latency and cost, but the per-session edit record still names realized document mutations rather than computation nodes. DocAble has not yet built a per-run execution graph joining that edit history to the computation model.

Figure 3.2-2 draws authority and correspondence apart.

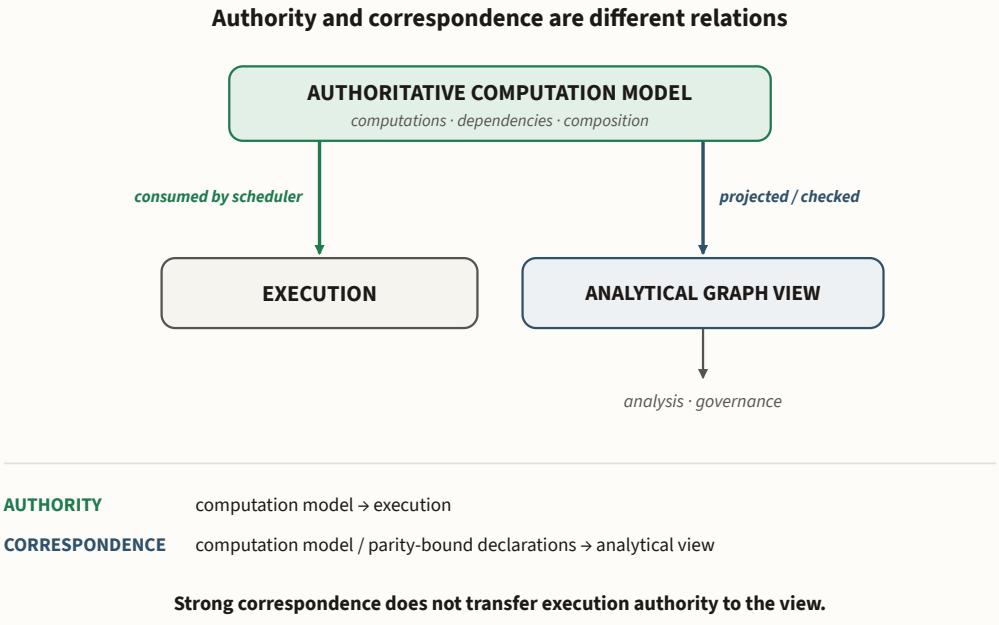


Figure 3.2-2: Correspondence strength and authority direction are independent. *DocAble's execution machinery consumes an explicit computation model to determine remediation composition. A separate analytical graph view projects that structure and is held in correspondence with its upstream declarations. Stronger correspondence makes the view more trustworthy as a representation; it does not make the view authoritative for execution.*

3.2.2 The Semantic Gap

A second problem is placement. **An obligation can be stated clearly and still be unenforceable at the boundary where you try to check it.** If the property exists only over a whole task, no single commit holds enough meaning to decide it. If data exfiltration appears only as a pattern across calls, no individual system call carries the whole property. A **semantic gap** occurs when an obligation requires meaning that the chosen boundary does not expose. Figure 3.2-3 shows the fix: lift the check to where the meaning is legible.
[10]

[10] *A footnote on where the placement rule comes from.* It echoes the end-to-end argument in system design: a function that needs end-to-end knowledge cannot in general be implemented completely at a lower layer that lacks that knowledge[44]. Their setting was where to place *functionality* in a distributed system; the setting here is where to place *authority* over engineering work. The design instinct is the same — put the deciding mechanism where the semantics needed to decide the property are available.

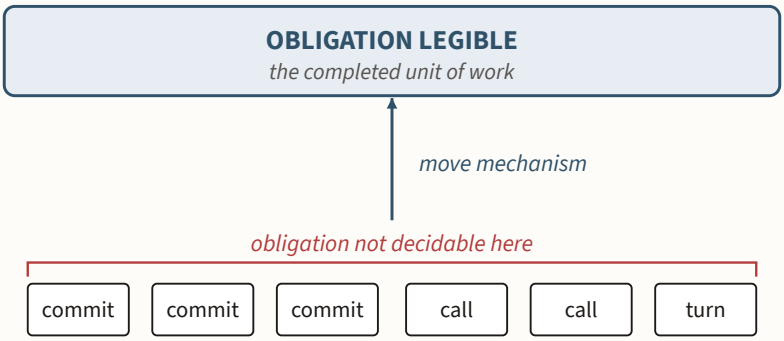


Figure 3.2-3: The semantic gap. *A property that spans several events cannot be decided from any event alone. Move the mechanism to the earliest boundary where the evidence required by the obligation is assembled.*

A deployment can pass twenty checks and still serve a blank page. No earlier check necessarily failed: the relevant property — does the deployed page correctly render the emitted data? — does not exist at the source-file, build, or emission boundary. It becomes legible only after those pieces compose in the served system. A smoke check that fetches the served page is the earliest boundary that can decide it.

The same gap appears inside a single task. Take a concrete obligation: *if this task substantially changes the call graph, the corresponding architectural model must still agree when the task closes.* A real task runs through several intermediate commits, and temporary disagreement during those commits can be legitimate. A pre-commit check therefore acts too early — it sees a fragment while the property belongs to the completed unit of work. At task return, the accumulated changes, model updates, tests, and other completion evidence can be weighed together — which is exactly where the companion repository places its definition-of-done audit, over the whole unit of work rather than any single commit inside it.

3.2.3 The Earliest Decidable Boundary

The agent-return check is one instance of a rule that runs under the whole of this Part:

> **Place authority at the earliest boundary where the obligation is both legible and enforceable.**

Earliest matters because delay makes a failure more expensive to repair. *Legible* matters because moving the check earlier than the property permits produces false refusals or meaningless checks. The rule explains the whole mechanism menu at a glance:

- **Types** act as soon as an expression is formed — a shape is legible the instant it is written.
- **Action hooks** act at one tool call, one command — where a single move is legible.
- **Task-completion checks** act at the work-unit boundary — where “done” first becomes readable across a body of commits.

- **Deployment gates** act after broader evidence has accumulated — the last boundary before the artifact is accepted.

And some properties stay illegible until later still: a leak visible only across ten calls, a cost overrun visible only across a season of jobs. No boundary makes them readable sooner. The correct placement is not the earliest available boundary; it is the earliest boundary that can actually decide the obligation.

Part 2 and Part 3 therefore use companion selection rules:

> **Modeling** — *choose the reduction that makes the engineering **question** answerable without carrying unnecessary detail.*
>
> **Alignment** — *choose the earliest boundary at which the **obligation** becomes legible and enforceable.*

Both rules keep irrelevant detail from driving the engineering decision.

- Modeling avoids carrying detail the question does not need.
- Alignment avoids binding a decision before the necessary evidence exists.

Choose the wrong abstraction in Part II and the required property is not represented. Choose the wrong boundary here and the required evidence is not available.

Real boundaries are sometimes approximate. In DocAble's own orchestration loop, the ideal moment to preserve durable session state would be the instant before a lossy compaction, but waiting for that exact hook leaves too little budget to write the hand-off comfortably. So the implemented mechanism fires earlier, when context is becoming full *and* the durable record has gone stale. The example is a useful warning: semantic placement is an engineering trade-off, not a demand for a theoretically perfect boundary.

With an independent obligation and a boundary where it becomes legible, the next question is what the mechanism at that boundary actually does.

3.3 Constraints, Sensors, Validators, and Gates

Environmental authority follows a familiar engineering pattern: restrict admissible behavior where possible, observe what occurs, evaluate the resulting evidence against a desired condition, and act on the verdict. Part II made properties explicit and showed how models can be analyzed for the questions they represent. Alignment must additionally connect those properties to the realized system. A mechanism may observe or derive evidence from the implementation, evaluate that evidence against an obligation, and give the resulting verdict consequence. MAGE separates these functions into four roles: **constraint**, **sensor**, **validator**, and **gate**. A mechanism may perform several roles at once; the distinction separates prevention, observation, judgment, and admission.

> **Governance mechanism.** *A repeatable structure in the engineered environment through which Alignment constrains work, produces evidence, evaluates evidence, or controls admission.*

The four roles are:

- **Constraint** — removes or narrows an action or state the system is allowed to express.
- **Sensor** — observes the work or system and produces evidence.
- **Validator** — evaluates evidence against an obligation.
- **Gate** — controls whether work may cross a boundary.

Implementations often bundle several roles.

The four split into two pairs. Constraints and sensors concern the work or system state: one restricts what may occur, the other observes what did occur. Validators and gates concern decisions over evidence: one evaluates the evidence, the other enforces the verdict.

3.3.1 Prevention and Observation

On the work-facing side of the loop, the two complementary moves are prevention and observation; Figure 3.3-1 draws the split. A constraint restricts what may occur; a sensor observes what did occur. Prefer a constraint where the unwanted action or state can be excluded cheaply and reliably. Otherwise, expose enough evidence for the obligation to be evaluated at an appropriate boundary.

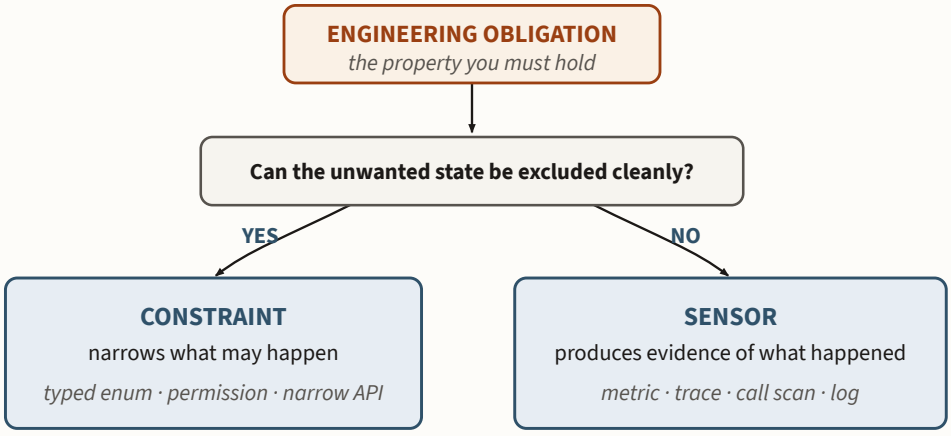


Figure 3.3-1: Constraint or sensor. *Prefer a constraint when the unwanted state can be excluded cheaply and reliably. Otherwise instrument the relevant state and produce evidence for later evaluation.*

An instruction carries authority only when a mechanism in the environment acts on it. If a brief must contain a required section, check for that section before launch and refuse the launch when it is absent.

A sensor observes what happened and produces evidence: `actual = observe(system)`. It does not judge; a validator decides whether that evidence satisfies the obligation. When observation exposes a correctable violation after the fact, repair costs an additional iteration. That makes prevention attractive where the unwanted state can be excluded cleanly; many important properties, however, can only be observed.

A CI test suite often bundles three roles: executing the code produces evidence, checking the result validates it, and failing the build gates progress. One artifact can perform several roles without collapsing the distinction among them.

The use of explicit evidence to justify consequential engineering claims also has a substantial history in safety and assurance engineering. Safety cases and the broader assurance-case tradition organize claims, supporting evidence, assumptions, and argument so that acceptance does not rest on an unsupported assertion that the system is safe.[45,46] MAGE inherits that concern but separates several roles that an assurance argument may connect: what evidence the system produces, what evaluates that evidence, and what consequence follows from the evaluation. An argument can justify a claim without itself controlling the

system; Alignment asks when the surrounding engineering environment can act on the claim.

A sensor is only as good as the signal it can read. **Observability** means that a running system exposes enough signal to explain what it did and what went wrong. Observability supplies the loop's evidence path: the relevant state or event must be exposed in a form a sensor can read. A gate can act only on evidence available at its boundary, whether produced earlier or derived there. That may be the event a gate fires on, the log a checker reads, or the trace that turns "the build failed" into "the build failed *here, for this reason.*" If the evidence an obligation requires does not exist, downstream authority cannot recover it; the environment must first instrument the relevant state or event.

A constraint removes an unwanted action or state from the admissible space. Unlike a sensor, which observes the resulting state after the fact, a constraint makes the prohibited move unavailable through that mechanism. Closed representations are the cleanest example. A lifecycle encoded as arbitrary strings admits values the design never intended, and a mismatch surfaces only an iteration later, when the test suite crashes. Encode it as a closed enumeration instead and the compiler rejects any value outside the declared set on the spot. The enum does not watch for the illegal state later — it removes that representation from well-typed code, and no iteration is spent catching it. Where the valid action or representation space is naturally closed, MAGE prefers representations that exclude invalid members structurally.

Programming-language theory made this move systematic: choose a representation or type discipline in which some invalid programs cannot be written as well-typed terms.[11] MAGE generalizes the engineering preference rather than the formalism: where practical, restrict the action or representation space so the bad state cannot be produced, instead of detecting it again on every later iteration.

[12]

Human processes can play the same roles: logging senses, review validates, and approval gates. Human judgment remains necessary. Repeated, mechanically decidable judgments are candidates for durable structure.

The sharpest instance of the pair is ownership of a unit of work. DocAble's in-flight lease makes active ownership visible and lets apparently abandoned work become recoverable. The lease does not itself grant final authority to perform the consequential side effect. A durable compare-and-set supplies that constraint: competing claimants cannot both acquire it, so at most one performs the effect. The lease may expire because a worker is merely slow, so redelivery cannot imply permission to repeat the effect. **Recovery is not permission.**

[11] Benjamin C. Pierce characterizes type systems as syntactic methods for classifying program phrases so that classes of erroneous behaviors can be ruled out automatically[47].

[12] *A footnote on control engineering.* The decomposition echoes the closed-loop posture of control and systems-safety engineering without treating an agent fleet as a control system. See Leveson 2011[41].

3.3.2 One Obligation, Several Ways to Hold It

A modeled obligation does not imply one particular enforcement technology. Ask first what kind of property it is and where the environment can act on it.

If an invalid state can be removed from the action space, prefer the constraint. DocAble's work-item claim is the simple case: a database compare-and-set updates the row only while it remains in the expected prior state. Two competing claimants cannot both win the same transition. The ownership obligation is therefore carried by the runtime primitive itself rather than by a later checker.

Other properties live across sequences of otherwise legal actions. The in-flight lease is one example. Reclaiming a stale lease, issuing a new epoch, and receiving a late release from the previous holder are each individually plausible events; their interleaving can still violate the ownership invariant. Here the environment needs a validator that can reason over the composed state space. DocAble uses a small, purpose-built model checker: an executable transition model enumerates the reachable interleavings within explicit bounds and searches them for violations of the ownership invariants.[13] The checker is deliberately specialized to the protocol. For this small protocol, a purpose-built checker keeps the method close to the model and avoids unnecessary tooling.

Temporal obligations require another shape of evidence. "No two workers own the same claim" is violated by one reachable bad state. "Every submitted job eventually terminates" is not. A finite trace can show progress, but it cannot establish that every fair execution eventually reaches a terminal state. A temporal model checker can reason about that stronger claim.

[13] More precisely, this is bounded explicit-state model checking. "Bounded model checking" conventionally refers to bounded executions encoded for SAT/SMT solving; DocAble's checker exhaustively explores an explicitly bounded state space instead.

These are not levels of sophistication. They are different matches between property and mechanism. The engineering preference remains the same throughout MAGE: make the bad state impossible when you can; otherwise produce proportionate evidence that an appropriate evaluator can use to decide the obligation. Figure 3.3-2 sets the three matches out.

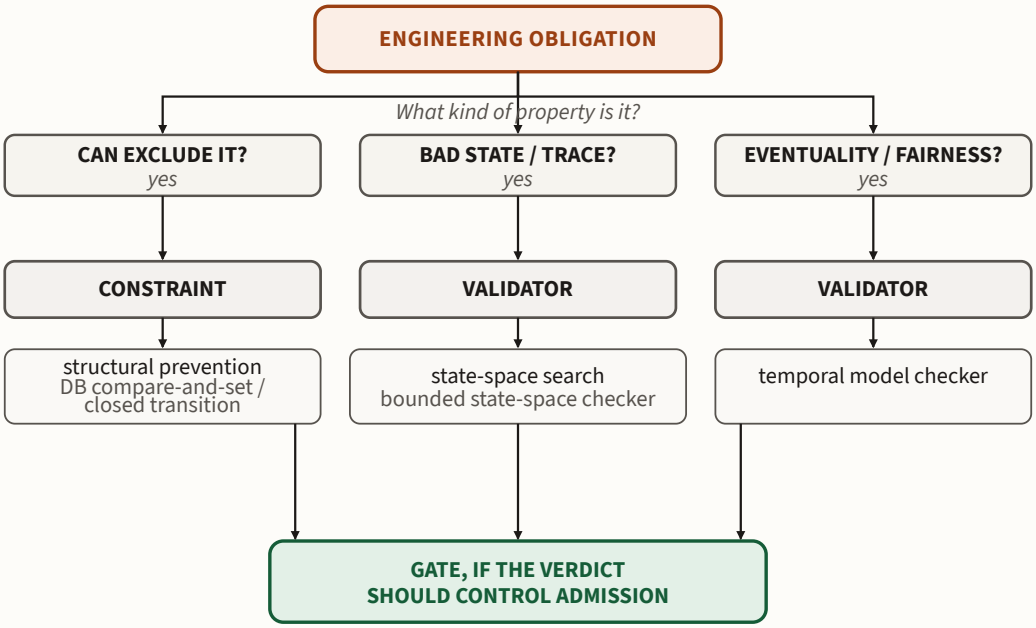


Figure 3.3-2: Match the mechanism to the property. *An obligation may be held structurally by excluding the invalid state, evaluated by exploring reachable states, or checked as a temporal property over executions (for example with TLA+/TLC). These are alternatives selected by the engineering claim, not stages on a maturity ladder. A gate is a separate decision that enforces the resulting verdict.*

Inset — Synthesizing the Guardrail PROGRAMMING LANGUAGES

Hard controls raise an obvious scalability question: who builds all of the checkers? The simplest answer is the same one we have used throughout this book: work with the agent. Once an invariant is explicit, an agent can help translate it into executable checks —linters, schema constraints, architectural tests, static-analysis rules, or mappings from model concepts to implementation artifacts. The engineer remains responsible for the invariant and for validating that the resulting mechanism actually enforces it, but need not hand-code every guardrail.

There is a stronger version of this idea. A static analysis can itself be represented with degrees of freedom: specify what property the analysis should establish while leaving choices about how to perform the analysis open. Recent work has proposed self-adaptive static analysis, in which the analysis representation admits automatic optimization for the particular program and analysis. The aim of such methods is to generate precise

and efficient analyses through self-aware, self-optimizing, sound analysis machinery —automating the realization of the control without making its enforcement probabilistic.[48,49]

This is the same modeling move applied recursively. The obligation enforced by the guardrail is fixed; the implementation of the guardrail need not be. Rather than asking an agent to probabilistically judge whether generated code satisfies an obligation, automation can help construct and optimize the mechanism that checks it. As implementation becomes cheaper, verification need not remain artisanal. The factory can help build its own guardrails.

Whichever validator supplies the judgment, authority still depends on what the environment does with its verdict.

3.3.3 Validators and Gates

The other two roles act on evidence and admission. A **validator** turns evidence into a judgment: does the observed call graph contain only permitted edges, did this journey establish its post-condition, does the provenance record account for every mutation? It reads the evidence a sensor produced — or derives it straight from the artifact — and asks whether it satisfies the obligation. A validator may be deterministic, probabilistic, or human; its authority comes not from itself but from what the environment does with its verdict. A correspondence validator can enforce agreement without making either representation authoritative for execution. DocAble's remediation-graph parity check blocks drift between the projected graph and typed pass declarations even though the executable declarations remain upstream.

A **gate** supplies that consequence. It decides whether the work may cross a boundary: commit, merge, deploy, execute, close, or draw on a scarce resource. Validation and gating are separate design decisions. A cost ceiling can be measured and evaluated without blocking production; a security invariant may deserve immediate refusal. **Giving a validator's verdict gating authority is a commitment.** Giving an uncertain validator blocking authority converts its false positives into rejected work or outages.

Authority is conditional CAVEAT

A mechanism can guarantee only the property it actually governs, under the assumptions on which its evidence and implementation depend. A validator can faithfully enforce the wrong obligation; a model can omit a consequential distinction; a sensor can fail to observe the relevant state. Hard control therefore does not mean "the system is correct." It means that, within a stated scope, acceptance no longer depends on the agent voluntarily satisfying that obligation.

A loop may re-derive evidence at a later admission boundary rather than rely on the producing agent's self-report or on an earlier marker.[14] Re-derivation can strengthen independence and freshness: the later mechanism evaluates the artifact or state that is actually being admitted. DocAble's definition-of-done audit takes this approach, recomputing its evidence at close rather than relying on evidence produced earlier in the task. The principle is simpler than the implementation choice: the thing asking to pass does not get to define what counts as having passed.

3.3.4 One Measurement, Different Authority Decisions

The four roles are not stages of maturity. DocAble's GenAI cost-and-usage model makes the point. Related measurements from the same model can support different authority decisions, and Figure 3.3-3 fans them across the roles. Usage is continuously sensed. Capacity signals can adapt request behavior. Exhausting a per-job budget can produce soft completion rather than outright rejection. A hard daily-budget gate exists but is dormant unless configured. One quantitative model can support observation, adaptation, graceful degradation, or hard admission control. The appropriate consequence depends on the obligation attached to the quantity, not on the supposed maturity of the environment.

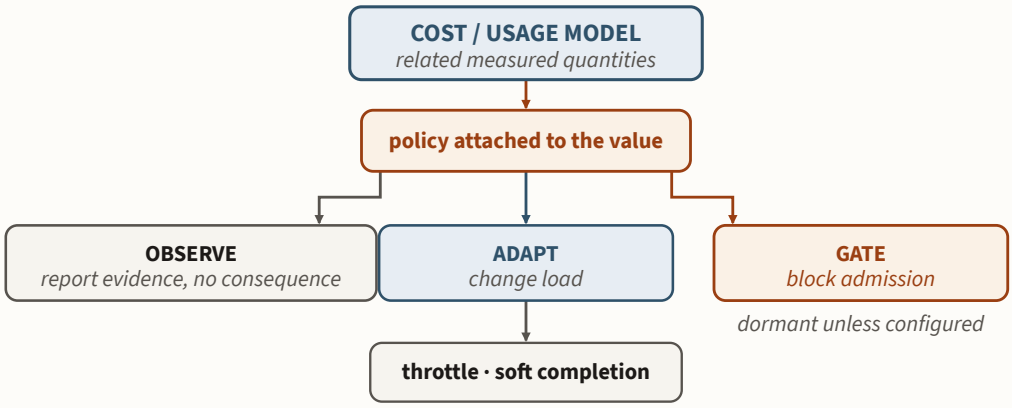


Figure 3.3-3: One Measurement, Different Authority Decisions. *The same measured quantity may be observed without consequence, used to adapt behavior, trigger graceful degradation, or control admission. The appropriate response follows from the obligation attached to the measurement, not from a maturity sequence.*

A quantitative tolerance can also expose margin. If a value remains inside its acceptable bound, margin describes the room between the realized value and that boundary. A request

[14] Re-derivation trades additional computation and latency for stronger assurance that the evidence corresponds to the artifact or state being admitted. Reusing or caching an earlier result can be a sound optimization when the system can establish that the relevant inputs have not changed; the optimization should preserve that correspondence rather than merely assume it.

completing in 120 ms against a 200 ms ceiling has 80 ms of margin. That information can matter before conformance fails: two realizations may both satisfy the same obligation while one operates much closer to its boundary and therefore has less room to absorb variation or future change.

That calculation is easy only when the model supplies a meaningful distance to the boundary. Quantitative obligations often do: latency, memory use, throughput, or cost can be compared directly with a declared bound. Many software obligations are qualitative instead. A dependency may cross a forbidden architectural boundary or remain within the permitted graph; a transition may be legal or illegal. Some such obligations admit useful notions of distance from failure, while others do not. MAGE therefore uses margin where the model provides a meaningful measure; it does not assume that every tolerance does.

3.3.5 Positive and Negative Constraints

Constraints are often introduced as prohibitions: this code must not call that library. DocAble's provenance machinery demonstrates the complementary form. Every typed mutation verb is required to emit attribution. A source-time rule checks that the required call is wired into the mutation path. Figure 3.3-4 sets the two forms side by side: one mechanism removes a forbidden action; the other requires evidence-producing behavior.

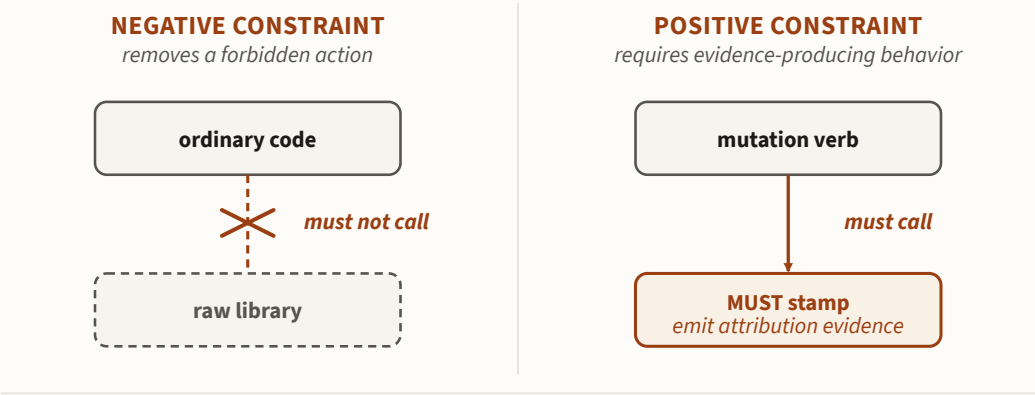


Figure 3.3-4: Negative and positive constraints. *A negative constraint removes a forbidden action; a positive constraint requires a specified action or evidence-producing behavior. Both restrict the admissible implementation space.*

3.3.6 Provenance-Carried Authority

Earlier in this Part, architectural conformance provided one example of making intended relations authoritative. Provenance is useful for a particular version of that problem: the required relation concerns not merely which components interact, but the path through which a consequential operation was derived.

Some obligations are cheap to enforce by controlling access to a recognizable primitive. Filesystem and network access are examples. The environment can identify the APIs that perform those effects, designate the small portion of the system permitted to use them, and reject other callers with comparatively simple static analysis. The prohibited interaction has a recognizable surface.

Other architectural obligations do not have such a convenient boundary. Suppose all color decisions must pass through a canonical color model before document mutation. The unwanted alternative is not necessarily a call to one forbidden API. A future implementation may combine otherwise legitimate resolvers, representations, and mutation APIs into a new path that reaches the same consequential operation. Each call can be legal, and the path may not match any bypass anticipated by a source-time rule.

This matters in agentic engineering because agents readily invent new ways to interact with a system. A prohibition against yesterday's bypass need not prohibit tomorrow's variation. Enumerating forbidden routes can therefore become an open-ended enforcement strategy.

Provenance can reverse the enforcement problem. Instead of trying to recognize every way an operation might have bypassed the sanctioned abstraction, require the consequential operation to carry evidence that it passed through that abstraction.

The sanctioned abstraction can issue an opaque provenance value that ordinary callers cannot construct. The consequential boundary then admits the operation only when that provenance is present and valid:

$$\text{admit } (x) \Leftrightarrow \text{valid } (x) \wedge \text{sanctionedProvenance } (x)$$

Figure 3.3-5 traces the single sanctioned path: an operation reaches consequence only by deriving through the sanctioned abstraction, which supplies the provenance the boundary demands.

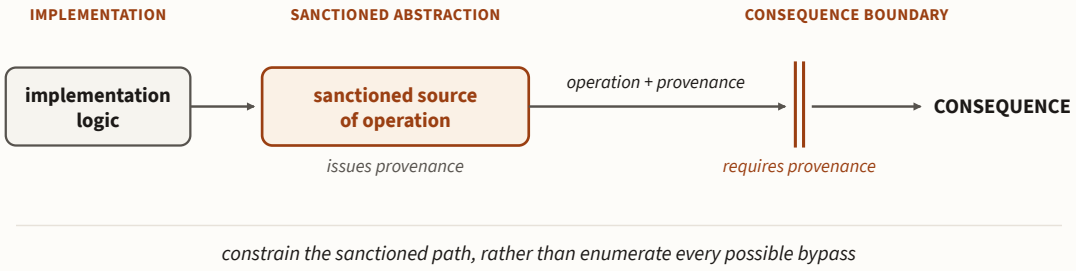


Figure 3.3-5: Provenance-carried authority. *A consequential operation must derive through the sanctioned abstraction, which attaches provenance to the resulting operation. The receiving boundary requires that provenance before permitting consequence. The mechanism removes the degree of freedom to obtain the operation through some other implementation path, without requiring the environment to enumerate those alternative paths.*

An agent remains free to invent new internal implementations, but invention does not create a new route around the obligation. A newly synthesized path that bypasses the

sanctioned abstraction arrives at the boundary without valid provenance and is refused. Enforcement therefore depends on recognizing the allowed derivation, not enumerating all possible forbidden derivations.

That distinction suggests when the pattern is useful. If consequential interaction occurs through a small, statically recognizable primitive surface, restrict that surface directly; simpler mechanisms are preferable. Provenance becomes attractive when the obligation concerns how an otherwise ordinary operation was derived, and alternative derivation paths are numerous, compositional, or difficult to identify cheaply from source.

The pattern is therefore not principally about attribution. It is a way to make a derivation path authoritative. A source-time rule can require known sanctioned implementations to emit provenance; an authoritative boundary can additionally require every consequential operation to possess provenance that only the sanctioned path could have produced. The two controls address opposite sides of the relation: one checks known producers; the other refuses unknown routes to the consumer.

The rule of thumb is short. **When the forbidden paths are enumerable, ban them. When the allowed path is easier to characterize than all possible bypasses, have the allowed path issue the provenance required at the boundary.**

Generative implementation sharpens the choice. An agent is unusually capable of exploiting degrees of freedom left open by the engineered environment, including finding valid implementation paths its designers did not anticipate. A blacklist of forbidden routes therefore becomes systematically less attractive as the space of possible realizations grows.

Provenance-carried authority removes a particular degree of freedom: a consequential operation must derive from the sanctioned source or path. Other implementation choices can remain open, but an agent cannot invent a new source of that operation and have it acquire the same authority merely because the resulting value is otherwise valid. The environment need not enumerate those alternative realizations; the consequential boundary requires provenance that only the sanctioned derivation can supply.

This is a graph-shaped variation on architectural enforcement. Layering constrains which components may interact across a layered structure. Provenance-carried authority instead constrains a particular derivation through an otherwise richer interaction graph: many implementation paths may exist, but only a path through the sanctioned node can produce an operation with authority to cross the consequential boundary.

3.3.7 What Counts as a Mechanism

Keep **policy**, **guidance**, and **governance mechanism** separate. A model file can state policy; a playbook can guide an agent; neither becomes an authoritative mechanism merely by existing or by appearing in an agent's context. Authority begins when something in the environment actually constrains the work, produces evidence, evaluates that evidence, or controls admission. The model states the obligation; the consuming mechanism gives it authority.

The same artifact can play several roles. A structured model injected into context guides reasoning. A validator reading that same model uses it as the obligation against which observed code is judged. A generated permission table turns the same modeled relation into a constraint. The model is the representation; the consuming machinery supplies the authority.

Implementations routinely bundle roles. A test runner may sense, validate, and gate; a narrow interface may constrain while emitting provenance. Name the functions inside the bundle rather than stretching every influence into a “soft constraint” or “soft sensor.”

Some mechanisms can be designed before the first autonomous change; others become visible only after the environment fails. The next chapter shows how those failures become durable structure.

3.4 Growing the Governed Environment

No engineered environment begins with every future obligation already represented and mechanized. Some obligations are known in advance: an API must require authentication, a state transition is forbidden, a format mutation must cross a particular seam. Others become visible only when the work exposes a failure nobody represented or controlled. A governed environment acquires durable structure in two ways: **ex ante from obligations already known, and ex post from failures and surprises.**

The ex post route is **governance conversion**. A failure exposes a recurring class, engineering identifies what was missing, and the environment encodes the lesson as durable structure. Later work inherits the result. The repair fixes this instance. Conversion changes the environment.

> **Inset — Failure modes before and after failure** ENGINEERING
>
> Engineering need not wait for a failure to learn how a system can fail. Techniques such as Failure Modes and Effects Analysis (FMEA) work ex ante: engineers identify possible failure modes, consider their effects, and use that analysis to decide where prevention, detection, mitigation, or other controls are warranted. Known obligations can therefore produce durable engineering structure before experience forces the issue.
>
> Governance conversion works in the complementary direction. A failure or consequential surprise supplies evidence that something was missing. Engineering identifies the failure mode behind the instance and asks what durable structure should address the class: a model, obligation, sensor, validator, constraint, gate, or other mechanism. The repair fixes the instance; conversion changes what future work inherits.
>
> The distinction is about how the need became known, not about the resulting mechanism. Ex-ante analysis and ex-post learning may lead to the same control. Governance conversion names the ex-post route; it does not imply that engineers should wait for failures they can reasonably anticipate.

3.4.1 Ex Ante: Encode What You Already Know

Start with obligations you can state before the work begins and that are consequential, recurrent, or reusable enough to make durable. A required output shape may become a schema; a forbidden dependency may become an architectural relation plus a validator; a service boundary may become a permission or narrow API; a legal lifecycle may become a closed transition table. Encode recurring obligations when the environment can carry them more cheaply and reliably than repeated judgment.

Part IV turns this into a method for starting and migrating systems; here the distinction is simply that these obligations are known before failure forces the issue.

3.4.2 Ex Post: Convert What the Work Teaches

The harder half begins when the system surprises you. A failure may reveal a recurring failure mode: the environment lacks a representation, an obligation was never stated, the relevant evidence is not observable, a validator is too weak, or a judgment exists but carries no authority. **Do not assume every failure calls for another gate. Diagnose what was missing.**

The conversion can therefore land in several places:

1. **Missing representation / semantics → improve the model.** Make the fact or relation durable at the abstraction where the question becomes answerable.
2. **Missing obligation or tolerance → state the governing boundary.** A failure may show that variation treated as free was not actually acceptable. Make the invariant, policy, bound, or permitted set explicit enough for future work to reason against it.
3. **Missing evidence → add a sensor or trace.** Make the relevant state observable.
4. **Missing evaluation → strengthen the validator.** Turn evidence into a repeatable judgment.
5. **Missing consequence → add or strengthen a constraint or gate.** Make the verdict matter.

Viewed from Part I, these interventions do different things to the probabilistic surface. Better representation can make a satisfactory realization cheaper to find by reducing reconstruction or exposing the property more directly. Better evidence can make the realized state more legible. A repeatable validator can carry an evaluation independently of the producer, and a constraint or gate can enforce that verdict. Governance conversion is therefore not simply the accumulation of checks; it is the selective transfer of recurring engineering judgment from per-instance reasoning into durable structure.

This is why Modeling and Alignment form a feedback loop in practice. A failure can expose knowledge that remained implicit, or reveal that a choice treated as a degree of freedom actually carried a tacit obligation or tolerance. Modeling can make that distinction explicit; Alignment can then give the resulting obligation appropriate evidence and authority. The environment improves when recurring reconstruction or judgment moves into durable structure that later work can inherit. Figure 3.4-1 draws the diagnosis.

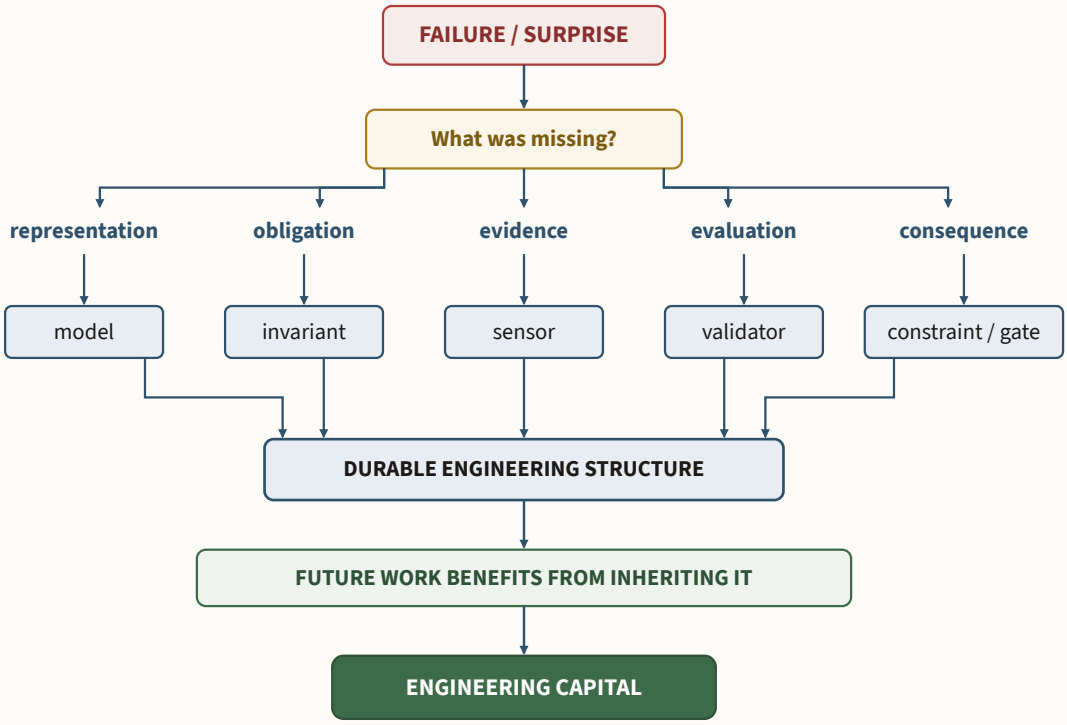


Figure 3.4-1: Governance conversion. *When a failure exposes a recurring failure mode rooted in missing representation, obligation, evidence, evaluation, or consequence, encode the missing structure at the appropriate layer. Future work inherits the result as engineering capital when that durable structure continues to lower future cost or uncertainty.*

3.4.3 Correct Judgment Without Consequence

A merge automation drained a backlog of agent worktrees into the mainline each tick. Most rebased cleanly; a fraction conflicted, and a single convenience option resolved those conflicts by taking one side automatically. On overlapping edits it silently dropped content. An audit already understood the danger: a guard detected when a conflict touched substrate-class files, judged the resolution unsafe, and logged it correctly, as a warning only. The operation proceeded, and the tree lost content. Figure 3.4-2 traces the cascade.

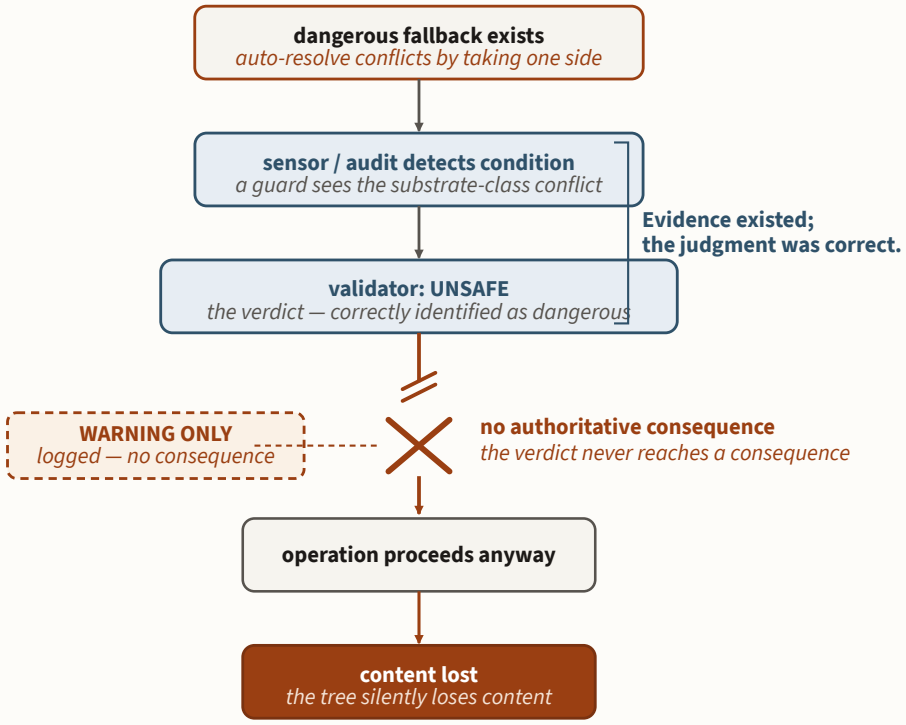


Figure 3.4-2: Correct judgment without consequence. *The system observed the dangerous condition and evaluated it correctly, but no authoritative mechanism acted on the verdict. The durable repair removed the unsafe capability rather than making the warning louder.*

The failure was not missing evidence. The system had evidence. Nor was evaluation missing: the audit correctly identified the danger. What it lacked was consequence. The warning described a condition the environment still permitted.

The durable repair did not make the warning louder. It removed the dangerous auto-resolution fallback and escalated ambiguous merges to a human instead. **Governance conversion can add a mechanism. It can also delete a capability the environment should never have offered.** Where a warning marks a known-dangerous condition intended to become binding, its temporary status should be explicit, with an owner and resolution condition.

3.4.4 Engineering Capital

Governance conversion motivates a useful accounting concept: **engineering capital**. The field has a mature metaphor for the opposite condition. Technical debt is a liability carried forward: an expedient decision today imposes costs on future change. Engineering capital is durable engineering structure that lowers the future cost or uncertainty of change.

The capital must be something future work can actually inherit. A model can eliminate repeated reconstruction; a constraint can prevent a recurring failure; a sensor can expose previously hidden state; a validator can make repeated evaluation mechanical; a gate can enforce that evaluation; a skill or runbook can package judgment for reuse. These assets change the starting conditions of future work. They earn a return when future work requires less reconstruction, repeated judgment, failure, or rework because the structure exists.

The engineered environment evolves with the system it governs.[15] Capital also **depreciates**. A model can drift from the system it represents. A validator can encode an assumption the architecture has outgrown. A sensor can become noisy; a gate can impose more delay than the failure it prevents; a once-important control can survive long after its obligation disappears. Maintenance preserves useful capital; reconciliation repairs assets that no longer match the system; retirement removes assets whose carrying cost now exceeds their return. This is why accumulation cannot be the objective. A governed environment with more mechanisms is not necessarily richer than one with fewer. What matters is the future engineering value those mechanisms continue to produce.

Debt makes future work pay again for today's expedience; capital lets future work inherit today's judgment. Governance conversion is one way engineering effort stops being consumed once and becomes reusable structure.

[15] Lehman's work on software evolution made the analogous observation about software itself: software used in a changing real-world environment must continually adapt, and its complexity tends to increase unless effort is spent maintaining or reducing it. MAGE extends that caution to the engineered environment surrounding the software. Models, validators, constraints, and procedures evolve too — and can become liabilities when they no longer fit what they govern. See Lehman & Ramil 2003[50].

3.4.5 Where Enforcement Stops

Not every quality goal should end in enforcement. A consequential concern may remain unrepresented, require expert judgment, be evaluated without enforcement, or be governed. Figure 3.4-3 shows the four possible stopping points.

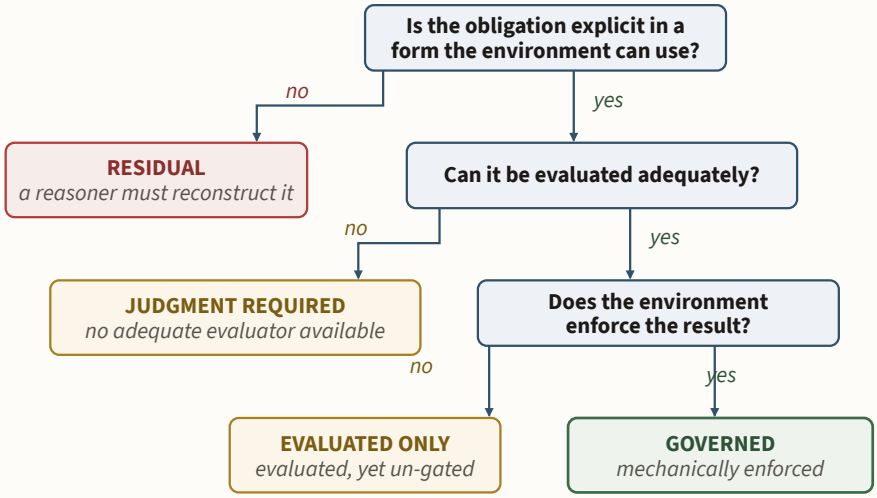


Figure 3.4-3: Where enforcement stops. *A concern may remain unrepresented, require judgment, be evaluated without enforcement, or be governed. These are design choices, not maturity levels; different obligations should stop at different points.*

Residual. The obligation is not explicit enough for the environment to use, so an engineer or agent must reconstruct the missing meaning.

Judgment required. The obligation is explicit, but available mechanisms cannot evaluate it adequately. Architecture may be represented while a design trade-off still requires expert judgment.

Evaluated only. The environment can produce and evaluate relevant evidence, but does not enforce the resulting judgment.

Governed. The environment enforces the obligation in a form appropriate to the engineering decision: an action may be constrained, a violation refused, or admission made dependent on the verdict.

The same restraint applies to tolerance. Making an acceptable boundary explicit does not imply that the environment should tighten it, or even enforce it mechanically. A broad tolerance may be the correct engineering decision, and variation inside it remains legitimate realization freedom. Stronger Alignment means more dependable enforcement of the obligation engineering actually chose, not progressively narrowing the acceptable realization space.

DocAble contains examples of each non-governed stopping point, each for a different reason. Some cost controls are deliberately non-binding because imposing a hard daily budget remains a policy choice. DocAble measures governance conversion but does not target a required rate, because doing so would invite gaming. Some accessibility judgments remain outside deterministic enforcement because the available representation does not contain enough semantics to evaluate them reliably. **Non-enforcement is sometimes the engineered result.**

The mechanism that discovers an obligation need not be the mechanism that ultimately carries its authority. DocAble's primitive-density lint remained an audit signal because high density was only a smell. The migration it provoked produced more specific types, boundaries, and drift controls whose predicates could justify stronger consequences. Part IV follows that migration in detail.

Different obligations should stop in different places. Governance conversion closes recurring gaps when doing so is worth the cost. It also creates the next problem: the environment now contains an expanding portfolio of controls, and those controls can themselves become inconsistent, redundant, or orphaned.

3.5 When Controls Become a System

As governance mechanisms accumulate, they become control machinery: the constraints, sensors, validators, gates, permissions, and other mechanisms through which the environment enforces its obligations. The machinery can fail even when every individual control is reasonable. Two controls can impose incompatible demands; their execution order can be ambiguous; an obsolete control can survive after its purpose disappears; an important obligation can remain uncovered. None of these failures is visible by reading one mechanism's code.

At that point, the controls themselves have become an engineering system. They must be understood and maintained as one. MAGE asks three questions: **What controls exist? How do they interact? Which obligations do they cover?** These questions produce three useful views: a catalogue, an interaction graph, and a coverage relation. Represent that machinery explicitly, so engineers and tools can reason about how its controls compose and what obligations they cover—and, once it becomes consequential, hold it to its own obligations.

3.5.1 Catalogue the Controls

Start by giving each control a stable identity: the event or boundary at which it acts, the resources it reads or changes, the obligation or failure class that justifies it, and its authority role or roles. The catalogue gives the next two views stable identities to work from instead of forcing engineers and tools to rediscover controls from scripts, hooks, CI files, and institutional memory.

A catalogue alone cannot tell you whether two individually sensible controls conflict, nor whether an important obligation has no control. Those are relational questions. The first lives on the edges between controls.

3.5.2 Model Interactions Between Controls

Two collisions from the case study make it concrete. In the first, one control required a squashed commit while another treated the resulting broad diff as unauthorized scope expansion. Each was reasonable alone; together they imposed contradictory demands on the same commit set. In the second, two hooks fired on the same turn-end with no order declared between them, each able to block, and their composition was left implicit. Neither collision lives in any one control's code. Each lives in the space *between* two controls. That relation had not been represented. Figure 3.5-1 draws it.

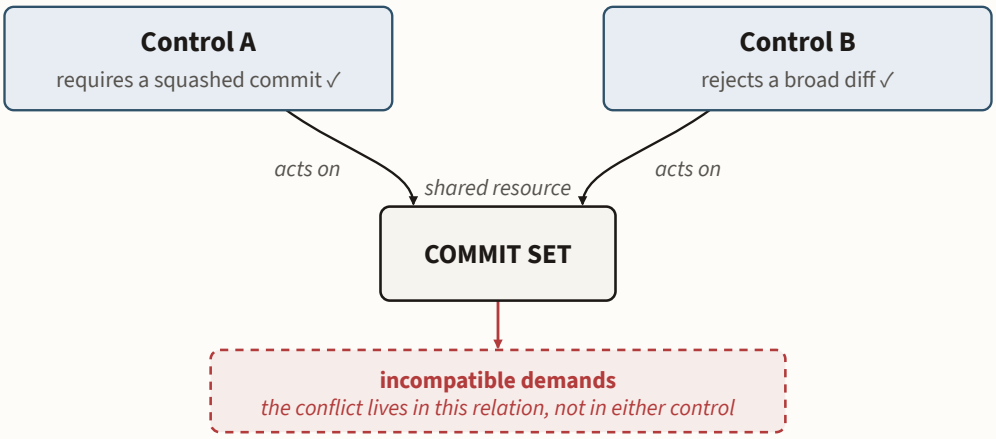


Figure 3.5-1: The conflict is the edge. *Two controls may be individually valid yet impose incompatible demands on the same resource. The interaction belongs to the relation among the controls and shared resource, not necessarily to either control's implementation.*

The interaction graph uses controls as nodes and shared resources as the principal join key. A resource may be ordinary — a lock, a file, a commit-set, a deployment slot — or operational, such as a lifecycle event or a context budget. Controls need not call one another to interact; they only need to make potentially incompatible demands on the same modeled resource.

DocAble's implemented governance graph is intentionally partial. It models only the high-value interactions for which recurring questions justified explicit representation, including those around the orchestrator lifecycle.

A shared resource is a conservative signal of interaction, not proof that every conflict has been captured. Within the represented surface, it lets engineers and tools ask useful questions before the controls interact in production work: which controls fire on this event, which touch this resource, which pairs have no declared order, and which known combinations impose incompatible requirements.

Some candidate conflicts are mechanically decidable: a lock cycle, two exclusive writers, or two same-slot mechanisms with no permitted ordering. Others are semantic. A static check can **sense** that two controls touch the same commit-set; a validator — or a human reviewer

— must decide whether their requirements actually conflict. Again the four roles matter: producing evidence of an interaction is not the same as judging the interaction invalid. In the motivating commit-set collision, the machine could identify the shared resource, but judgment was still required to recognize the requirements as contradictory. The graph lets the environment surface the interaction for evaluation before the controls collide during work.

3.5.3 Trace Obligations to Mechanisms

Interactions are only half of governing the control machinery. A catalogue tells you what controls exist; an interaction graph tells you where controls may collide. Neither tells you what *should* be governed but is not. For that, join the mechanisms to the obligations or failure classes that justify them.

The result is a small requirements-traceability problem. An important obligation with no adequate mechanism is a coverage gap. A mechanism with no current obligation, failure class, or documented purpose is an orphan. The useful question is not "how many controls do we have?" It is whether consequential obligations have appropriate coverage and whether every surviving control can still explain why it exists. Figure 3.5-2 sets the three views side by side over the same control machinery.

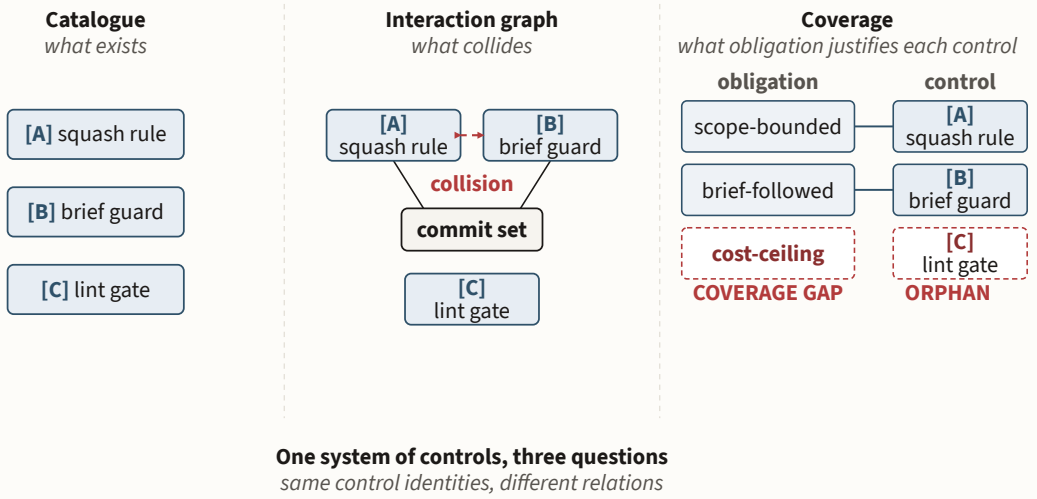


Figure 3.5-2: Three views of the control machinery. *The catalogue identifies controls; the interaction view relates controls through shared resources; the coverage view relates controls to the obligations that justify them. Stable identities connect the three views while letting each answer a different engineering question. A missing mechanism appears as a coverage gap; an untraced mechanism appears as a candidate orphan.*

Two findings do the diagnostic work, and neither verdict is automatic.

- **Coverage gap.** A consequential obligation for which the intended evidence or authority is not adequately supplied.

- **Orphan.** A mechanism that no longer traces to a current obligation, failure class, or documented purpose. An orphan is a finding to reconcile, not proof that the mechanism should be deleted.

3.5.4 Add, Reconcile, and Retire Controls

Governance conversion and coverage auditing provide complementary pressures. Conversion adds durable structure when experience exposes a recurring gap; coverage auditing finds controls that are missing, redundant, or obsolete.

The same audit also pushes against permanent accumulation. A control whose purpose has disappeared should be reconciled or retired; two controls that now overlap may be consolidated; a mechanism whose maintenance cost exceeds the failure it prevents may no longer be worth carrying. This echoes Section 3.4: **engineering capital is not the number of controls.** A control that no longer produces value has depreciated into overhead; the task now is to reconcile or retire it.

The control model must remain connected to the controls it represents. Where DocAble models controls explicitly, entries can be anchored to their implementations and checked for drift, so a control the rule cannot place fails explicitly rather than dropping into a silent default bucket. This applies the broader rule from Part II: if engineers and agents are going to reason through a model of the control machinery, the model must stay connected to the territory it claims to describe.

Do not recursively govern every relation merely because it can be modeled. Extend the control model only when a recurring engineering question or observed failure justifies the added machinery.

The control-machinery model is grounded primarily in the originating case.[16]

> WORKED EXAMPLES
>
> **Two Safety Rules.** A building's fire system unlocks doors when it detects smoke; its security system may lock those same doors under an intrusion policy. Each rule is sensible in isolation while the pair demands incompatible states of the same resource. The contradiction belongs to the interaction, not to either rule's own logic.
>
> **DocAble.** DocAble models its governance mechanisms as a system rather than only as individual scripts. Controls are anchored to their implementations, interaction relations identify shared resources, and coverage traces controls back to the obligations or failures that justify them. The model does not make every possible conflict mechanically decidable. It exposes enough structure to ask, before the next collision, which mechanisms interact and which important obligations have no adequate coverage.

[16] I found no comparable control-machinery model in the external systems reconstructed for Part V.

> **Takeaway.** Governance conversion adds durable engineering structure; governing the control machinery keeps that structure coherent. Catalogue the controls, expose their interactions, trace them to the obligations they serve, and retire machinery that no longer earns its cost.

Alignment does not require comprehensive Modeling: permissions, types, APIs, and local gates can bind actions and artifacts directly. Explicit models expand the semantic reach of that authority.

Part 3 developed Alignment in five moves: locate the boundary where the obligation can be enforced; state an independent obligation, including its tolerance where acceptable variation matters, and place its mechanism where that obligation becomes legible; distinguish constraints, sensors, validators, and gates; convert recurring failures and judgments into durable structure; and govern the resulting control machinery as a system in its own right.

Modeling and Alignment are now both on the table. That still is not a method. The remaining problem is judgment: when a question deserves a model, when an obligation deserves authority, how work should move through the environment, and what to do when the available representation or control is insufficient. Part IV turns those decisions into method.

Summary

1. **Authority makes obligations enforceable.** Guidance can influence a reasoner; an authoritative obligation is one the engineered environment can enforce.
2. **Place the mechanism where the obligation becomes decidable.** Constraints act before work proceeds; sensors produce evidence; validators evaluate evidence; gates control admission. Prefer the earliest boundary where the environment has enough meaning and evidence to decide the obligation adequately.
3. **Correspondence is not correctness.** Models, implementations, and evidence can agree while being wrong together. Independent obligations and evidence are needed to distinguish faithful realization from mutually consistent error.
4. **Convert recurring judgment into durable structure.** When failures or repeated engineering decisions reveal knowledge future work should inherit, encode that knowledge into models, mechanisms, architecture, procedures, or evidence. When later work benefits from that structure, it becomes engineering capital.
5. **Govern the governance machinery.** Constraints, sensors, validators, and gates form an engineering system of their own. Their coverage, correspondence, cost, authority, and failure modes must themselves be understood and controlled.

ALIGNMENT PRINCIPLE

Give engineering obligations authority by encoding them into mechanisms that constrain actions, produce evidence, evaluate that evidence, and control admission.

An obligation is authoritative when the engineered environment can enforce it.

Interlude

One Problem, Many Models

Parts II and III separated Modeling from Alignment so that each could be examined clearly. In engineering work, the two are often entangled as new knowledge shapes the engineer's understanding of a problem. This interlude illustrates that process through one problem in DocAble.

The engineering techniques in this interlude may be familiar to you. An experienced engineer might recognize several of them immediately: reduce a working set, separate structure from payload, represent dependencies explicitly, analyze a critical path, bound concurrency, and measure the result. MAGE does not claim these techniques as new, nor does it replace the experience that helps an engineer recognize when to use them.

What MAGE does offer is a way of approaching problems with GenAI that affords fast iteration and sound, drift-resistant results. The engineer does not need to know the final model or architecture in advance. A problem may begin as an observation: a file fails, a service slows down, or a new requirement exceeds the system's current range. Modeling makes one consequential relation explicit; analysis suggests what to change or measure; implementation supplies new evidence; and that evidence may expose the next thing worth representing.

This interlude follows that loop from a memory failure through several changes in representation and architecture. The interest is not whether an experienced engineer could have arrived at the same design. It is whether the engineering process can discover, test, and preserve such knowledge systematically while implementation moves at agentic speed.

I. One Document, One Worker

DocAble originally treated an entire document as one unit of execution. A long-running Kubernetes worker opened the document, held the state needed for remediation in memory, modified it, and produced the result. Figure 3.5.1-1 traces how that execution unit later changed.

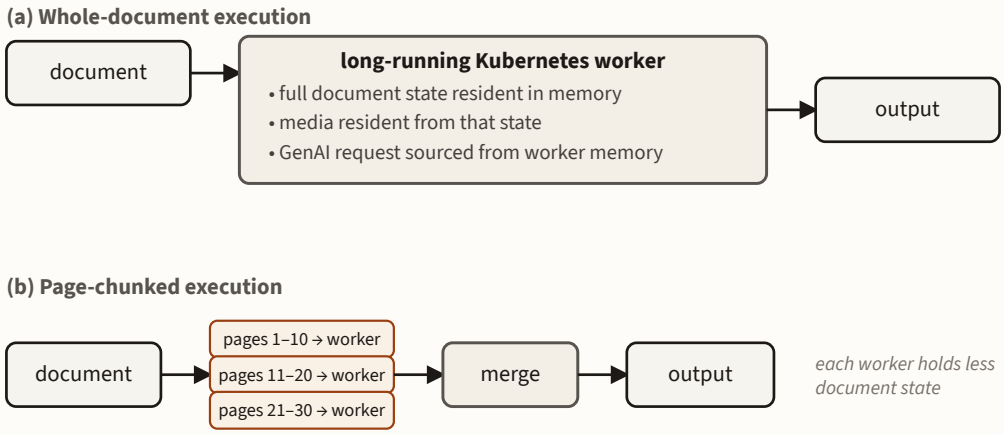


Figure 3.5.1-1: Two execution units for document remediation. *(a) Whole-document execution keeps the artifact's working state and media resident in one long-running worker. (b) Page-chunked execution divides the document across workers, reducing the working state and memory required by any one worker.*

The document therefore served as both the user's artifact and the worker's in-memory unit of computation. Media needed by GenAI operations was read from that resident representation and sent to the model as needed.

For the small presentations used to develop the system, this worked well. The architecture was simple, and the document was a natural boundary: a user submitted a document and expected a document back. Nothing in the observed workload justified another unit of execution.

Experience eventually expanded the workload. Real teaching presentations carried more slides, images, and other media, and handling the whole document in one process began to put enough pressure on worker memory that the execution unit needed to shrink.

II. Pages as the Unit of Scale

As documents grew larger, whole-document execution put increasing pressure on worker memory. DocAble therefore introduced ChunkWorker, which divided a document into page ranges (Figure 3.5.1-1, panel b).

The new representation treated contiguous pages as independent work partitions. For a presentation, a chunk contained a range of slides together with the document state needed

to remediate them; the system later recombined the resulting patches into the complete artifact.

Each worker now handled only part of the document, and multiple chunks could run concurrently. More importantly, chunking reduced the memory required by an individual worker. Memory affects both infrastructure cost and cold-start behavior, so a smaller worker was useful even for documents that did not strictly require chunking to fit.

Chunking worked across a much broader workload: ordinary teaching presentations and hundreds of documents from multiple universities. Page ranges gave each ChunkWorker a practical bound on its working state, and experience provided no reason to replace that design merely because another architecture was possible.

This architecture accommodated the inputs encountered so far. The page boundary gradually became part of how DocAble was understood: a document was divided into chunks because chunks were how large documents were processed. A roughly 250 MB media-heavy presentation eventually exceeded that operating regime.

Page boundaries were not the smallest possible partition. A sufficiently large slide could itself be divided into regions, shapes, media objects, or individual remediation operations. That would reduce the working set further. But finer partitions also impose costs: more coordination, more reconstruction, and less surrounding context available to each operation. Eventually the partition begins to cut across the semantic relationships needed to reason about the artifact.

This is the reasoning-horizon problem from Part I in architectural form. An agent can reason only over the information made available to it. Shrinking an execution unit may reduce its resource requirements while simultaneously shrinking the semantic context within which a judgment is made. Page chunking therefore had no hard scaling wall. It had an increasingly unattractive tradeoff: continue subdividing the artifact and pay both systems overhead and semantic loss, or reconsider why so much of the artifact had to reside in the worker at all.

The 250 MB presentation made the second question worth asking.

III. A New Class of Input

A 250 MB PowerPoint file does not imply that remediation needs 250 MB of document content resident in memory.

Modern Office documents use Office Open XML (OOXML). A `.pptx`, `.docx`, or `.xlsx` file is a package containing many parts. Much of its semantic structure is XML: text, slides, relationships, identifiers, and references among objects. Large binary assets such as images and other media live alongside those XML parts inside the package.

The file therefore holds two broad kinds of information. One kind describes document structure: slides, shapes, text, relationships, identifiers, and references. The other is media payload: images, animated GIFs, movies, and other embedded binaries.

The structural portion is comparatively small. It is largely text, and even an unusually large Office document may require only low megabytes to represent its semantic skeleton. The

media payload can be hundreds of megabytes. Most remediation operations need only the structural portion; they may reach into the media, but not every media byte need be resident at once.

Until this point, DocAble had largely accepted the storage representation supplied by the document format as its working representation. For Office documents, that meant opening the OOXML package through the document library and letting its structure and media participate in the in-memory object graph; other formats followed analogous format-native representations. That was reasonable while the input sizes fit the worker. Handling a roughly 250 MB media-heavy file without allocating correspondingly large workers required us to reconsider the arrangement itself.

The existing representation coupled the two, and Figure 3.5.1-2 contrasts that arrangement with the one that replaced it.

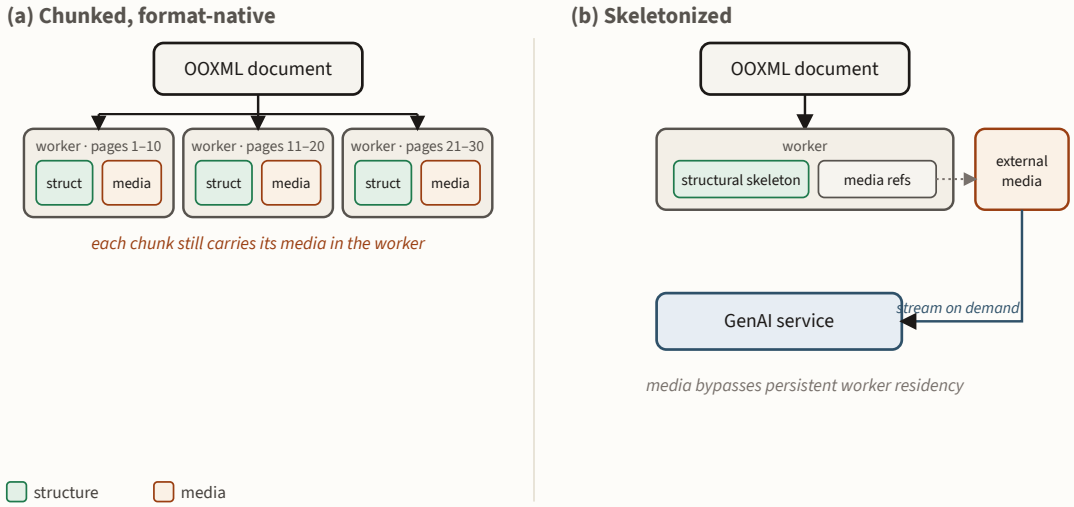


Figure 3.5.1-2: Chunked residency versus skeletonized residency. *(a) Page chunking limits how much document state and associated media a worker holds at once, but media remains part of the worker's resident representation. (b) Skeletonization keeps only document structure and media references resident. Individual media objects stream from external storage to the consumer on demand, so total media size no longer determines persistent worker memory.*

So the architecture carried an implicit scaling relation: total media size drove resident worker state, and resident state drove the risk of running out of memory. Finer chunking could continue reducing how much of the document met this relation at one time, but it could not remove the relation itself—and increasingly fine partitions would introduce coordination costs and erode the semantic context available to remediation. On the observed workload that made no difference; the new file made it decisive.

A residency model made the alternative visible (panel b): the structural pipeline could operate on a lightweight representation while media lived in object storage, and operations that needed media could reach it by reference.

The 250 MB requirement had therefore exposed a relationship worth changing: the storage representation was also determining the worker's memory representation.

IV. Remove Media from the Working Set

The memory problem came from coupling document structure to media residency: total media size increased the worker's persistent working set even though most remediation operations needed only the document's structural skeleton. Skeletonization removes that coupling. For OOXML documents, the pipeline streams through the package, moves media into external storage, and leaves stable references in a much smaller structural skeleton.

Remediation then operates on the skeleton. If a GenAI operation needs an image, the worker follows the reference and streams that media through the network to the GenAI service without first materializing the document's complete media payload in worker memory. Recombination follows the same discipline in reverse. The worker no longer needs memory proportional to total media size.

The corresponding memory invariant states the property directly:

```
peak worker RSS <= C
```

where, within the declared operating envelope,

```
C =
    rolling stream buffer
  + structural skeleton
  + handles
  + remediation working set
```

and C does not grow with total media size.

The claim is stronger than the empirical claim available from chunking. Chunking had accumulated evidence that increasingly large documents worked, but those observations could not establish behavior for every larger media payload. Skeletonization supports a structural argument instead: within the declared operating envelope, peak worker memory is bounded independently of total media size.

No stage in the modeled path retains the complete media payload. OOXML media moves through bounded streaming buffers, the structural skeleton stays small relative to the media it references, and a media consumer holds bounded active state rather than accumulating the document's media. The guarantee depends on those assumptions; in particular, the write path must stream too. A supposedly streaming architecture that buffers the complete payload before writing it has only moved the memory spike.

The implementation enforces the assumptions behind the bound as invariants; measurement checks that the realized system behaves as the model predicts. The PDF path already shows approximately flat peak-RSS scaling across a count-scaled media sweep when each

object is flushed; the corresponding OOXML media-size sweep remains a designed validation step.

None of this proves that arbitrary documents can never exhaust memory. A document could expose another scaling dimension: pathological structure, an individual object beyond the supported bound, excessive concurrency, or something not yet modeled. It closes a narrower class: within the modeled envelope, total media size no longer determines worker memory.

V. Once Memory Is Solved, Why Are We Chunking?

Page chunking had served two roles at once: it partitioned execution and bounded each worker's memory. Skeletonization largely removed the memory requirement. That left a separate question: what should determine the unit of execution?

A page tells us where content occurs, not which computations depend on which others. DocAble already represented remediation as a dependency graph. Consider alt-text generation as a concrete example. Each page can first be analyzed independently. Those analyses feed an operation that builds shared document context. Once that context exists, the GenAI queries that generate alt text for individual images become ready to execute independently. The actions and their information dependencies form a remediation graph; the currently executable actions form its frontier.

The same graph admits different execution policies. Figure 3.5.1-3 shows the distinction.

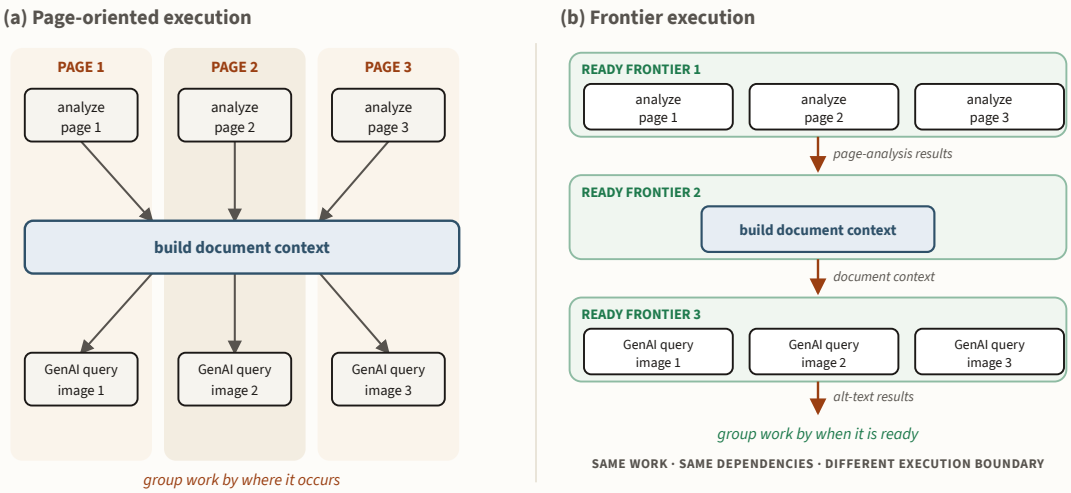


Figure 3.5.1-3: Two execution policies over the same remediation graph. *The graph contains actions—page analysis, construction of document context, and GenAI queries—and information dependencies among them. (a) Page-oriented execution groups actions according to where their artifacts occur, so a page partition can span several dependency levels. (b) Frontier execution groups actions according to when they are ready: page analyses execute first, their results enable construction of document context, and that context enables independent GenAI queries for alt text. The work and its dependencies are unchanged; only the execution boundary changes.*

The graph explains the ordering. Independent work can run together; a consumer waits only for information it actually needs from upstream. Pages do not create those dependencies. Information does. Expressed as a directed acyclic graph, the work currently ready to execute forms a frontier: the nodes with no unmet dependencies. A worker executes that frontier, materializes its outputs, and thereby exposes the next frontier.[17]

This reframes ChunkWorker. Page-oriented and frontier-oriented execution are not different remediation graphs. They are different ways of partitioning the same graph. Page execution groups work by where it occurs; frontier execution groups work by when it is ready.

Skeletonization removed the memory pressure that had made page ranges the natural execution boundary, so page identity no longer needs to determine execution. It can survive as metadata on graph nodes—useful for locality, debugging, rate limiting, or failure isolation—without remaining the fundamental unit of executable remediation.

For the present problem, the architecture therefore needs only a clean seam between the remediation graph and its executor: the graph supplies ready work and its dependencies; the executor consumes them. The executor can schedule by frontier now and exploit page locality later if measurements justify it. There is no need to build a strategy hierarchy merely because two policies are conceivable. The graph/executor seam preserves the degree of freedom without spending it. In the realized architecture, that execution-driving representation remains distinct from richer analytical projections built over the same computation identities. The distinction matters later: a model may govern realization while another view of the same structure serves analysis.

[17] This is ordinary topological traversal of a directed acyclic graph, familiar from compilers, build systems, and dependency schedulers.

VI. Analyze Before You Build

Representing remediation as a dependency graph does more than provide an execution plan. The graph can be analyzed before an executor — or a proposed architectural change — is implemented. Figure 3.5.1-4 shows three such analyses.

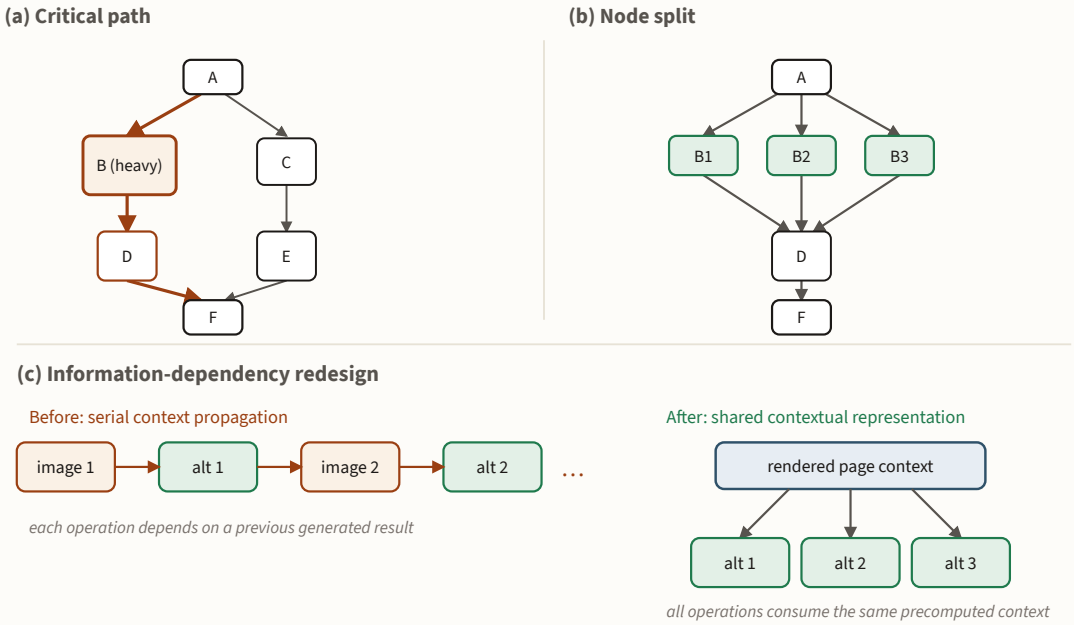


Figure 3.5.1-4: Analysis before implementation. *(a) In a weighted dependency graph the heaviest chain is the critical path, which sets the latency floor and names an optimization target. (b) Decomposing the dominant node exposes parallel children and can shorten the path, a possibility visible in the model before the split is built. (c) A serial information dependency can be removed by changing the representation supplied to each operation: instead of sibling-to-sibling context propagation, each image consumes one shared precomputed page context, so the alt-text operations run independently.*

Given the weighted graph in panel (a), the model reports which operations may run concurrently, which must wait, and that adding workers cannot shorten a dependency edge; the heaviest chain from start to finish is the critical path. If one node dominates that path, the model has found an optimization target, and the next question is structural: must it be one node? If its semantics permit decomposition, splitting it can expose parallelism and shorten the path (panel b), discoverable from the model with no implementation of the split. The graph cannot promise the optimization will pay off, since smaller operations may add coordination cost, lose batching efficiency, or contend for another resource; but it names the candidate and the quantities that decide whether it helps.

Dependencies can be redesigned

The same analysis exposed a problem in contextual image description (panel c). If each image description consumed the description generated for the preceding image, the operations formed a serial SCAN: for N images the dependency depth approaches N, so if each

GenAI call takes roughly t, the serial floor grows as about N·t, and more workers cannot remove it. But the requirement is not to generate alt text serially; it is to give each image enough surrounding context to produce an adequate description. Rendering the surrounding pages up front and treating those thumbnails as a shared visual IR supplies that context: each image then consumes bounded neighboring visual context instead of a sibling's generated alt text, the payload grows but the dependency disappears, and the operation stays a parallel MAP. The analysis also clarified the requirement. Good alt text needs sufficient visual context; it does not require serial generation. That context can itself be represented and reused, just as textual document context already is. The model changed the proposed implementation before that implementation existed.

Predicting latency

The same graph supports a first-order latency model. For the representative workload used during this analysis, the relevant GenAI chain had three dependent levels whose per-call times add to an irreducible floor (Figure 3.5.1-5).

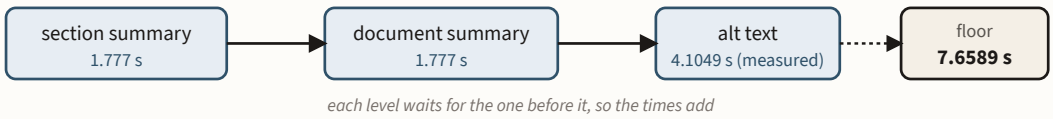


Figure 3.5.1-5: The GenAI dependency chain and its latency floor. *A section summary feeds a document summary feeds alt text; because each level waits for the one before it, their per-call times add to an irreducible floor of 7.6589 s. The vision value is measured; the text-summary value is estimated.*

The current in-process design had local concurrency of six, so a 37-image alt-text frontier required seven waves, for a predicted total of about 32.788 s. A distributed frontier executor with modeled cloud concurrency of 64 could execute the 37-wide level in a single wave, but would pay cold-start and round-trip taxes for each graph round — about 11.4589 s under the current estimated parameters.

The local concurrency limit of six was not merely a scheduler setting. In the ChunkWorker architecture, each concurrent worker opened and manipulated its own format-native document state, including associated media objects. Increasing concurrency therefore multiplied pressure on CPU and, especially, the machine's working memory. The concurrency cap kept that aggregate resource demand within the practical operating envelope of the host.

Skeletonization changes that constraint as well. Once workers retain the structural skeleton while media streams on demand, concurrency no longer multiplies the same large resident media state. The frontier executor can therefore exploit graph width without requiring every concurrent operation to carry a page chunk's format-native working set.

The model therefore predicted roughly a 2.9× advantage for frontier execution on that workload: the existing design pays about 24.6 seconds in width serialization, while the

distributed design trades that for about 3.3 seconds of modeled round overhead. Several cells remain estimated and need staging measurements before any production cutover, but the number is useful before it is perfectly calibrated. It says why the candidate should win, and it says what to measure next.

Predicting the cost of a design change

Models can also answer questions about designs that do not exist yet. Suppose an engineer proposes another intermediate representation. Adding that IR creates a new fact and perhaps another dependency edge; the critical-path model can insert the proposed node into the DAG and recompute:

- Δ critical-path floor
- Δ above-floor latency
- Δ execution rounds
- Δ peak width
- whether the proposed node lands on the critical path

An IR off the critical path may add no serial latency; one that deepens the critical chain adds its call time to the floor and may add another distributed round. The model therefore supports an explicit marginal-cost query for a proposed representation — again, before any implementation. This is ordinary engineering analysis, applied before code exists.

Use prediction error as evidence

Implementation eventually supplies measurements, and the useful comparison is not simply whether the result was fast. A residual — the gap between prediction and observation — turns into an engineering question, and it drives a loop that runs both before and after implementation (Figure 3.5.1-6).

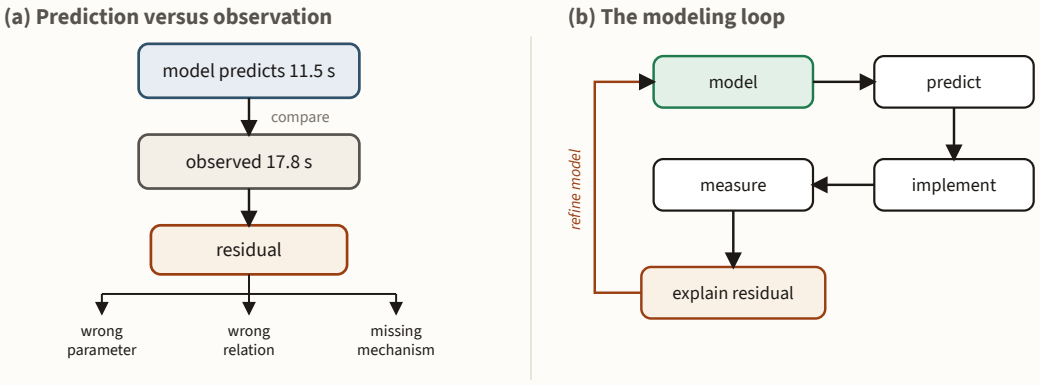


Figure 3.5.1-6: Using prediction error as evidence. *(a) The gap between a model's prediction and the observed measurement is a residual, pointing to a wrong parameter, a wrong relation, or a missing mechanism; the numbers are illustrative. (b) Prediction and measurement create a feedback loop: explain the residual, refine the model, and test the revised model against subsequent observations.*

The residual creates an engineering question. The queue overhead may have been underestimated. Two apparently independent nodes may share a serialized resource. Media loading may add a cost the latency model omitted, or the graph itself may be missing a real dependency. A prediction that matches observation provides evidence that the model captures the dominant mechanisms; a miss identifies where the model or system deserves investigation. Modeling thus contributes both before and after implementation: before, it compares designs and exposes optimization targets; after, disagreement between prediction and observation helps find what the model missed.

VII. The Resulting Model Family

The large-file requirement did not produce one model. It produced a connected family, each member answering a different question. Table 3.5.1-1 collects them.

Model	Question
Media residency	What must reside in worker memory? How should memory scale with media size?
Batch execution	How do batch size and concurrency affect latency, cost, and truncation risk?
Critical path	What sets the latency floor? Which nodes are optimization targets?
Distribution	Where should ready work execute? What coordination costs does that introduce?
Remediation graph	What work exists, and what depends on what?
Fact / IR catalogue	Which derived facts feed later computations? How deep is the information-dependency chain?
Image fidelity	What representation of media is required, and what does rendering it cost?

These models take different forms because they answer different questions. Some are typed graphs, some quantitative functions, some declarations of entities and invariants. They stay separate and compose through shared identities and explicit relationships: the remediation graph and fact catalogue supply the static dependency structure; the batch model supplies per-call cost; the critical-path model combines them into latency; the distribution model supplies the execution topology the coordination taxes are charged against; and the media-residency model supplies the independent memory bound that makes the execution design affordable.

Together they answer questions that would otherwise invite implementation experiments: what happens if we add an IR, where the critical path lies, whether splitting a node exposes useful parallelism, at what frontier width distributed execution beats local execution, how peak memory changes as media grows, and which assumption a measurement would have to violate for a prediction to be wrong. Some answers are structural and some require calibrated measurement; a good model says which is which.

From examples to a bound

The memory story shows the progression clearly:

1. “My simple decks work.”
2. “My real decks work.”
3. “Hundreds of documents from several universities work.”
4. “The 250 MB media-heavy deck does not.”
5. Model the relationship that failed.
6. Remove total media size from the worker-memory equation.
7. Enforce the assumptions that the bound requires.

Steps 1–3 accumulate empirical evidence; step 4 exposes the boundary of that evidence. Steps 5–7 do something different: they seek a property over a declared class of inputs. The resulting guarantee is deliberately narrow — within the modeled envelope, total media size does not determine peak worker memory. Other unknowns remain possible; a future artifact may expose another structural dimension or another resource limit, and MAGE offers no reason to pretend otherwise. But if the next 250 MB deck arrives because it carries more media, the architecture should not have to learn this lesson again.

Three days

Laid out this way, the sequence looks like a substantial architecture project. It included the large-file requirement, skeletonization and streaming, typed-patch work, the reconsideration of ChunkWorker, remediation and fact graphs, quantitative analysis, and the chunkless execution design. It also included implementation and corrections when assumptions proved wrong. The elapsed engineering time was about three days, working inside the existing governed engineering environment.

The significance of that number is not that these are unusually difficult techniques, or that an experienced engineer could not have reached the same conclusions. It is that the models were cheap enough to participate in the engineering loop rather than follow behind it. Dependency graphs, performance models, data-flow models, invariants, and intermediate representations are established software-engineering tools. Their cost has long limited how aggressively engineers use them. A model that takes a week to build may describe code that changes the next day, so the rational response is often to model less, implement the candidate, and measure.

Agentic implementation changes that calculation—but only if engineering reasoning can move with it. In this example, models and implementation evolved together. The residency model stated the memory property the implementation had to preserve. The remediation graph showed that page chunking was one scheduling policy rather than the structure of remediation itself. The critical-path model rejected a serial SCAN before anyone built it and identified a wide node as an optimization target. When implementation disagreed with a prediction, the discrepancy became another engineering question rather than an invitation to keep patching until the tests passed.

None of those models needed to describe the whole system. Each needed to make one consequential question cheap enough to answer before implementation answered it by

default. Three days matters because, at that timescale, modeling and analysis can remain inside the implementation loop.

Finding the models

Part IV asks engineers to find the models their work needs. This example shows what that instruction means in practice. Nobody began with a plan to construct seven models. The whole-document architecture worked until larger workloads made memory consequential. Page chunking solved that problem across the workload then observed, but a media-heavy document exposed a remaining scaling relationship. Modeling residency led to skeletonization; skeletonization removed the memory rationale for page partitioning; the remediation graph then exposed causal structure; and critical-path analysis exposed latency limits and optimization opportunities.

The sequence was discovered, not prescribed. Each model appeared because the previous design exposed a question that implementation alone could not answer well. Some questions concerned structure: what depends on what? Some concerned bounds: what determines peak memory? Others concerned prediction: where is the latency floor, and would another worker help? The form followed the question.

This is why the process is not the linear one—design, then implement, then document with models—but a loop in which implementation, measurement, and analysis expose the next thing worth representing (Figure 3.5.1-7). A model need not describe the whole system, nor need it survive forever. It needs to make some consequential property easier to reason about than the implementation itself does. Some models become durable engineering capital because later decisions keep depending on them; others settle one question and disappear. Keep a model while the questions it answers justify its cost.

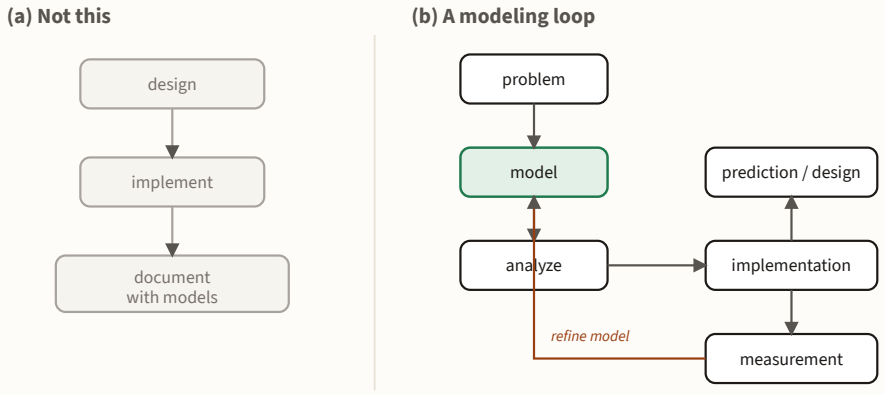


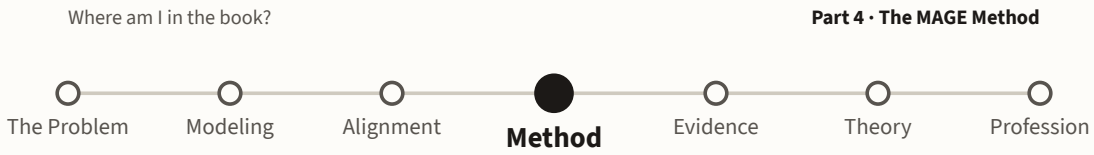
Figure 3.5.1-7: Modeling as a loop. *(a) Not the linear design → implement → document-with-models. (b) A problem drives a model; analysis yields a prediction and a design; those drive implementation and measurement; measurement refines the model, and the loop continues while the questions justify the cost.*

This changes what "model-based" can mean in an agentic engineering process. It need not mean specifying the system completely before implementation begins. Implementation can move quickly, expose a question, and supply measurements while the engineer builds just enough representation to reason about what matters next. The danger is that cheap implementation makes the other half of that loop easy to skip: try something, see whether it works, and try something else. That approach produced the first three steps of the memory story—more examples of success—but it could not produce the bound.

None of the engineering techniques in this interlude requires MAGE. An experienced engineer might have recognized some of them earlier, chosen different representations, or found a better architecture. MAGE offers something different: a disciplined way for the engineer and the agent to discover what the particular system needs. Observations become questions; questions motivate models; models support predictions and constraints; implementation supplies evidence; and discrepancies expose what remains poorly understood. What is learned can then become part of the engineering environment rather than remaining only in the engineer's head.

The objective is not to replace engineering judgment with models, or to make familiar software-engineering techniques sound new. It is to make good engineering reasoning systematic, controlled, and fast enough to keep pace with agentic implementation.

Part 4: The MAGE Method



QUESTION THIS PART ANSWERS

How do I practice MAGE?

THE MAGE CYCLE

Model the intent. Give stable obligations authority. Convert recurring judgment into durable engineering structure.

Repeat.

CARRYING FORWARD

Model · Modeling Principle · Alignment Principle · Governance Mechanism · Governance Conversion · Engineering capital · Governed Engineering Environment

NEW HERE

MAGE workflow · Brownfield migration · Generative validation

Parts II and III developed Modeling and Alignment separately. This Part runs them together as a method. Modeling supplied a repertoire of representations for making different engineering questions tractable; Alignment supplied the mechanisms through which selected obligations can acquire authority. In practice, engineers choose and connect these moves as the work demands. MAGE begins by asking what should be made explicit, places authority where stable obligations become legible and enforceable, and converts recurring judgment into durable engineering structure.

The result is a cycle rather than a catalogue. A task begins from the representations, controls, and evidence the environment already owns. Work exposes what those structures can and cannot answer. Some gaps earn better models; some earn sensors, validators, constraints, or gates; some remain judgment because the underlying obligation is still uncertain. Each useful conversion changes what later work inherits. Useful durable structure becomes engineering capital when later work benefits from it. That capital can drift, depreciate, and cost more to maintain than it returns, so this Part covers operation and retirement as well as accumulation.

DocAble supplies the deep case. Other organizations appear where they sharpen a move. The question here is practical: **what should you do next?**

4.1 The MAGE Workflow

MAGE has three recurring moves: **Model, Align, Convert**. Model the engineering knowledge or intent that should survive the current reasoning step. Give stable obligations authority at a boundary where they can actually be evaluated. When work repeatedly exposes missing knowledge, evidence, judgment, or authority, convert the lesson into durable structure. Then run the cycle again. Each pass asks where consequential judgment is still being purchased per realization: what judgment should be better supported, what should be given authority, and what should deliberately remain open.

The ordering is methodological. Local controls can act before a rich system model exists. MAGE asks about representation first because explicit models enlarge the engineering questions the environment can answer. Choose the representation that makes the question tractable, then attach authority where the obligation is stable enough and the evidence adequate enough to justify it.

THE MAGE CYCLE

Model the intent. Give stable obligations authority. Convert recurring judgment into durable engineering structure.

Repeat.

4.1.1 One Turn at a Time

Need. A turn begins with something you want to learn, change, preserve, or make true. Sometimes that need is already precise: an endpoint must require authentication; this transformation must preserve content; this state transition must never occur. Sometimes it is not: onboarding feels confusing, operators think the system is slow, customers may want a different workflow. Do not treat the second kind as a defective specification waiting for an agent to elaborate. It is an unresolved engineering question.

MAGE does not require certainty before work begins. Keep the uncertainty explicit. Preserve what is already known; explore what is not. A non-negotiable security boundary may deserve authority immediately while the product behavior inside that boundary remains experimental. As evidence accumulates, promote stable claims into requirements, models, and invariants and — where justified — give the resulting obligations authority through mechanisms that enforce them.

Be precise about what you know, experimental about what you do not, and make learning durable as it stabilizes.

Model. Start from the engineering question and choose a representation whose semantics make that question tractable. A structural model may expose dependency or reachability; a behavioral model may expose legal states and transitions; an ownership model may expose

control and transfer; a decision model may expose what is permitted; a measurement model may expose a quantity against a bound; and a provenance model may expose lineage and evidence. Use only the distinctions the question requires. The representation earns its cost when later work benefits enough to justify building and maintaining it.

Do not confuse explicitness with certainty. A model can represent the current behavior, the constraints already known, and the measurements needed for exploration while leaving the proposed behavior provisional. Modeling an unsettled hypothesis can help reason about it; giving that hypothesis authority merely because it is machine-readable is a different move. **Uncertain intent should not acquire premature authority.**

Explore or implement. If the problem is still being discovered, implementation can itself become an instrument of inquiry. Build cheap alternatives, measure them, put them in front of users, or construct the smallest prototype that can falsify the assumption. Commodity intelligence reduces the implementation cost of this exploration. That does not make the resulting code authoritative: prototype to learn, then preserve what stabilizes in the representations and controls that deserve to persist.

Once the task is sufficiently settled, let the fleet work against the representations, procedures, and controls appropriate to it. Delegability — the degree to which a settled task can be safely handed to the fleet — comes from the **whole engineered environment**.

Align and evaluate. Decide which obligations deserve environmental authority and place the enforcing mechanism at the earliest boundary where the obligation is legible and enforceable. Ask what role the environment needs to play: constrain an inadmissible action, sense relevant state, validate evidence against the obligation, or gate whether work may advance. One mechanism may play several roles. Some obligations are local enough to decide at an action boundary; others require a whole work unit, a system-level model, runtime evidence, or human judgment. The mechanism must have the semantics and evidence needed to decide whether the obligation is satisfied.

Prefer evidence independent of the producing agent. Such evidence can be used to challenge and test the producing agent's claim: re-run the check, derive the state, measure the property, search for counterexamples, or send the result to an independent reviewer.

Convert. Work will expose gaps. A repeated failure may reveal missing representation, a missing obligation, inadequate observability, a weak validator, or authority placed at the wrong boundary. A repeated manual judgment may reveal a procedure worth preserving. Diagnose the class rather than reflexively adding another gate.

When the future return justifies the upkeep, make the lesson durable: improve the model, state the invariant, add the missing evidence, mechanize repeatable judgment, remove an unsafe action, or package a recurring procedure.

Repeat. Models drift, controls become noisy, assumptions expire, and mechanisms can outlive the failures they prevent. Maintenance, reconciliation, and retirement belong in the cycle. The objective is a productive governed environment.

Figure 4.1-1 shows how precision increases as knowledge stabilizes.

Figure 4.1-1: Precision Follows Knowledge. *Explore uncertain needs; represent claims as they stabilize; grant authority only when an obligation is stable and consequential enough to justify it.*

4.1.2 Closing the Loop

When judgment repeats, ask what future work should inherit from it. The answer may be a model, invariant, sensor, validator, constraint, gate, runbook, or skill. Some recurrences are too cheap or unstable to encode.

Price the conversion against the recurring cost it retires. A repeated mechanical review may earn a validator; repeated reconstruction may earn a model; a catastrophic failure may earn prevention before recurrence.

The improvement loop has familiar ancestors: mistake-proofing[51], *jidoka*, continuous improvement, resilience engineering[52], and the postmortem discipline of converting incidents into durable changes[53]. Implementation abundance changes the economics: autonomous work can expose gaps faster than human attention can repeatedly absorb them, increasing the return on durable engineering structure.

4.1.3 Guidance or Authority?

Keep one distinction sharp. **Guidance influences behavior; an authoritative obligation is one the environment can enforce.** A brief, convention, example, or skill can strongly influence an agent, but the agent can still misunderstand or ignore it. A type or permission can exclude an inadmissible action; a validator-backed gate can reject a result. These mechanisms do not depend on the producing agent remembering the obligation or voluntarily preserving it. A useful diagnostic for any step is therefore to ask: *what consequential judgment does this step still trust the agent to make?*

Do not make authority the automatic destination of every piece of guidance. Authority is appropriate when the obligation is sufficiently stable, the environment can evaluate it at the right semantic boundary, and the cost of violation justifies enforcement. Guidance remains appropriate where judgment is unsettled or where the cost of mechanization would exceed the failure it prevents.

“Must never happen” identifies a candidate obligation. Find the property and the boundary where it can be decided, then choose a mechanism capable of enforcing it.

4.1.4 Engineer the Judgment

Not every important judgment should become deterministic. MAGE provides two complementary ways to improve recurring work. One is to improve the conditions under which a reasoner makes a judgment: give it better representations, domain knowledge, examples, procedures, tools, and context so that a satisfactory realization becomes more likely. The other is to give selected obligations authority in the environment so that satisfying them does not depend entirely on the reasoner choosing correctly.

The distinction is not between two kinds of engineering artifact, but between two effects those artifacts can provide. A model can guide an agent and state properties consumed by a validator. A skill can improve judgment and invoke controls that enforce authoritative obligations. Evidence can inform a decision or feed an admission gate. A governed engineering environment composes these roles rather than assigning each artifact permanently to one side.

The choice is not simply between probabilistic judgment and deterministic enforcement. Start with the obligation and ask what representation makes it tractable, then use the cheapest judge adequate to the representation and the decision at hand. A type checker or predicate should decide a property it can decide completely; a test or model checker may be appropriate where the property concerns behavior; a model or human may be necessary where the remaining question is semantic. There is no virtue in asking a frontier model to decide what a predicate can establish, and no virtue in forcing a semantic judgment through a cheap deterministic proxy that cannot represent what matters.

Representation can change that allocation. A judgment that initially requires expensive interpretation may become mechanically decidable once the relevant states, relations, or invariants are explicit. Conversely, a cheap judge can be perfectly reliable over an inadequate representation and still answer the wrong engineering question. In DocAble, pixel variance was easy to measure but could not decide whether a low-variance image carried semantic content; the representation had discarded the distinction the obligation required. The engineering move was not to tune the threshold harder, but to route the unresolved semantic question to a richer judge.

Use the weakest representation that preserves the distinction the obligation requires, and the cheapest judge adequate to decide it. Then decide whether the resulting judgment warrants enforcement. A deterministic check may remain advisory; a probabilistic evaluator may sometimes be reliable enough to support enforcement. Adequacy of judgment and whether the resulting judgment should be enforced are separate engineering decisions.

There is an older way to understand this composition. Work on distributed cognition studies reasoning performed by systems of people, representations, procedures, and artifacts rather than locating all cognition inside one individual.[54] External representations matter in that account because they do cognitive work: they preserve state, change what operations are easy, and coordinate reasoning across participants. A Governed Engineering Environment has a similar systems character, although it adds an engineering concern that

distributed cognition does not by itself supply: selected obligations can be enforced by mechanisms outside the immediate reasoner.

Inset — The GEE as a Distributed Cognitive System COGNITIVE SCIENCE

Studies of distributed cognition ask us to move the boundary of analysis outward. Edwin Hutchins's classic study of ship navigation, for example, does not explain navigation solely by asking what one navigator knows. Position is established through a system of people with different roles, instruments, charts, procedures, communications, and intermediate representations. The relevant cognitive system is larger than any individual participating in it.[54]

The same perspective is useful for agentic engineering. A coding agent need not carry the whole engineering problem inside its context window. Models preserve selected facts about the system; skills preserve procedures; tools perform operations that need not be approximated through language-model reasoning; evidence records what occurred; people or agents supply judgments that remain difficult to evaluate mechanically. The capability of the resulting engineering system is therefore not simply the capability of its foundation model.

External representations matter for more than storage. Work on distributed cognitive tasks shows that the form of an external representation can change the operations required to solve a problem.[55] A state machine exposes legal transitions; a dependency graph exposes reachability and coupling; a measurement model exposes a bound. The representation changes the reasoning problem presented to whoever—or whatever—reasons next.

A GEE adds another step. Its surrounding machinery can sometimes act on the represented property. A dependency model may help an agent understand the architecture, but a validator can also compare an observed dependency with an architectural obligation, and a gate can refuse the change. The environment therefore does not merely distribute cognition across reasoners and artifacts. For selected obligations, it can enforce what those representations express.

This perspective helps explain why replacing one model with a more capable one does not make the surrounding engineering environment irrelevant. The environment carries system-specific knowledge, procedures, evidence, and controls that no general-purpose reasoner should have to reconstruct for every task. Better reasoners change what that environment needs to supply. They do not eliminate the value of putting durable engineering knowledge in the system around them.

Nor is the boundary fixed. A judgment that initially resists authority may become better understood through repeated use, failure, and refinement. Its obligations may become clearer, suitable evidence may become available, or a qualitative distinction may become evaluable with sufficient reliability. At that point, some of what previously depended on judgment can acquire authority in the environment. Other portions may properly remain probabilistic. Figure 4.1-2 shows the recurring cycle.

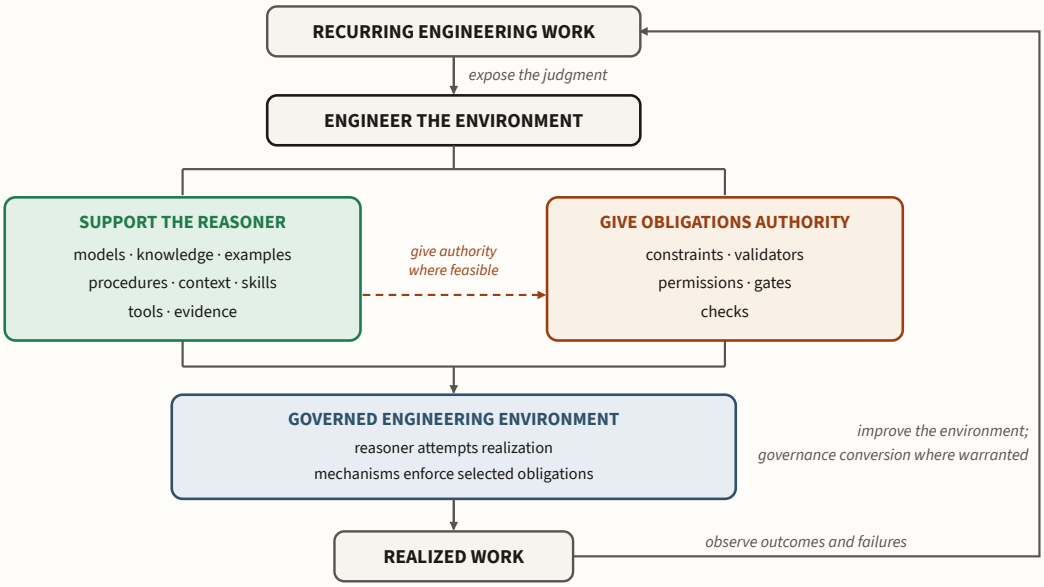


Figure 4.1-2: Engineering recurring work. *A governed engineering environment combines support for probabilistic reasoning with authority over selected obligations; the same models, knowledge, procedures, skills, tools, and evidence may contribute to both. The dashed arrow marks a recurring MAGE move: as judgment becomes better understood, some obligations may become sufficiently clear and evaluable to receive authority. The return path observes outcomes and failures and improves the environment; where recurring judgment can be encoded economically, this is the governance conversion developed in Section 3.4.*

The return path includes the governance conversion developed in Section 3.4. When an outcome or failure exposes missing representation, obligation, evidence, evaluation, or consequence, the organization can encode that missing structure so that future work inherits it. Not every observation warrants conversion: some judgments remain contextual, some failures are not recurring, and some structure costs more than it returns.

The idea that an organization should learn by changing the structures that govern later action also predates MAGE. Organizational-learning research distinguishes correcting an observed error from changing the governing assumptions, rules, or norms that repeatedly produce action.[56] Governance conversion makes a narrower engineering move. When experience exposes a recurring missing obligation, representation, procedure, or control, the lesson can become a model, skill, validator, permission, measurement, or other durable structure that later work inherits. The organization has then changed not only the failed instance but the environment in which the next instance will be produced.

A related lesson comes from the study of human error in safety-critical systems. Reliable systems do not assume that the person performing an operation will never make a mistake. They use multiple defenses so that an error need not become a system failure.[57] The same principle applies when the immediate reasoner is an agent. Better instructions and better context can reduce error, but recurring failures should also prompt a systems question: what representation, interface, check, independent evidence, or admission control could keep the same mistake from propagating? A GEE treats the reasoner as one component of the engineering system rather than its sole source of correctness.

The dashed arrow is an opportunity, not a required progression. MAGE does not assume that engineering knowledge should eventually become deterministic, or even authoritative. Some judgments remain qualitative because the underlying objective is contextual, plural, or difficult to observe independently. Others contain narrower obligations that can receive authority even while the larger judgment remains open. The relevant unit is therefore the obligation, not the task.

Example — good writing. Consider the obligation to produce "good writing." A style guide can externalize knowledge about that objective and improve an agent's realization, but there is no single construct called good writing that admits a sound checker. Some narrower obligations nevertheless admit deterministic checks: spelling, grammatical rules, paragraph-length bounds, prohibited terms, or excessive repetition. Others admit useful but probabilistic evaluation: whether a paragraph has an effective topic sentence, whether an introduction summarizes and prepares the material that follows, or whether prose exhibits a desired style such as terse and factual rather than metaphorical. Those evaluators can themselves receive authority where their reliability and consequences make that feasible.

None establishes that the resulting prose is "good writing." They establish narrower properties that have become sufficiently explicit to evaluate, while the broader judgment remains partly contextual and plural. Repeated experience can move this boundary. A preference first recorded only as prose in a style guide may later become a precise rule; a recurring qualitative criticism may become a probabilistic evaluator with useful measured behavior; some judgments may never justify either. The style guide remains useful throughout because its role is broader than the subset of its knowledge that the environment can enforce.

Example — from DocAble evidence to a memory bound. The DocAble memory example from the Interlude shows the same movement in a quantitative setting. The initial engineering knowledge was empirical: simple presentations worked, real teaching presentations worked, and eventually hundreds of documents from several universities worked. Those observations supported engineering judgment about the architecture's operating range, but they could not establish what would happen for a larger media payload. A roughly 250 MB presentation exposed the boundary of that evidence.

Modeling the artifact made a stronger obligation feasible. Separating the document's structural skeleton from its media exposed the relation that mattered: persistent worker memory

need not scale with total media size. The resulting residency model states a quantitative invariant,

$$\text{peak worker RSS} \leq C$$

where, within the declared operating envelope,

$$C = \text{rolling stream buffer} + \text{structural skeleton} + \text{handles} + \text{remediation working set}$$

and C does not grow with total media size. The architecture can enforce the assumptions behind that bound, while measurement checks whether the realized system behaves as the model predicts. What began as judgment supported by examples became a narrower property for which the system could provide substantially stronger evidence and authority.

The stronger mechanism still does not establish that DocAble "uses memory well," any more than spelling checks establish good writing. It governs a property that became precise enough to state and enforce: within the modeled envelope, total media size does not determine peak worker memory. Other dimensions — pathological structure, an individual object beyond the supported bound, excessive concurrency, or something not yet modeled — remain outside that claim. The progression from examples to a bound therefore illustrates the dashed arrow: modeling can make previously judgment-dependent obligations feasible to give authority.

Adequacy of judgment and enforcement are separate engineering decisions. A deterministic checker can remain advisory, while a probabilistic evaluator can support enforcement if its measured reliability and the cost of error make that appropriate. Deterministic evaluation provides stronger assurance for a covered decidable property; authority asks whether the obligation should govern the work, and enforcement determines whether the environment actually requires it.

4.1.5 Prefer Structural Prevention

Prefer structural prevention when the legitimate action space can be closed. A typed API, a sanctioned mutation seam, a closed verb set, or a forbidden dependency can make an invalid move simply unavailable, and the reviewer need not catch that class of mistake because the governed interface does not offer it. Where prevention would exclude legitimate behavior, or the property only becomes visible after the work runs, preserve the action space and add independent evidence instead.

Where an invalid state can be made unrepresentable without excluding legitimate behavior, prefer a constraint. A property visible only after execution needs evidence and evaluation.

DocAble put both sides to work at once. It narrowed document repair to a closed set of bounded, typed edits, routed ordinary document-format mutation through sanctioned structured seams, and rejected direct raw-library access from code outside those seams. It turned recurring review findings into build-failing checks — prevention wherever the action space could be honestly closed. It then validated content preservation as a post-condition

on every run, because "the output still says no less than the input" is a property that only becomes legible *after* the transformation.

4.1.6 Size Work to Reasoning and Assurance

Not every agent task deserves the full apparatus of governed autonomy. A bounded transformation whose relevant state fits comfortably in one pass, whose output is cheap to inspect or mechanically verify, and whose failure is cheap to reverse can often be delegated directly. Write the contract, let the agent produce the result, check it, and move on.

Size the task along two axes: reasoning burden and assurance need. The first is **reasoning burden**: how much intermediate state the task must preserve before it can produce a coherent answer. Large systems, cross-cutting changes, and long sequences of dependent decisions push that burden upward. The second is **assurance need**: how costly, irreversible, or difficult to detect a bad result would be. A five-line authorization change may have a short reasoning horizon and still deserve strong validation; a large disposable prototype may tolerate far less.

Size work as a sequence of transformations with explicit inputs, outputs, and checkable boundaries. Make each step large enough to exploit the model's capabilities but small enough that its result can be meaningfully evaluated before the next step compounds it. Better representation can support larger steps because it reduces the state the reasoner must reconstruct; stronger models may support larger steps too. Neither removes the need for an evidence boundary where the consequence warrants one.

DocAble's remediation edit language is the concrete pattern. Rather than ask a model to rewrite an entire document opaquely, the system asks for bounded, typed edits — set this alt text, reorder these children, change this role. Each edit can be stamped, reversed, and evaluated. The **unit of delegation should line up with a unit of evidence**.

Task shape matters as much as task size. A perfectly bounded transformation aimed at the wrong representation still fails. If a model performs poorly on geometry, carving the geometric problem into smaller geometric calls does not rescue the framing. Change the representation or change the division of labor. MAGE sizes work to the reasoner, the available representation, and the assurance required — not to an arbitrary line count.

4.2 Migrating to MAGE

Most readers do not begin on a blank page. You inherit code, tests, schemas, architectural documents, CI rules, permissions, dashboards, conventions, and operational lore. Some of that inheritance is debt; some is useful engineering structure that has become fragmented, stale, or difficult to reuse. Migration is therefore partly an exercise in recovery: find the representations and controls the system already owns, reconnect them to the territory they describe, repair or retire what has drifted, and invest in new structure where recurring engineering costs justify it.

> **Inset — Software evolves; its engineering structure must evolve with it** SOFTWARE ENGINEERING
>
> Software engineering has long treated change as a normal property of successful systems rather than an exceptional maintenance phase. Lehman's work on software evolution observed that systems embedded in a changing real-world environment must continually adapt, while their complexity tends to increase unless engineers act to control it[58].
>
> MAGE extends that maintenance obligation beyond implementation. Models, validators, gates, runbooks, and other governance mechanisms describe or constrain a system that continues to change. They can therefore drift, preserve assumptions that no longer hold, or impose costs that no longer earn their keep. A governed engineering environment is not a finished artifact. Its engineering structure must evolve with the system it governs.

Two axes fix the starting move. The first is **how much system already exists**. A mature system gives you implementation, history, and latent models to recover; a new system does not. The second is **how settled the relevant intent is**. A detailed contract can justify substantial modeling and authority before implementation; an exploratory product question cannot. These axes are independent. Greenfield does not imply uncertainty, and brownfield does not imply settled intent: a blank-page regulated system can carry highly settled obligations, while a mature product may still be exploring what its users need.

Figure 4.2-1 crosses how much of the system already exists with how settled the relevant intent is. Each cell suggests the starting move.

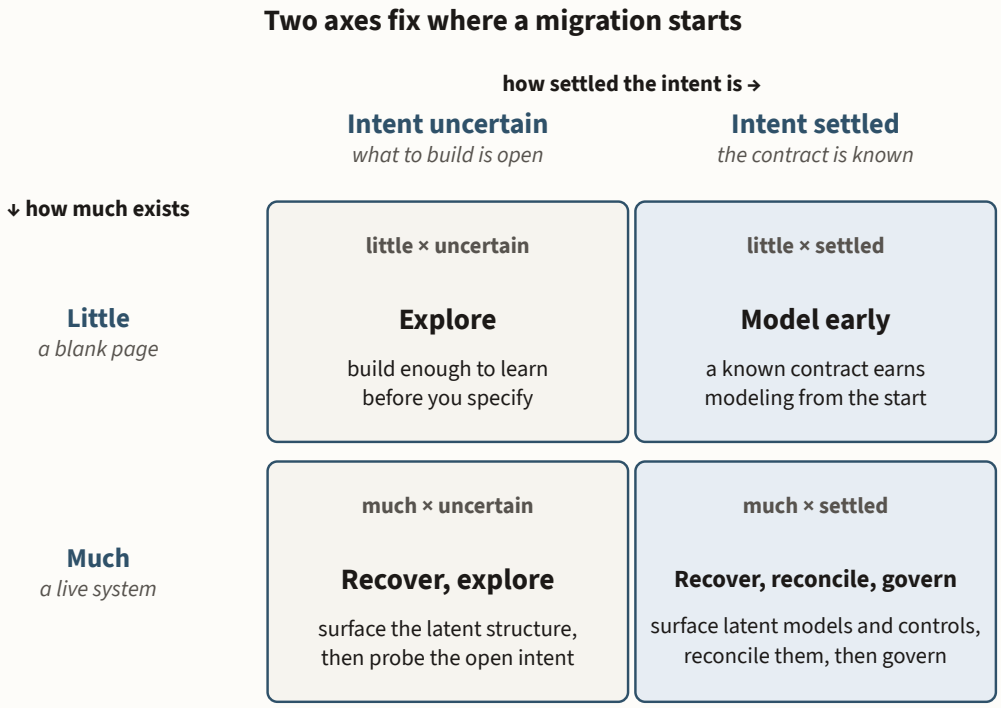


Figure 4.2-1: Where a migration starts. *Existing system stock determines what can be recovered; settled intent determines how much structure and authority the work can support initially. Authority is not a single project-wide dial — the more settled a* particular obligation*, the more authority it can safely receive, so a security, legal, or safety constraint may be settled even when the product behavior around it is still exploratory.*

Settledness does not determine whether an obligation requires judgment. “Make this explanation useful to its audience” can remain a stable objective without admitting a deterministic oracle. In that case, preserve and improve the representations, knowledge, examples, and procedures used to make the judgment rather than pretending to mechanize it. Conversely, exploratory work can contain settled sub-obligations that the environment can enforce. Decide authority obligation by obligation, and revisit that decision when better models or evidence make stronger controls feasible.

Brownfield work enters through many doors, but two are useful extremes. At one end is the fast-built system whose implementation outran its explicit engineering account: code exists, perhaps even a working product, but the architecture, obligations, and operating rules were never made durable. DocAble began near this end: a plausible product was built quickly, then migrated into a governed one. At the other end is the established organization with years of accumulated structure: ADRs, schemas, ownership files, developer portals, CI policy, tests, and operational playbooks. Its problem is not absence but fragmentation.

A fast-built system must author more of its initial models. An established system can recover and reconcile more of what it already knows. Both begin by inventorying the representations and controls already present, then investing where missing structure repeatedly consumes judgment.[18]

Neither starting condition requires a complete governed environment before useful work begins. Preserve the obligations already worth preserving, surface the representations already worth trusting, and let implementation and operation expose what is still missing. The method is constant; the justified starting stock is not. A safety-critical project may begin with extensive explicit requirements and heavy validation; a product search may begin with a few non-negotiable constraints and measurements; a brownfield system begins with the accumulated evidence of its own history, however poorly represented.

Model-first and implementation-first suit different starting conditions. Where consequential intent is already known and costly to reconstruct, representing it before implementation narrows the search toward useful candidates. Where the right shape of the system is still uncertain, implementation is itself the experiment: build cheaply, watch what the result teaches, and model what proves consequential.

So the practical question is not how much to model in the abstract. It is how much structure the next unit of work needs. A familiar problem with strong tests and a well-matched reasoner may need little added representation; an unfamiliar domain with weak observability may need substantial modeling before autonomous work can be trusted. **Add enough representation and evidence for the next unit of work to be delegated and evaluated responsibly; then let realization proceed.**

At the low-structure extreme, implementation-first agentic development is heuristic search driven largely by the model's learned prior. In a familiar domain a capable agent can infer adequate abstractions, architecture, and implementation from surprisingly little explicit structure. Repeated failure changes the calculation. When agents keep reconstructing the same concept, cross the same hidden boundary, or re-search the same unproductive region, another attempt is worth less than changing how the problem is represented. **Implementation-first becomes churn when the search repeatedly reconstructs structure whose explicit representation would cost less than continued rediscovery.**

4.2.1 Start with What You Already Have

Begin with two questions, because representation and authority are different things. First, what does the organization already reason through? Which documents, schemas, graphs, tests, or configurations repeatedly save someone from reconstructing the system from raw implementation? Second, what already has authority? Which permissions, types, validators, CI checks, reviews, or operational gates currently constrain or admit work, and what can

[18] A similar interpretation appears in recent developer-portal practice: a developer portal's value was never its UI but the curated context underneath — service and API catalogs, golden-path templates, ownership, CI/CD signals — and agentic engineering turns that human-facing surface into an execution substrate, "the portal isn't dying; it's becoming infrastructure"[59].

each actually decide? A mature system may already contain substantial local Alignment even when its system-level representations are weak. A schema may reject invalid data, a type system may exclude states, and CI may block a merge. The migration problem is therefore not simply to build a better map and add controls afterward. Strengthen representation and authority together: connect useful representations to the territory they describe, make important obligations explicit enough to evaluate, and place authority where the required semantics are available.

A wiki is a useful first path because many organizations already have one and the migration distance is short. Treat it as an **informal model**: a reduced representation people already use to reason about the system, even though its semantics are weak and nothing mechanical guarantees correspondence with the code. Pages supply nodes of a kind, links supply relationships of a kind, and search supplies navigation. That does not make the wiki equivalent to the structured graph Part 2 built. It makes it raw material.

The first improvement is connection, not replacement. Important claims should point to the code, tests, configuration, or runtime evidence that realizes them; important implementation regions should point back to the pages that explain why they exist.[19] The resulting trace is not yet a proof that the prose is true. It is a disciplined join that lets audits and later mechanisms discover when the map and the territory disagree. Figure 4.2-2 draws the two independent ways that join can be strengthened.

Brownfield representations can be strengthened incrementally from informal descriptions to structured, machine-readable models and, where a consumer justifies the cost, executable models.

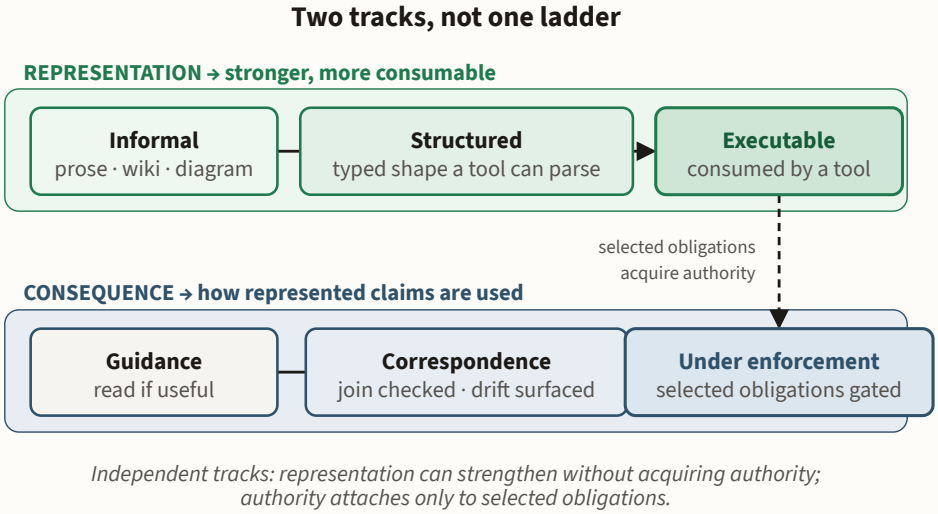


Figure 4.2-2: Representation and authority strengthen independently. *A representation may become structured or executable without becoming a gate; selected obligations acquire authority only when enforcement is warranted.*

[19] The earlier Parts develop the stronger form of this join: every model node can point to the implementation it represents, and that implementation can point back to the model node that explains it.

4.2.2 Finding Models in the Implementation

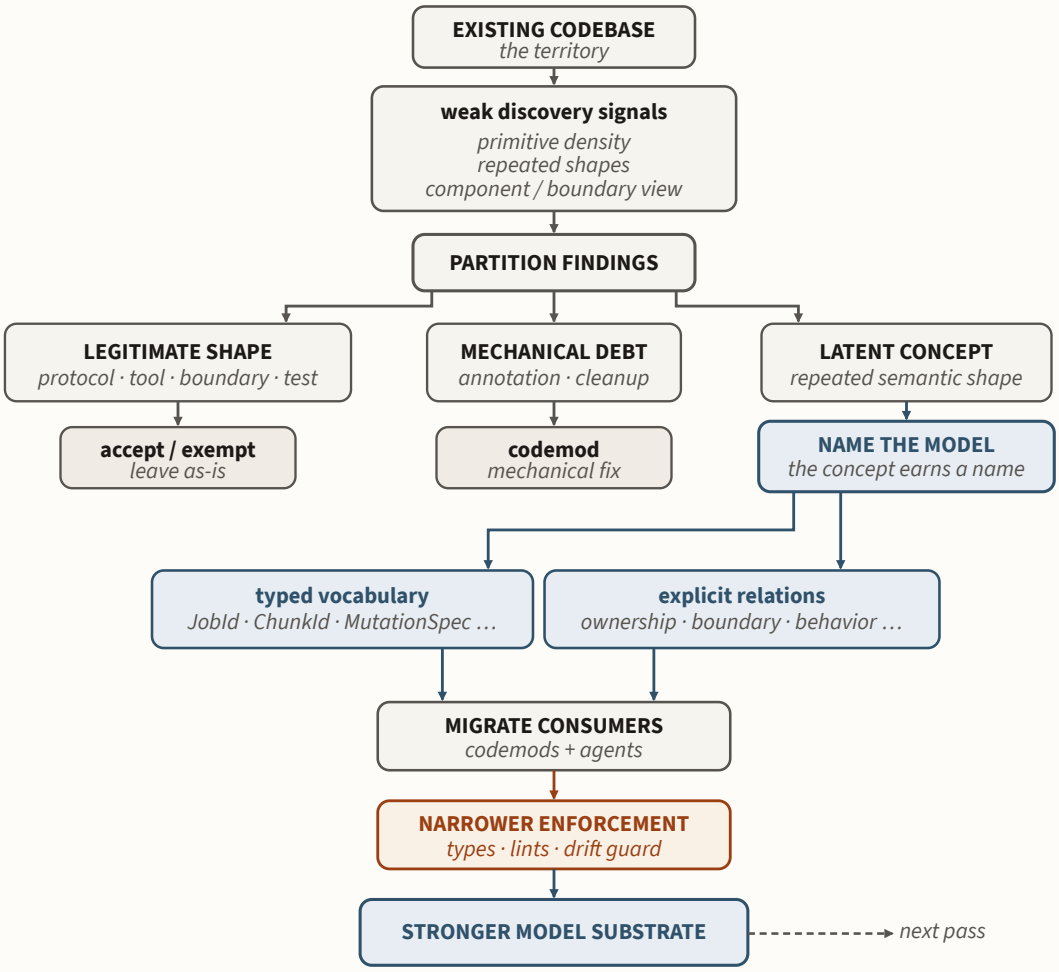
Brownfield models need not begin with an engineer inventing the right abstraction in advance. Existing implementation often contains repeated structures that point toward models the system has never named. Static analysis, search, clustering, and other detectors can surface those patterns at scale, but their findings are evidence rather than conclusions. Repetition may reveal a missing domain concept; it may instead reflect appropriate low-level implementation, a protocol boundary, test scaffolding, or ordinary local cleanup. The detector finds places worth examining. Engineering judgment determines whether the repeated structure carries a shared meaning worth representing.

DocAble encountered this problem through a primitive-density detector, which identifies interfaces expressed heavily through bare strings, integers, tuples, and other low-level values. Primitive density is intentionally a weak signal: such interfaces sometimes indicate that a concept is being repeatedly reconstructed rather than represented explicitly, but they are also common in perfectly legitimate code. The detector therefore produced hundreds of findings without claiming that hundreds of models were missing. Most findings needed ordinary annotations or local cleanup, or occurred in test fixtures and protocol adapters where primitive-heavy interfaces were appropriate.

A smaller subset behaved differently. Across related document mutators, the same parameter shapes recurred: language mutations repeatedly carried the same language and provenance concepts; alt-text mutations repeatedly carried description and attribution state. Looking across those instances revealed something that no individual finding established on its own. The repeated primitives were not merely similar code. They were recurring expressions of domain concepts that the implementation had never named.

Repetition alone is not enough to make that inference. The finding must be interpreted in its architectural context. A primitive-heavy external protocol adapter is different from primitive-heavy internal domain code: the former may be faithfully representing an external interface, while the latter may be forcing every consumer to reconstruct a concept the system already understands. The same distinction applies more generally. A recurring structure may be legitimate shape, mechanical debt, or evidence of a latent concept. The useful question is therefore not simply *does this pattern repeat?* but what explains the repetition, and would naming the shared meaning change future reasoning?

This gives a practical route from weak implementation signals to explicit models. Surface repeated structure; partition findings by architectural role and likely cause; compare related instances for shared meaning; name a concept only when that meaning earns a representation; then migrate consumers and enforce whatever narrower relationships the new model makes decidable. Figure 4.2-3 summarizes that process. The schematic is intentionally simpler than the judgment that produces it: detection narrows the search space, while engineering interpretation decides whether there is a model to recover.



Weak signals locate the work; they do not dictate the remedy.
Only repeated structure that earns a name becomes a model — the rest is accepted or mechanically cleaned.

Figure 4.2-3: Discovering models during a brownfield migration. *Weak signals — primitive density, repeated shapes, the component and boundary view — identify places worth inspecting; they do not dictate the remedy. Partitioning separates legitimate primitive-heavy code and mechanical debt from repeated shapes that reveal missing engineering concepts. Naming those concepts as typed vocabulary and explicit relations changes the substrate on which later migration and enforcement operate.*

Field record — Primitive-density migration

The primitive-density backlog fell from 698 file-level findings on June 8 to 472 on June 9 and 299 on June 10. The declining count is not the important result. What matters is that the remaining population was repeatedly reclassified as it shrank:

- **June 8 — 698 findings**, partitioned into cluster-uniform, mechanical, per-site-judgment, boundary, and test-fixture-or-exempt buckets.
- **June 9 — 472 findings.**
- **June 10 — 299 findings**, dispositioned as annotate, exempt, extract-a-model, redesign, retire, or defer-for-judgment.

Some findings disappeared through mechanical work, some exposed missing shared types, some warranted exemptions or lint changes, and some required design judgment. The chronology records those observations; it makes no claim that a single planned wave caused each step of the decline.

Inset — Rotate the problem PROBLEM SOLVING

When a problem is difficult, try looking at it from another angle. George Pólya made this a basic strategy of problem solving: draw a figure, introduce different notation, consider a related problem, work backward, or restate the problem in another form[60]. These moves do not change the underlying problem. They change the representation in which you are trying to solve it. In engineering, the same move can turn an awkward collection of conditions into a state machine, a tangle of calls into a graph, or a complicated sequence of events into a lifecycle.

Metaphor and analogy provide another way to rotate the problem. They let us understand an unfamiliar system using relationships we already understand elsewhere. Lakoff and Johnson argue in *Metaphors We Live By* that metaphor is not merely decorative language: it shapes what we notice and how we reason[61]. This is useful for modeling because a good analogy can suggest a useful representation. But every representation emphasizes some relationships and suppresses others. Calling a system a *pipeline* makes stages and flow easy to see, while directing attention away from other aspects of the same system.

The goal, then, is not to discover the one correct representation. Different representations can make different aspects of the same system easier to reason about. Rotate the problem and ask what becomes visible: a graph may expose reachability, a state machine may expose legal transitions, and a lifecycle model may expose temporal obligations. Keep the representations that make consequential engineering questions easier to answer, and use more than one when the questions demand it. This is why Part II develops several model classes rather than prescribing a single universal model.

4.2.3 Reconcile the Map, Then Promote Obligations

Begin with audit. Join the representation to the implementation and surface disagreement without blocking work. A legacy map is expected to drift; making every inherited claim authoritative before reconciling it would merely give old errors new authority. Repair disagreements through ordinary feature and maintenance work without assuming in advance that prose or code is correct: the representation may be stale, the implementation may be wrong, or the representation may be asking the question at the wrong grain. Promote selected obligations only after the representation is trustworthy enough for the consumer that will rely on it. Add structure when retrieval, analysis, generation, validation, measurement, or gating will use it; do not enrich a model merely because more structure can be represented. The companion From-Drifted-Wiki-to-Trusted-Model operator card holds the drill whole.

To make the shape concrete, suppose an inherited wiki describes the system but has drifted from the code, so agents and engineers repeatedly re-derive what is actually true from source. Audit the map against the implementation and reconcile disagreements through ordinary feature and maintenance work: sometimes the prose is stale; sometimes the code is wrong. Promote only claims trustworthy enough for the consumer that will rely on them. The result is a queryable model later work can use, while unresolved claims remain advisory rather than acquiring false authority.

4.2.4 Migrate One Bounded Surface at a Time

The immediate goal is not a complete model of the inherited system. It is a **trusted map** over the surface that matters now: important claims have known anchors, important implementation regions have explanatory homes, and agents can navigate between them without repeatedly reconstructing the relationship. Build only as much further representation and evidence as the **quality case** the system must defend requires — enough structural, behavioral, ownership, measurement, and traceability information to answer the properties you mean to stand behind. A live system rarely tolerates wholesale migration, so choose a bounded surface — a subsystem, service, policy family, or recurring workflow — and let old and new coexist while the replacement earns trust. Preserve reversibility, and generalize only what the bounded migration teaches you.

DocAble's serverless re-platforming ran this shape — a stateful job pipeline moved off long-lived polling workers onto instances that scale to zero behind a managed push queue; the push plane grew beside the poll plane behind a default-off toggle, proved green on a staging namespace running identical code, took the traffic, then the old cluster was deleted and preserved on an archive branch.

Within that slice, recover the representation using whichever direction the evidence supports. **Top-down** work starts from intended architecture or requirements and refines toward implementation. **Bottom-up** work induces stable abstractions from the code and rises toward a model. Removing accidental complexity — under tests — first can make

either direction easier. The directions should meet at the grain where the relevant property becomes answerable.

A disagreement then has the familiar three-way diagnosis — the representation is wrong, the grain is wrong, or the implementation is wrong. Do not decide which in advance; shifting a legacy codebase from "the tests pass" to "these properties hold across all behaviors" surfaces disagreements by design.

Inset — When the model and the code disagree SOFTWARE ENGINEERING

Tighten a model against real code and it will, sooner or later, contradict the code. The disagreement usually points to one of three questions:

- **Is the model wrong?** It may claim something the code never promised. Refine the model.
- **Is the model at the wrong grain?** It may be asking a question at a level of abstraction that cannot see the answer. Build a different model at the grain the property lives at.
- **Is the implementation wrong?** The model may be right, the code disagrees, and the disagreement may expose an implementation defect.

The discipline is to decide *which* before you act. Fixing code when the model was wrong bakes in a false claim; refining the model when the code had a bug hides the bug behind an adjusted spec.

For consequential substrate changes, represent important dependencies strongly enough to query the migration blast radius before cutover; the control↔substrate dependency model gives the operator pattern.

4.2.5 Audit → Drain → Promote

A valid future invariant is not always a valid present admission rule. A legacy system may violate a worthwhile new rule everywhere on the day you introduce it. Making the rule blocking immediately does not improve the system; it simply stops existing work. Introduce the obligation in three beats: Audit, Drain, Promote.

THE MOVE — AUDIT → DRAIN → PROMOTE

- **Audit.** Install the proposed rule as evidence only. It reports every detected violation and blocks no commit, so it never breaks an in-flight agent.
- **Drain.** A fix wave repairs the existing findings under sufficient tests and other evidence to preserve intended behavior, converting the legacy code to the new convention behavior-for-behavior.

- **Promote.** Once the governed surface satisfies the obligation, enforce it at the boundary so detected violations cannot re-enter.

Cloudflare's public account describes a similar **approved → enforced** sequence, landing each policy control advisory and flipping it to enforced once the tree is clean, so the promotion never breaks work in flight[62]. Figure 4.2-4 draws the three beats as one lint is worn into a legacy tree.

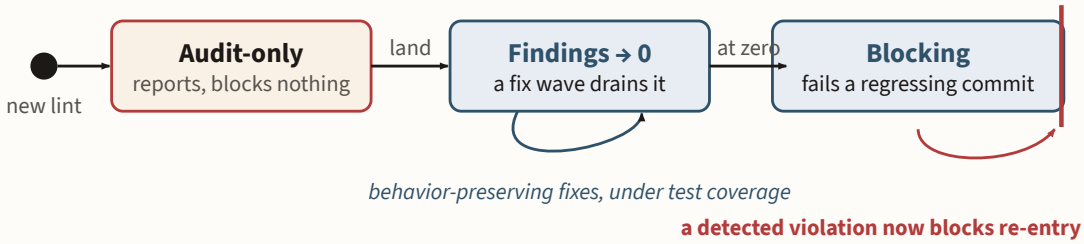


Figure 4.2-4: Audit, Drain, Promote. *A new lint lands audit-only — every finding reported, no commit blocked, so it never breaks an in-flight agent. A fix wave drains it to zero, and only then is it promoted to blocking, so future violations within the mechanism's declared detection surface are refused at that boundary.*

The final step requires care. DocAble did not promote primitive density itself into a universal blocking rule. The detector was a smell: useful for locating work, but too broad to define correctness. Once the findings had been understood, narrower mechanisms could prevent the specific regressions that mattered. Sometimes you promote the obligation, not the instrument that discovered it. An audit may reveal an obligation that deserves authority without being the right mechanism to enforce it; the detector can remain advisory while a narrower type, constraint, validator, or gate enforces the obligation.

4.2.6 Derive, Don't Copy

Recovering a model is only the first step. The next question is what happens to the facts it owns. A common failure is to declare one representation the source of truth while continuing to copy its facts elsewhere: ownership appears in the model and again in configuration; an architectural edge appears in the graph and again in an allowlist; a requirement relation appears in the model and again as a string in a test. Each copy creates another place that can drift. If a fact already has a queryable source of truth, derive it rather than copying it. Prefer a join over parallel lists, a stable identifier over a copied path, and generation over synchronized hand-editing when one representation can honestly own the result. Sometimes

neither side should own the other; in that case, make their correspondence observable and check agreement rather than silently declaring either the model or the implementation authoritative.

Figure 4.2-5 draws the shape: once a fact has a source of truth, downstream consumers query, join, derive, or generate from it, and where neither fully determines the other, a correspondence check makes disagreement visible.

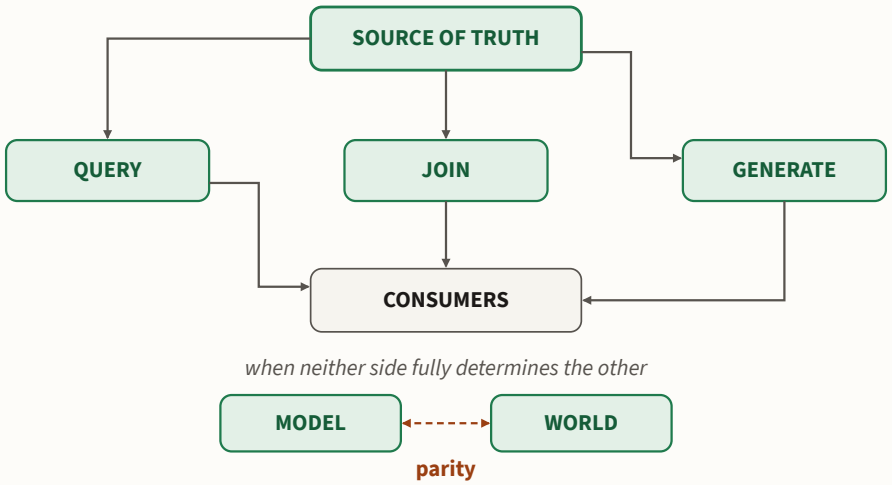


Figure 4.2-5: Derive, don't copy. *Once a fact has a source of truth, downstream consumers should query, derive, join, or generate from it rather than maintain parallel snapshots. Where neither representation fully determines the other, explicit correspondence checks make disagreement visible.*

Hidden assumptions deserve the same treatment as copied facts. A control that depends on a particular runtime, directory layout, deployment topology, tool version, or model capability carries that dependency whether or not anyone wrote it down. When changing the substrate could invalidate the control, represent the dependency strongly enough that its blast radius can be *queried* before the change — the blast-radius move above is exactly this, worked through on a real re-platforming. Do not model every dependency. Model the ones whose rediscovery by breakage is expensive. **Make consequential assumptions queryable before they become surprises.**

4.2.7 Price the Investment

A recurring gap does not automatically deserve machinery. Price the expected future loss against the cost of building *and carrying* the response. Frequency and consequence are useful first approximations; detectability, reversibility, legal exposure, and the stability of the obligation can move the decision substantially. Cheap, rare failures often remain judgment. Cheap, frequent failures may justify a lint, script, or reusable procedure, because repeated agent and human attention is itself cost. Consequential failures justify a **larger investment and a stronger assurance response**: better representation, prevention by construction

where possible, stronger sensing and validation, and authority at an appropriate boundary. Catastrophic or legally intolerable failures need not recur even once to earn that investment.

Table 4.2-1 crosses how much a failure costs against how often it happens; it gives a first-pass heuristic, not a decision rule. Do not build the larger mechanism until the expected cost of the failure justifies it.

Table 4.2-1: The Sizing Matrix. *Failure cost runs against failure frequency, and each cell suggests a proportionate first move — a heuristic, not a decision rule. Cheap, rare failures stay with judgment, while cheap, frequent ones earn a lint, script, or reusable procedure. Costly failures earn a recovery path with admission control, and prefer structural prevention once the failure's shape is decidable. The costly, frequent corner earns both prevention and a gate.*[20]

Cost × frequency	Rare failure	Frequent failure
Cheap	Judgment.	A lint, script, or reusable procedure.
Costly	Recovery with admission control.	Structural prevention with admission control.

Price the **upkeep** as well as the construction. Models drift, validators require calibration, and skills age. Engineering capital depreciates: a control that once retired meaningful cost becomes overhead when the obligation disappears or the upkeep exceeds the return. **Build the smallest asset that retires the recurring cost; retire it when the return disappears.** Fleet-scale migration amplifies the same economics: Spotify reports a migration workflow in which work that previously involved hundreds of teams over weeks could be scoped by one engineer over days, with tooling targeting and scheduling concurrent agent work[63].

WORKED EXAMPLES

DocAble. A fast-built prototype was matured in place as domain models, typed seams, and controls were introduced incrementally — a roughly 300,000-line codebase reduced to roughly 200,000 production lines, lint by lint and model by model. Part 5 (The Evidence) reconstructs the migration.

Takeaway. Start from the system you own. Recover useful structure, reconcile it to reality, and invest where stronger representation or authority repays its cost.

[20] Escalate one tier when the failure is hard to detect, hard to reverse, or legally or safety intolerable.

4.3 Validating Change

Every engineering process eventually asks the same question: **what evidence is sufficient to accept this change?** Code review, testing, static analysis, operational measurement, and expert judgment answered that question long before coding agents arrived. Commodity implementation does not make those techniques obsolete. It makes them more important, because changes can now arrive faster than human inspection can scale.

Validation begins with an obligation. If the obligation is settled, evidence can test whether the change satisfies it. If the obligation is still a hypothesis — whether users want the feature, which workflow they prefer, which tradeoff is acceptable — no test suite can manufacture the missing product knowledge. In that regime the evidence is experimental. MAGE can govern the experiment and preserve known constraints; it cannot turn an open question into a mechanical truth.

4.3.1 Independent Evidence

To validate a change is to assemble evidence that bears on its obligations and decide what consequence that evidence should have. The producer's own report is rarely enough. A human says the change is correct; an agent says the tests pass; a tool announces completion. Each claim may be useful, but the strongest assurance comes from evidence generated independently of the judgment seeking admission.

For consequential changes, prefer evidence that can challenge the producer's own claim rather than relying on self-report alone. Where the consequence warrants it, place admission authority outside the producer's ability to redefine success.

That evidence can come from several sources. Part II showed that purposeful models can make system properties explicit and make analyses over those properties tractable. DocAble uses many of its consequential models this way: structural models expose path and dependency properties; behavioral and ownership models support transition, reachability, and invariant checks; measurement models make quantities comparable against declared bounds. Other models primarily support navigation, traceability, context, or reconstruction and need not carry an invariant. A model therefore need not be an oracle to be useful, but where it exposes a consequential property, an appropriate analysis can produce evidence about whether that property holds.

Evidence can also come from outside a model. A test may exercise a behavior. A compiler or type system may reject an illegal state. A reference implementation may supply a differential oracle. A metric may reveal degradation. A human reviewer may decide a property that remains judgment-laden. MAGE does not require that all validation route through one kind of representation. It requires that the obligation and the evidence be adequate to the claim.

4.3.2 Build the Oracle

An **oracle** is whatever supplies the judgment against which the observed result is evaluated. An oracle is stronger when its judgment is sufficiently independent of the implementation being judged and explicit enough that its verdict has a stable meaning. Sometimes that oracle is a simple property: the output is an ordered permutation of the input. Sometimes it is a type or schema. Sometimes it is a reference implementation. Sometimes it is a human rubric.

> **Inset — The oracle problem** SOFTWARE ENGINEERING
>
> Software testing has a longstanding asymmetry: executing a program can be much easier than deciding whether the result is correct. Testing research calls this the oracle problem[64]. An expected output supplies an easy oracle for some cases; complex behavior may instead require properties, reference implementations, metamorphic relations, models, runtime evidence, or human judgment.
>
> Commodity implementation widens the asymmetry. Agents can cheaply produce implementations, variants, and test inputs, but producing more candidates does not supply an independent basis for judging them. As generation becomes cheaper, the engineering bottleneck moves toward stating what must hold and constructing evidence capable of distinguishing acceptable realizations from unacceptable ones.

Existing behavior can stand in for a surprising amount of written specification. In a compatibility port or a structure-preserving migration, a reference implementation supplies an executable oracle: challenge a new realization with the same inputs and compare it against known behavior. Tests, benchmarks, compatibility suites, and production traces close the target further. The implementation problem may stay enormous, but much less of the engineering question remains open.

Part II established the broader relationship between representation and analysis: a purposeful model makes some properties directly expressible and thereby makes particular analyses tractable. In validation, that relationship can supply an oracle for claims that do not exist at the level of one input/output example. A state machine can define legal transitions; an architectural graph can define permitted edges; a performance model can define an acceptable bound. State exploration, graph analysis, comparison against a bound, or another model-appropriate analysis can then produce evidence about the represented property. The same representation that helps an engineer or agent reason about the system can therefore become input to independent validation. **Modeling enlarges the semantic reach of the oracle, but an oracle need not come from a model.**

This is where generative validation becomes powerful. An example test pins one point: given *this* input, expect *that* output. A property states a law over a domain and lets a generator search for a counterexample. A stateful model can extend the same idea over sequences; a bounded concurrency model can extend it over interleavings. The useful progression is not "simple test to sophisticated test." It is **from one known case to a claim over a space.** Figure 4.3-1 draws the difference.

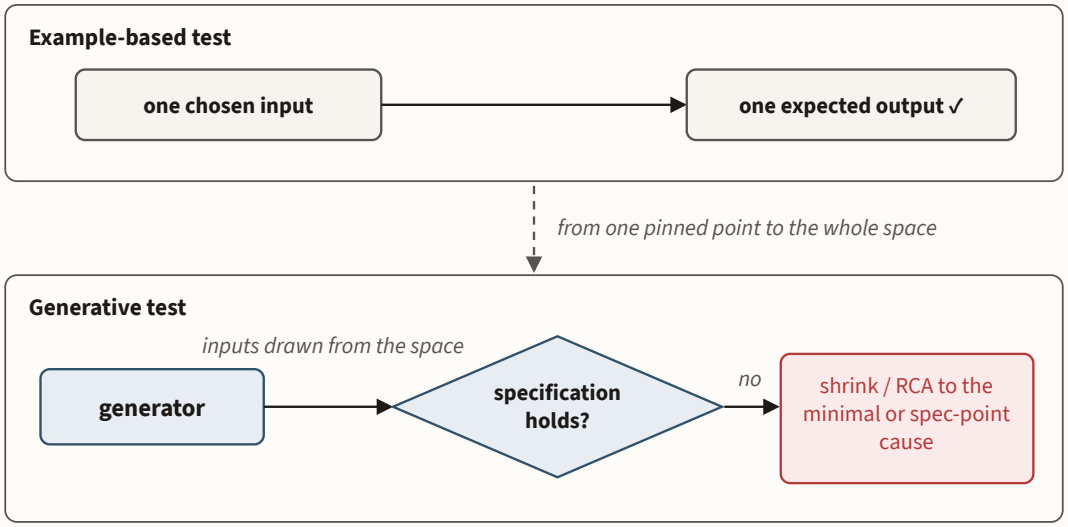


Figure 4.3-1: A Point and a Space. *An example pins one chosen case; a generative test states a property over a domain and searches for a counterexample. Use the broader claim when the obligation itself is broad enough to support it.*

A failing example invites a local repair, because the evidence names one point: patch the branch that makes *that* input pass and the example goes green. A broader property makes that repair easier to falsify if the underlying class remains open: the generator keeps drawing neighbours that still exercise the law, so a branch-cut that satisfied one input fails the next. *When the defect exposes a general law, preserve the law rather than only the example that revealed it.*

The techniques that generate evidence this way share one loop — **generate, execute, judge, shrink or diagnose** — and differ in the generator that makes the inputs and the oracle that judges them. **None is a maturity rung.** Choose the generator and oracle that match the property you are trying to falsify (Inset 1 lays them side by side).

Inset — Example, property, fuzzing, model-based: what changes? SOFTWARE ENGINEERING

Four ways to produce evidence, one shared loop — **generate → execute → judge**. They differ in where the inputs come from, where the verdict comes from, and what each is best at.

Technique	Inputs come from	Oracle comes from	Best at
Example test	engineer-chosen examples	the expected output, written down	known cases, regressions
Property test	an authored input domain	an invariant the output must obey	laws over many valid cases
Fuzzing	mutation and adversarial generation	a robustness contract, plus richer auxiliary oracles	malformed and edge inputs
Model-based / stateful	model-generated actions and states	an explicit behavioral model	sequences, protocols, state

Read each row left to right: the oracle defines what counts as success; the generator determines how the technique searches for evidence. The boundary is not even sharp: some systems combine an explicit input-language grammar with semantic constraints, generating high-diversity inputs while keeping tight control over what each one *means*[65] — a hybrid that sits between the property and fuzzing rows.

These are not rungs on a maturity ladder. Choose the technique whose generator and oracle match the engineering claim.

4.3.3 Generate Falsifying Evidence

Let the engineering claim choose the search. Not every obligation wants generated test inputs, and not every consequential property wants formal verification. Use the cheapest mechanism that can falsify the claim. Figure 4.3-2 sorts the claim kinds. As with modeling, changing the question may change the representation and the machinery that make the answer tractable.

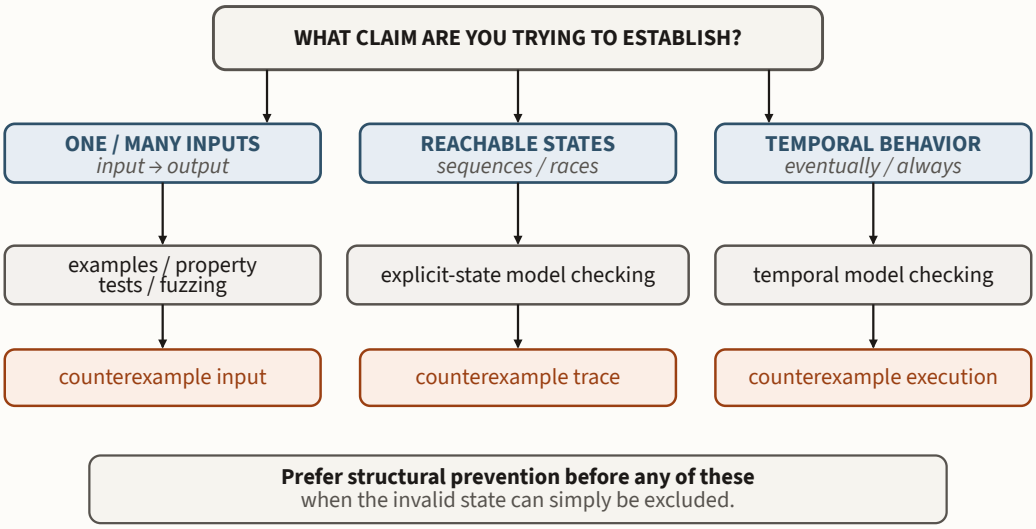


Figure 4.3-2: Let the claim choose the search. *Input/output properties, reachable-state invariants, and temporal properties require different kinds of falsifying evidence. Structural exclusion is preferable when the invalid behavior can be removed from the admissible space entirely.*

Start with prevention. If the forbidden state can be removed structurally, remove it. DocAble's pervasive work-item ownership rule is enforced by a database compare-and-set: an update succeeds only while the row remains in the expected prior state. Concurrent claimants race on the same transition, and only one can win. For that local ownership property, the runtime constraint can enforce the required transition directly; a temporal specification would answer a broader question than the one being asked.

When the property lives in interleavings, reduce the relevant state until it can be explored. DocAble's in-flight lease has this shape: a worker can appear stale and be reclaimed, another worker can acquire a fresh epoch, and the first can later wake and attempt release. The ownership question does not require the rest of the implementation. A reduced transition model retains only lifecycle state, lease epochs, reclaim, release, and the variables needed to state the invariants.

A small explicit-state checker exhaustively explores the resulting bounded state space. Separate checks compare selected production paths with the abstract model. These establish different things: exhaustive exploration searches the bounded model; the implementation checks provide evidence of correspondence. They do not establish formal refinement.

Some obligations cannot be falsified by a finite bad state. DocAble's serverless recovery path carries the temporal requirement that every submitted job eventually reaches a terminal state. A finite test can show that one execution terminated; it cannot show that no fair execution can remain stranded forever. For such liveness properties, DocAble uses small TLA+ specifications and TLC to reason about the temporal model. The representation and its checker answer a different question from the database CAS and the finite-state safety checker.

The three cases make the rule concrete: make violation impossible when the action space can be closed; exhaustively search a bounded model when the risk lives in finite interleavings; use temporal model checking when the obligation ranges over executions. None is automatically stronger engineering. Strength means adequate evidence for the claim that matters.

When a generated input exposes a defect, fix the **stable obligation** rather than merely the seed. The durable repair should close the class of behavior exposed by the counterexample whenever that class can be stated. A one-byte crash may reveal a parser invariant; a bad transition may reveal a missing state rule; a valid-but-unusual producer may reveal that the authored input grammar was incomplete.

Where available, real producers are valuable sources of dialects and edge cases that a synthetic generator may miss. Build richer synthetic generation when the engineering claim justifies the additional model and upkeep.

> *Three properties, three mechanisms*
>
> - **Single ownership.** The property is local enough to encode structurally. A PostgreSQL compare-and-set admits the ownership transition only while the row remains in its expected prior state. Concurrent claimants cannot both win.
> - **Lease fencing.** The property lives in interleavings. DocAble maintains a reduced executable transition model and exhaustively explores its bounded reachable state space; selected runtime paths are separately checked against that model. Turning off the epoch fence produces counterexamples; the production configuration produces none within the explored bound.
> - **Eventual termination.** The property is temporal. A serverless job must eventually reach a terminal state even when individual pushes are lost. A TLA+ model states that liveness obligation and can expose the stranded-job execution when the recovery sweep is removed.
>
> The important difference is not the tool. It is the shape of the property.

4.3.4 Name the Obligation Set Before Claiming Coverage

Coverage answers a question about a population; the population must be named before the percentage has engineering meaning.

Ordinary test coverage starts from the implementation, or from the tests that already exist. It can tell you a line was never executed. It cannot tell you that an important behavior has no test at all, because nothing first named that behavior as an obligation. Implementation-derived coverage cannot reveal an obligation that was never represented in the population being measured.

Explicit models can supply the missing census. A state model enumerates the transitions that require evidence. An architectural model enumerates the forbidden or permitted edges. An error model enumerates the failure paths. A requirement model enumerates the claims that must be discharged. Derive that set from the models — every seam that owes a fuzzer, every failure edge that owes an injection test, every invariant that owes a checker — and a missing test becomes a named finding rather than an absence nobody notices.

From there, the assurance sequence is straightforward. Figure 4.3-3 lays it out: name the model, derive the obligations it implies, take the census, choose evidence appropriate to each claim, and only then claim coverage against that set.

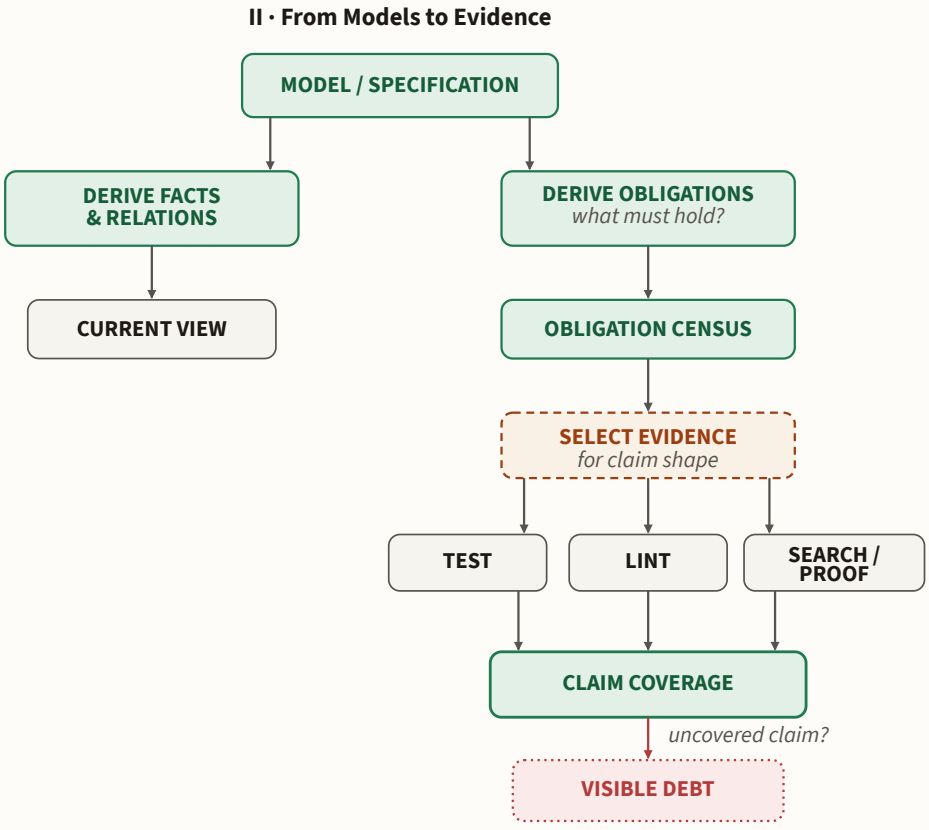


Figure 4.3-3: From models to evidence. *Coverage becomes meaningful only after the engineering obligations have been named. Explicit models can derive that population; tests, lints, searches, proofs, and human review then supply evidence appropriate to each claim.*

With the census in hand, a high aggregate percentage stops being reassurance and becomes a place to look: *this invariant has no exercising test.*

4.3.5 Did the Search Cover the Claim?

Generative validation adds an honesty question that a small example suite can often avoid: **did the campaign search the semantic region the claim is about?** Code coverage answers where execution went. Input-space coverage answers what kinds of cases the generator produced. Neither alone proves the relevant obligation was exercised.

Where traceability exists, a more semantically targeted question becomes possible: which **model claims** were actually exercised? Follow an invariant, transition, or architectural relation to the code that realizes it, then ask whether the campaign reached that implementation under evidence relevant to the claim. A high aggregate percentage can then become a named gap: *this invariant has no exercising test.* The degree question — *how much* coverage is enough — belongs to the metrics treatment in Operating MAGE, which owns the discipline to **measure one level deeper** than a raw percentage.

Coverage remains evidence about the search, not proof of correctness. Its purpose is to keep claims such as "we fuzzed it" or "the model is tested" from becoming ceremonial. Figure 4.3-4 draws the loop: a generator makes inputs, the system runs, an **independent oracle** judges each outcome, a counterexample feeds back to shrink or diagnose, and coverage asks whether the search reached the claim before refining the generator.

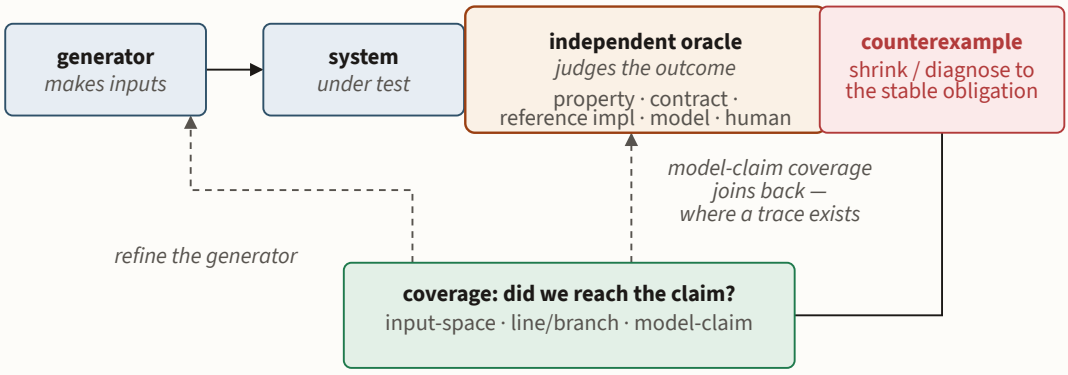


Figure 4.3-4: Generate, Judge, Search Again. *One loop, whatever the oracle's source: a **generator** makes inputs, the **system** runs them, an **independent oracle** judges each outcome, a **counterexample** feeds back to shrink or diagnose, and **coverage** asks whether the search reached the region the claim is about — refining the generator, and where traceability exists joining model-claim coverage back to the claim. The oracle may be a declared property, a robustness contract, a reference implementation, or a structured model; the loop does not care which.*

4.3.6 Give the Verdict the Right Consequence

Validation may report or gate. Where a verdict should control admission, place that authority outside the producer's discretion. At consequential boundaries, consider re-deriving evidence whose freshness or independence matters to the admission decision rather than relying automatically on an earlier marker. A fuzz campaign may run nightly and report counterexamples without controlling merge; a cost validator may stay advisory; a security invariant may deserve immediate refusal. **Authority is a separate design decision from evidence quality.**

When such a campaign carries gating authority, the environment evaluates the obligation rather than relying on the producer's previous report.

4.3.7 Two Boundaries for Evidence

Part III gave a placement rule for authority: **evaluate an obligation at the earliest boundary where it becomes legible and enforceable.** That catches a problem close to its cause. A malformed brief should be rejected before an agent spends an hour acting on it. A structural violation visible at compile time should not wait for deployment. Early evidence shortens the feedback loop and keeps later work from compounding a defect that was already knowable.

Consequential work often deserves a second boundary: **re-evaluate at the last safe point before the consequence becomes difficult to reverse.** The two rules do not oppose each other. They answer different questions. The first asks: when can this property first be decided honestly? The second asks: what is the last point at which stale or invalid evidence can still be caught before the consequence?[21]

[21] The security analogy is time-of-check to time-of-use (TOCTOU): a property established at one instant may no longer hold when the protected action occurs because relevant state changed in between. The problem here is broader than the classical TOCTOU race, but the engineering instinct is the same. Early checks establish defects cheaply; a consequential boundary may still need fresh evidence about the state actually being admitted.

Figure 4.3-5 draws the span between the two.

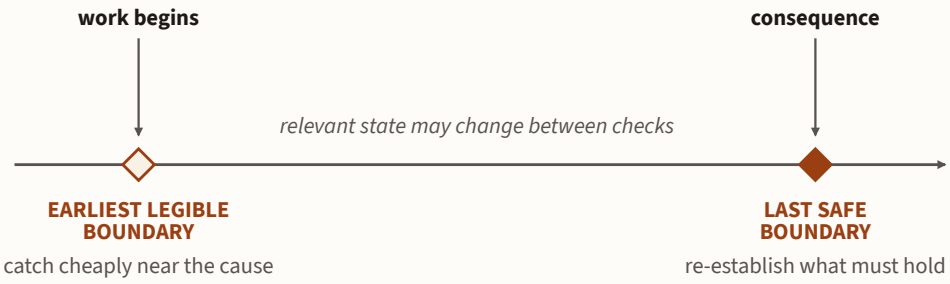


Figure 4.3-5: Two evidence boundaries. *Evaluate a property as soon as it can honestly be decided, but for consequential work, a later boundary may re-establish evidence whose freshness matters before admission or exposure. Early evaluation limits wasted work; final evaluation protects against stale evidence and intervening change.*

The second check is not a substitute for the first, and it need not repeat every earlier check. Where a second boundary is warranted, re-run or otherwise re-establish the evidence whose freshness matters to admission. A test result recorded hours ago describes the revision that produced it. A review verdict describes the change that was reviewed. A deployment rehearsal describes the configuration it exercised. If relevant state can change before admission, the evidence can cease to describe the artifact now crossing the boundary: a recorded claim is a statement about the past, not the present, and it rots as sibling work churns the ground under it.

Done is a claim, not a stored fact. At a consequential boundary, ensure that the evidence still justifies the consequence; where freshness cannot otherwise be established cheaply, re-derive the relevant evidence rather than trusting a stale green checkmark. The gain compounds with velocity: the more often a system ships, the more often an un-gated build reaches someone, so a cheap re-check at the last safe boundary is worth more, not less, as release frequency rises.

4.3.8 Why Abundance Changes the Economics

None of these techniques is new. Property-based testing, fuzzing, static analysis, model-based testing, independent review, and automated admission all predate coding agents. What changed is the relative price of implementation and inspection. When implementation was scarce, human attention could sit close to every change. When implementation becomes abundant, repeatable assurance mechanisms become increasingly valuable because human inspection does not scale with implementation volume.

The engineer does not disappear from validation. Human judgment moves toward the decisions for which it has the highest marginal value: choosing obligations, designing representations, selecting evidence, calibrating validators, deciding which verdicts deserve

authority, and resolving the cases the environment cannot decide honestly. The old techniques become more central because the economics around them changed.

WORKED EXAMPLES

Takeaway. Separate producer from grader. Generate evidence built to falsify the claim, and place admission where the producer cannot redefine success.

4.4 Operating MAGE

A method has to survive ordinary days, not only successful changes. Repositories fill disks, branches conflict, agents disappear, queues stall, tests flake, deployments degrade, and assumptions age. Engineers often know how to recover, but that knowledge may live only in someone's head, where an autonomous agent cannot reliably reach it. MAGE therefore brings the same modeling and authority disciplines into operation. Lifecycle models represent healthy progress and important failure states. Events make consequential transitions observable. Runbooks preserve sanctioned reactions, including where judgment remains. Metrics and measurement models expose the quantities needed to judge whether the loop is healthy. Together they support an operating loop: recognize the state, surface the right reaction, take the appropriate action, and observe what happened next.

These structures play different roles. A lifecycle is a model; an event supplies evidence that a transition occurred; a runbook preserves a procedure; and a metric supplies measurement evidence. None is authoritative merely by existing. Constraints, validators, and gates give selected operational obligations consequence where the required state is legible and the evidence adequate. Keeping those roles separate lets the operating environment grow without collapsing representation, evidence, procedure, and authority into one mechanism.

4.4.1 Model the Lifecycle

Start with the healthy path of a recurring activity: how a bug moves from reported to fixed, how a feature moves from designed to shipped, or how an agent moves from dispatched to landed. Then name the failure states that matter and the sanctioned recovery from each. The result is a lifecycle model: a behavioral model specialized to operation, retaining the states and transitions needed to recognize healthy progress and recover from important breaks.

FMEA — failure mode and effects analysis — offers a useful neighboring practice: enumerate important failure modes before recovery is needed. MAGE represents the states, legal transitions, and sanctioned recoveries needed to operate the lifecycle.

Pair prohibitions with recovery paths. "Never use this destructive merge command" leaves a stuck agent with a prohibition and no exit. "When this state occurs, do not use X; use Y because it preserves the already-applied work" supplies the missing path. Where the prohibition can be made structural, encode it as authority. Where it cannot, preserve the guidance explicitly.

A prohibition without a recovery path can strand the actor. DocAble encountered this when agents used a destructive Git escape during stuck merges. The durable rule named the forbidden action and the sanctioned recovery that preserved already-applied work.

The fleet's own lifecycle deserves the same treatment as the product. Point the discipline at the substrate that *produces* the software and you get the **agent-orchestration model** — an agent moving through explicit states (`Dispatched → Working → Landed → Tombstoned`, with `Abandoned` and recovery paths), and the orchestrator's refill-and-bank loop as the journey

that drives it. Once those states are explicit, the environment can check the transitions it can decide mechanically — refusing, for example, to tombstone work before the required commit exists. Not every lifecycle property is a state-transition check. *A landed agent eventually tombstones* is a liveness claim: no single bad state falsifies it, so the obligation routes to a temporal checker instead. The lifecycle supplies the vocabulary; the shape of the obligation determines the evidence.

That state machine is also why an operator never takes a report at face value. **"Done" is a claim, not a fact.** An agent reporting a task complete is describing its *intent*, not the state of your repository, and the two drift apart constantly — a sibling change broke a test, a marker rotted, the thing it "verified" quietly routed through a fallback. The report therefore should not be confused with repository state. Where completion is consequential, the environment may re-evaluate the relevant entry conditions against current state rather than relying on the report alone. In DocAble, the lifecycle names `Landed` as a state with an entry condition, and the orchestrator re-runs the relevant gates at HEAD before accepting that transition.

4.4.2 Attach Reactions to Lifecycle Events

A lifecycle model says which states matter. Operation improves when an important transition can also make the right procedure available at the right moment. Do not rely on an actor to remember to inspect the lifecycle, notice that a transition happened, retrieve the right procedure, and invoke it. Where a recurring event is mechanically observable and a standard response is useful, bind the reaction to the event.

A mechanical trigger does not require a mechanical response. **The firing can be deterministic while the payload still contains judgment.** An agent reaches a context threshold: a hook deterministically requests a handoff, but the agent must still summarize what matters. A deployment fails a health check: the event deterministically opens the recovery path, but an engineer may still decide whether rollback is warranted. A repeated failure is observed: the reflection procedure runs on its own, but whether that failure deserves a new mechanism stays an engineering decision.

Bind known failure events to their sanctioned reactions: the reaction fires rather than waiting for an operator to notice the failure. The environment emits structured events, and each can trigger the appropriate reaction, reducing dependence on an operator noticing and polling the signal manually. **Automate the arrival of the decision procedure.** Figure 4.4-1 draws the full path from a lifecycle transition to the next state.

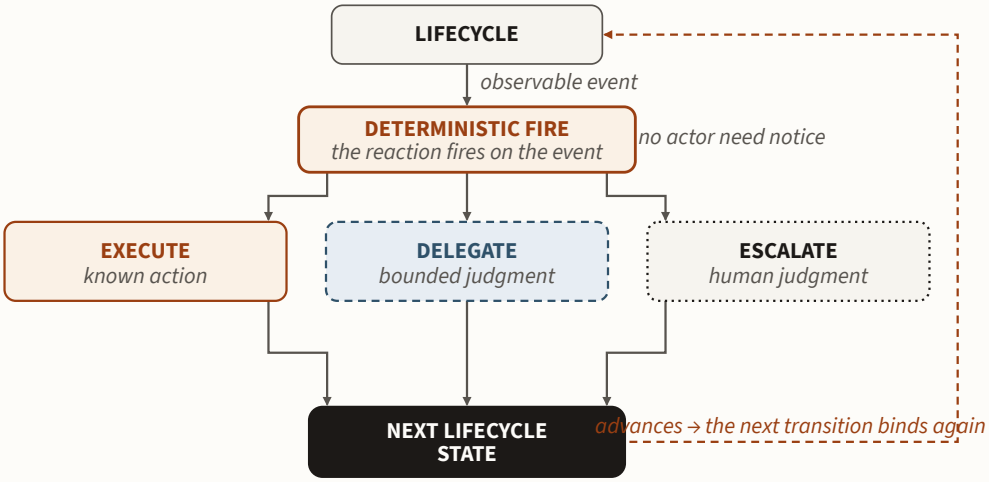


Figure 4.4-1: Binding a Reaction to a Lifecycle Event. *A lifecycle transition emits an observable event that fires the reaction deterministically, though its payload still splits three ways — execute a known action, delegate a bounded judgment, or escalate to a human. The firing is mechanical; the procedure it delivers may still hold judgment.*

4.4.3 Externalize Operational Judgment

A runbook preserves how an experienced engineer reasons through a recurring situation, including which steps are mechanical and which require judgment.

Inset — Runbooks PRACTICE

A runbook is a prepared procedure for a recurring operational task or failure. Site reliability engineering uses runbooks and incident playbooks to preserve diagnostic and recovery knowledge before an outage requires it[53]. An empirical study of real incidents shows how variable that recovery work is[66]. Executable runbooks can combine commands, scripts, API calls, and manual steps into a repeatable workflow[67].

For agentic operation, one useful addition is to record who or what can decide each step. Deterministic steps can execute directly; bounded judgments can be delegated with prepared context; irreducible judgments stay with the operator.

Runbooks can drift. Their procedures therefore need the same maintenance discipline as other engineering models.

A runbook preserves the sanctioned response to a recurring operational situation, but not every step has the same decision structure. Some steps have one mechanically determined outcome; some require bounded reasoning; some require a human decision. Type each step accordingly: **Execute** when the outcome is mechanically determined and can be preserved as an executable operation; **Delegate** when the step requires bounded judgment, supplying the relevant state, choices, criteria, and required output; **Escalate** when the decision remains irreducibly human, preparing the evidence and surfacing the decision explicitly. This is the same split used by event reactions: the event determines when the procedure arrives; the runbook determines what happens next. Figure 4.4-2 shows the three kinds and the form each takes.

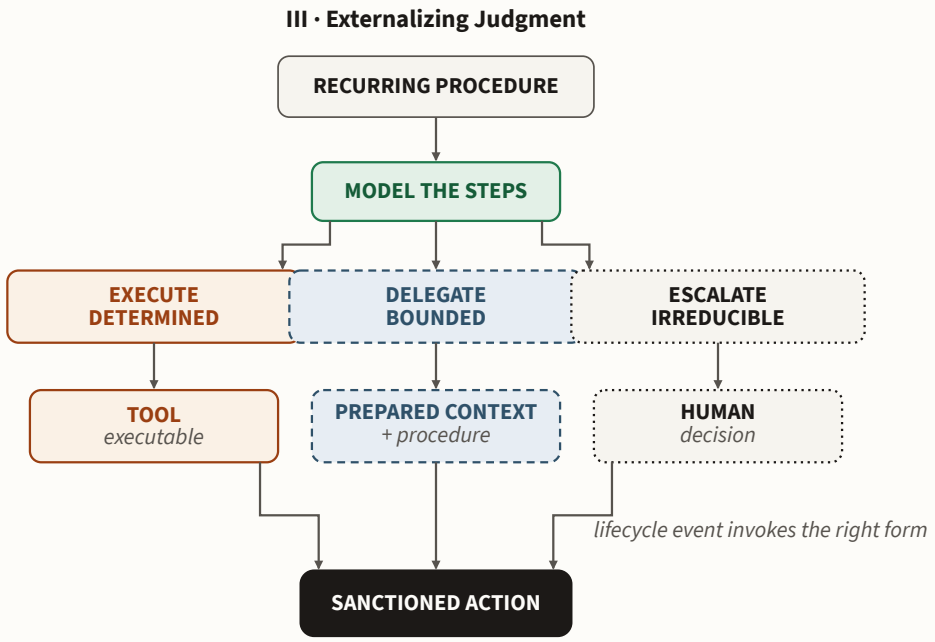


Figure 4.4-2: Externalizing operational judgment. *Each runbook step is typed Execute, Delegate, or Escalate: an Execute step becomes an executable tool; a Delegate step receives prepared context and a decision procedure; an Escalate step is surfaced explicitly to a human.*

A structured operational playbook can serve as a model of engineering practice. It externalizes not only what to do, but which kind of reasoning each step requires — and it connects directly to the next chapter, where a mastery skill packages exactly this kind of recurring judgment.

Where possible, surround a judgment-laden step with deterministic work. Prepare the data mechanically before the decision; after the decision, execute and measure mechanically. Preserve the rationale for the judgment separately from the trace of what ran. This does not make the judgment deterministic; it bounds where judgment occurs while leaving the surrounding work replayable and checkable. Figure 4.4-3 draws the shape.

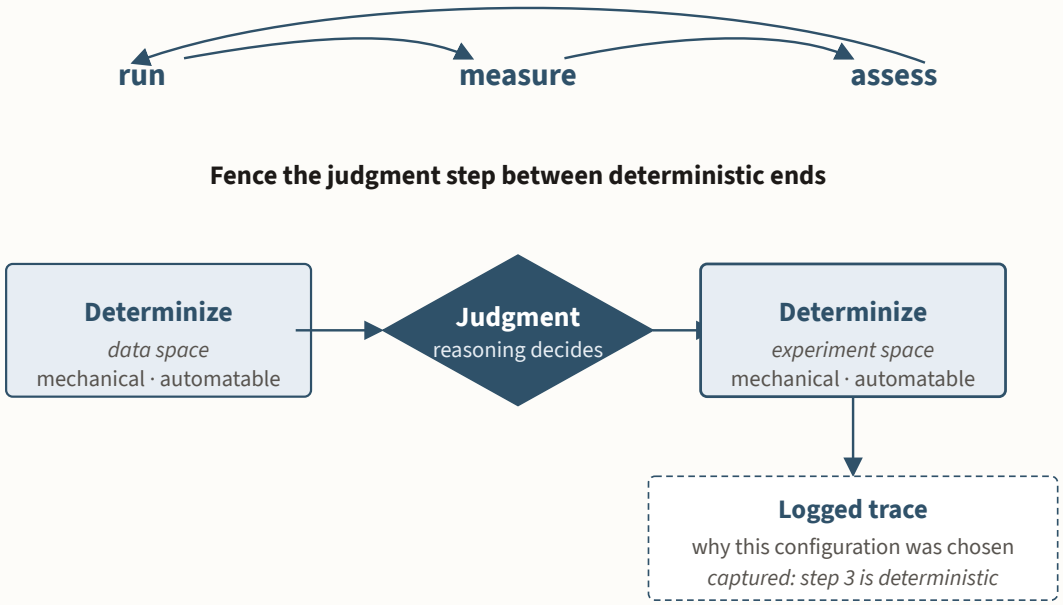


Figure 4.4-3: Surrounding a judgment step with deterministic work. *A run-measure-assess loop wraps a three-step spine: determinize the data space, apply judgment, then determinize the experiment space. Bounding the one judgment step between deterministic ends leaves a logged trace a later checker can evaluate.*

The split is not unique to DocAble. Systems that share no code and answer to different regulators reach for the same fence between the step a machine can decide and the step that needs a mind.

WORKED EXAMPLES

Siemens. Draws the fence as a clean division of labor across a safety-critical toolchain: probabilistic semantic reasoning upstream, deterministic physics and verification downstream. The decidable obligation lands on a deterministic mechanism; the open-ended reading of intent stays with the model.

Zenseact. Hardened three judgments other sites leave probabilistic — execution through deterministic tools, authority settled by an explicit permission table, and context selection decided by a router rather than a guess.

Docker. Determinizes what carries hard evidence — the tests run as deterministic gates — but keeps admission deliberately probabilistic, an independent reviewer plus a human, because collapsing that call to a threshold would forfeit the judgment it exists to weigh.

DocAble. Routes structural work (reading a tag tree, checking reading order) to deterministic tools that leave a replayable trace, and judgment-laden work (describing an image) to a model against a rubric, because no script can write the alt text.

Takeaway. Separate what can be decided independently from what still requires judgment. Mechanically decidable steps can become executable tools and replayable traces. Qualitative decisions can receive better representations, context, procedures, tools, and evidence without pretending to eliminate the judgment they require. Where stronger authority becomes feasible, give the relevant obligation to the environment; where consequential judgment should not be delegated, escalate it to a person. The objective is not to remove judgment from engineering, but to stop repeatedly spending judgment on obligations the environment can decide more reliably.

4.4.4 Measure the State That Matters

Operational measurement should expose the state the decision actually depends on. A worker count may be a poor proxy for memory pressure; measure the pressure. Process isolation may not imply resource isolation; mediate the shared resource. A marker on disk may not establish liveness; record lifecycle state explicitly. When an aggregate hides the decision you need to make, measure one level deeper — not arbitrarily deeper, but far enough to expose a useful engineering target. Keep measurement separate from authority: a metric supplies evidence, a validator may interpret that evidence against an obligation, and a gate may act on the resulting verdict.

Inset — When a measure becomes a target ECONOMICS

Engineering metrics are proxies for properties we care about. Line coverage stands in for something about test adequacy; defect counts for something about quality; lead time for something about delivery performance. The proxy is useful only while it tracks the consequential property well enough for the decision being made.

Goodhart's law warns what can happen when the proxy itself becomes consequential: once a measure becomes a target, optimizing the measure can cease to improve the property it was meant to represent. A team rewarded for closing tickets may split or prematurely close them. A coverage threshold may encourage tests that execute lines without testing important behavior. An agent rewarded for reducing lint findings may suppress findings rather than remove the underlying defects. In each case, the measured number improves while its correspondence with the engineering concern weakens.

The lesson is not to avoid metrics or targets. It is to treat the metric as a model whose correspondence matters. Prefer measures closer to the consequential state; combine signals when one proxy is inadequate; and be especially skeptical when a metric

> acquires authority or incentives. The stronger the consequence attached to a measure, the stronger the case needed that improving it still means improving what you actually care about.

Improve the signal when a proxy drives a consequential decision and no longer tracks the relevant state.

An operating loop needs evidence about its own state. A useful metric changes the next engineering decision. A whole-project cloud bill tells you what you spent; cost by service may tell you what to change. Aggregate line coverage tells you how much source ran; claim-level coverage may tell you which invariant your suite never exercised. Counting AI-assisted pull requests or generated lines tells you activity changed; pairing throughput with yield and rework[68] gives a more informative view of engineering outcomes than activity counts alone.

Ousterhout's rule is apt[69]: **measure one level deeper.** One level, not arbitrarily deep. A metric earns its maintenance cost when the deeper cut exposes an optimization target, failure class, or decision the surface number hides. An aggregate such as "87% line coverage" reports activity at a level that may hide the engineering claim that matters — a line in a throwaway string helper counts the same as a line in the job state machine.

Some evidence is mechanically decidable and appropriate for automated admission; other evidence is intentionally judgment-laden and should point a reader toward a decision rather than masquerade as a threshold. The next two examples sit at opposite ends of that spectrum.

One hard example: coverage measured one level deeper

Requirements-based coverage joins model-to-code traceability with ordinary test coverage: it follows each modeled claim to the code that realizes it, then to the tests that exercise that code, and reports which claims have no exercising evidence. It exists because ordinary line and branch coverage measure whether code ran, not whether the properties that matter were checked — a suite can report ninety-five percent line coverage while an entire requirement stays untested. Consider a requirement that uploaded documents over a size bound are rejected without crashing: the traceability chain names the guard that enforces the bound and the test that drives an oversized file through it, so a missing test shows up as an unexercised claim rather than a slightly lower percentage. The payoff is a mechanically detectable gap the environment can name — *this invariant is untested* — instead of an aggregate line percentage. Because that detection is a deterministic join over explicit models, the example sits at the mechanized end of the spectrum.

One judgment-laden example: doc-derived tests

Doc-derived tests are generated from consequential claims in prose documentation and keep a trace back to the source claim. They exist because documentation makes promises the code can quietly stop keeping; turning a promise into an executable check keeps the two

aligned. Suppose the documentation promises that a failed conversion preserves the user's original file. The derived test induces a conversion failure and asserts the original survives, recording the link from that sentence to the test and its result. Machinery can preserve this claim-to-test-to-result trace and re-run it on every change. What machinery cannot settle is whether an arbitrary sentence expresses a consequential, testable claim at all, and whether the generated test captures what the sentence meant — both can require expert judgment. Because the semantic adequacy of that translation stays judgment-laden, the measurement should direct a reviewer's attention rather than drive an admission threshold, which is why it sits at the judgment-laden end of the spectrum.

Let useful models drive machinery

An executable model earns more than documentation value when tools consume it directly. A deployment-topology model can tell a latency analysis which services and paths matter. A traceability graph can tell a coverage tool which code realizes a particular claim. A component graph can help a validator decide whether a finding is local to the current change or belongs to an unrelated part of the system. This is one of Modeling's practical payoffs: the representation becomes a reusable query surface. It does not itself supply authority — the consuming validator or gate determines what follows from the model-derived result.

A model can also scope a mechanism. DocAble locates a finding and the current change in the component model and uses their relationship to determine which admission boundary owns the finding. A local finding can therefore block the current change, while unrelated debt remains the responsibility of a broader gate.

The grade is evidence read from the model at check time; the gate carries the authority. Read the number one level below the surface, where it names a target instead of a total.

4.4.5 The Operating Rhythm

Put the pieces together and operation becomes a recurring rhythm. The lifecycle tells you what healthy progress and important failure look like. Events bind the reaction to the transition, so the right procedure arrives when it matters. The runbook gives each state a sanctioned next move and separates deterministic execution from judgment. Metrics expose enough state to tell whether the loop is improving or merely busy. When a break recurs, governance conversion asks what later work should inherit from the lesson.

The operating loop will still stall. When it does, resist the reflex to add another mechanism. First diagnose what kind of problem you have. A searcher problem calls for better search or reasoning; a model problem for better representation; an oracle problem for better evaluation; and a target problem for stopping long enough to learn what the system should actually do.

Inset — When work stalls, diagnose before adding machinery PRACTICE

Repeated failure does not always call for another control. Before you build one, ask which part of the loop is limiting the work.

What may be limiting	The question	If that is the gap
Searcher	Can the available intelligence find a good candidate?	Change the model, tools, decomposition, or search strategy.
Model	Is consequential structure missing from the representation?	Make it explicit.
Oracle	Can the environment tell good candidates from bad?	Strengthen the evidence, validation, or simulation.
Target	Do we know what "good" even means?	Stop building and learn.

A stronger reasoner does not settle an unknown requirement; another validator does not supply a missing abstraction; more modeling cannot answer a product question whose answer must come from users or deployment. Diagnose what is limiting the work before adding machinery — the same distinction governance conversion already draws between failures of knowledge, obligation, evidence, evaluation, and consequence.

Run this rhythm long enough and recurring operation becomes less dependent on memory. Healthy paths become explicit; known failures summon known responses; mechanical steps execute mechanically; judgment arrives with prepared evidence; and repeated breakdowns feed back into the engineering environment. The engineer increasingly decides what the environment should know, what it should decide, and what must remain judgment. Part VI returns to that professional shift.

4.5 Packaging the Method as Skills

Some recurring engineering knowledge should acquire authority in the environment. Other knowledge properly remains available to the reasoner. That knowledge can still become engineering capital: package the representations, examples, heuristics, decision procedures, and tools that improve recurring judgment so that later work inherits them rather than reconstructing them from a generic prior.

A skill is one such package. It can range from a compact procedure resembling a disciplined prompt to a substantial body of explicit domain knowledge with structured representations, examples, correctness conditions, retrieval, and executable tools. The relevant distinction is not how elaborate the package is, but what it lets a fresh agent inherit. A useful skill gives the agent some of the abstractions, distinctions, evidence, and practices that an experienced practitioner would otherwise have to reconstruct or supply interactively.

Packaging does not itself create authority. Prose, examples, models, and decision procedures can make a satisfactory decision substantially more likely while leaving the decision probabilistic. The same skill can also invoke authoritative mechanisms or carry the models and correctness conditions those mechanisms consume. Where an obligation should not depend on the agent's reasoning, the environment still needs a constraint, validator, gate, permission, or other mechanism that can act on that obligation independently.

Two useful forms recur. A **tool skill** packages a repeatable procedure around a particular tool or interface. A **mastery skill** packages domain representations, knowledge, examples, heuristics, correctness conditions, and a repeatable decision procedure for doing a class of work well. The companion repository demonstrates both patterns through self-governance, self-operate, and self-communicate.

A skill is therefore not confined to one side of the improve-judgment / give-authority division. Its direct instructions may support a reasoner's judgment, but it can also carry or invoke the machinery through which selected obligations acquire authority.

Recent industry examples show related packaging moves.[22] Twilio's public account is sharp: an unguided agent asked to "verify users" hand-rolls a six-digit one-time password, while its skill redirects to a dedicated verification product and inherits rate limiting, fraud controls, and routing the operator never thought to request[71]. The encoded procedure is the capital; the class of work becomes inheritable.

4.5.1 What a Reusable Skill Needs

A reusable skill needs three things: the representations needed to reason about the work, the recurring decisions that matter within it, and a procedure for making those decisions. The representations preserve the distinctions needed for this class of work; the recurring

[22] Atlassian's Jira team reports a related move: it delegates keep-the-lights-on work — flaky tests, feature-flag cleanup, vulnerabilities, accessibility — by turning each into an explicit work item that carries context plus workflow instructions, so a maintenance class becomes delegable once the tacit repair procedure is encoded, not because the agent got clever[70].

decisions capture the alternatives, failure modes, and tradeoffs that repeatedly matter; and the procedure determines when to invoke that knowledge, what tools or evidence to use, and what result to produce. The point is not the particular file format or vocabulary. It is to turn tacit practice into a bounded reasoning environment that a fresh agent can enter repeatedly. Figure 4.5-1 draws the shape.

The resulting package can be small or substantial. Some skills need only a few distinctions, examples, and a procedure; others need explicit models, examples and counterexamples, correctness conditions, retrieval rules, or executable tools for obtaining and checking evidence. Package the smallest substrate that reliably lets later agents inherit the relevant expertise rather than rediscover it.

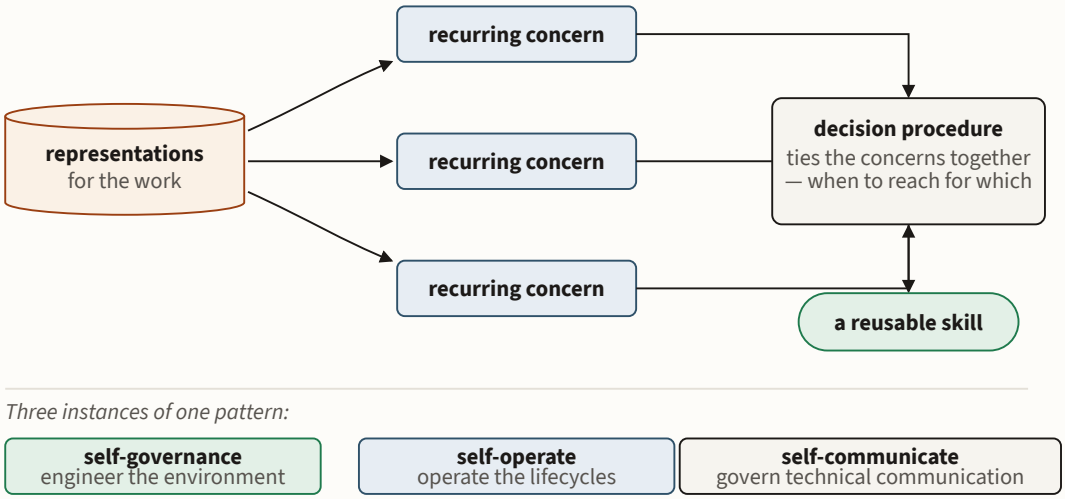


Figure 4.5-1: What a reusable skill needs. *A domain model, the recurring concerns that matter within it, and a decision procedure that ties them together compose into a reusable skill a fresh agent can enter repeatedly.*

4.5.2 Deliver the Right Context—and Measure It

Packaging knowledge solves only half the problem. The agent must also receive that knowledge when it matters. Loading everything all the time is the obvious solution, and often a bad one: every rule, model, example, procedure, and failure lesson competes for attention, so a larger context can make navigation harder rather than easier. The alternative is to separate what must remain continuously salient from what can be retrieved when the task makes it relevant.

Dynamic Context Injection (DCI) is one implementation of that pattern. The environment observes something about the current task, work surface, or lifecycle state; selects relevant engineering knowledge; and places that slice into the agent's working context. The standing context therefore carries the small amount of knowledge that should remain continuously available, while task-specific models, rules, examples, or procedures arrive when triggered by the work. The general pattern is shown in Figure 4.5-2.

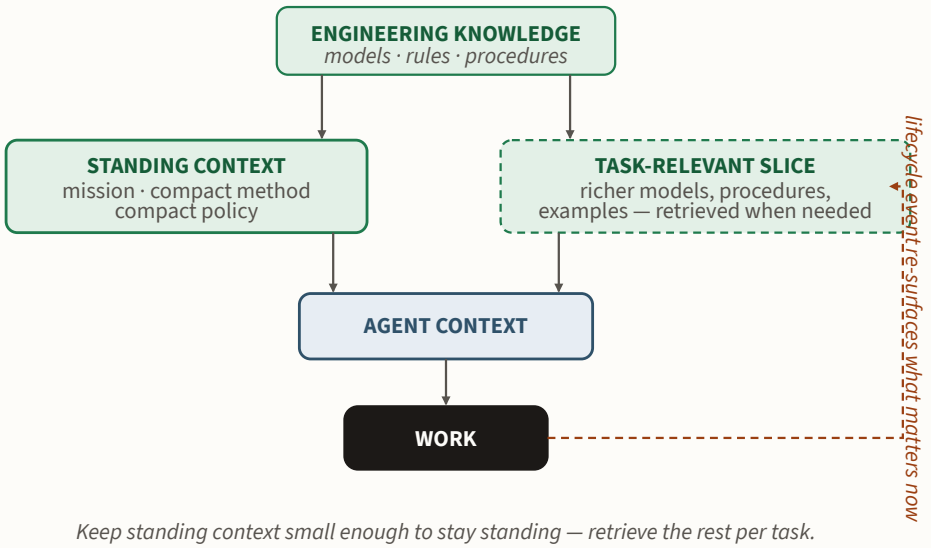


Figure 4.5-2: Delivering task-relevant context. *Engineering knowledge — models, rules, procedures — splits into a small standing context (mission, compact method, the few policies whose salience must survive every task) and a task-relevant slice retrieved when the work calls for it. Both feed the agent's context; a lifecycle event can re-surface what matters now.*

Standing context should contain the mission, compact method, and the few policies whose salience must survive every task. Retrieve richer model slices, procedures, examples, and local constraints according to the work. Where relevance can be determined mechanically, inject the appropriate slice rather than relying on the actor to discover and retrieve it.

But context has a cost. More guidance is not automatically better guidance. It consumes tokens, attention, and reasoning capacity, and stale guidance can actively misdirect work.

So treat context delivery as an engineering intervention and measure its effect. Its value may be probabilistic but is still empirical: compare realized performance with and without the added knowledge. A prompt, example set, model, or skill earns its carrying cost when supplying it makes satisfactory outcomes more likely or cheaper to obtain. Task success comes first; token counts, retrieval hit rates, and similar measures help explain why. A context mechanism that consumes more reasoning budget without improving the engineering outcome has not earned its place.

The model-based software-engineering pilot is one example: run the same task against different model-and-context conditions and reasoner classes, and ask whether the supplied

representation actually changes navigation cost or task success. What is portable is the discipline of testing whether engineered context does what we claim it does. Part V carries that pilot with its data and its limitations.

Context delivery changes what the reasoner can see; by itself, it does not change what the environment permits. A richer slice of the right rules makes a good action easier to find; only a constraint, validator, or gate makes a bad action unavailable.

4.5.3 Self-Governance: Engineering the Environment

Self-governance packages the MAGE method itself. It helps an agent ask the recurring questions: What should be represented? Does the representation still correspond to the system? What evidence exists? What deserves authority? How should a new mechanism be introduced? Has existing structure stopped earning its upkeep? The skill does not contain the system's concrete engineering knowledge; it teaches the agent how to reason about that knowledge. Models and tools in the governed environment supply the system-specific facts.

Inset — MAGE Can Help Build Its Own Environment PRACTICE

The MAGE method can itself be packaged as a skill. The companion self-governance skill gives an agent the recurring concepts and procedures developed in this book: identify the relevant models, state the obligation, distinguish guidance from authority, locate the boundary where evidence becomes available, choose an appropriate mechanism, and preserve useful structure as engineering capital. Models and tools in the governed environment supply the system-specific facts.

This creates a useful division of expertise. A practitioner can work with the agent in the language of the domain: this must hold; this evidence matters; this decision remains subjective; this failure keeps recurring. The skill supplies a procedure for translating those statements into candidate models, guidance, and controls. Commodity coding intelligence can then realize that structure using the mechanisms available in the environment: schemas and types, constrained interfaces, static analyses, hooks, permissions, validators, admission gates, provenance, telemetry, or other machinery.

The division is particularly useful when the practitioner is not a software engineer. An accountant need not know in advance that a recurring obligation is naturally represented by a schema constraint followed by a workflow gate. A scientist can identify the provenance that must accompany an observation without knowing how to implement the repository hook that checks it. The agent can supply much of the implementation knowledge. The domain expert remains responsible for the correspondence claim: whether the resulting representation or mechanism captures the intended obligation.

That translation does not establish its own correctness. Agent-generated machinery requires evidence appropriate to its consequence, just as other generated artifacts do. Nor should every judgment become machinery. The same procedure may identify

> better context, a mastery skill, an experiment, or human review as the appropriate intervention when stronger authority is not feasible.

Figure 4.5-3 traces this division of expertise.

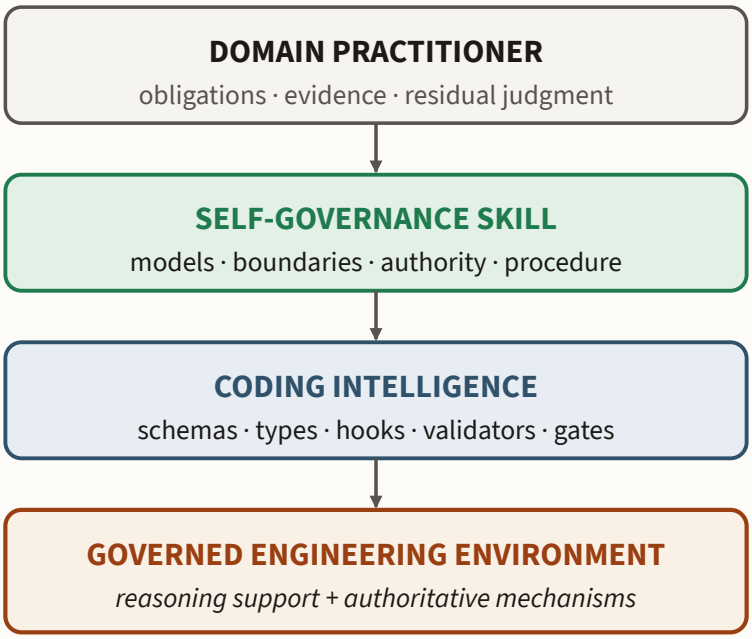


Figure 4.5-3: MAGE as agent knowledge. *Domain expertise supplies the obligations, evidence, and judgments that matter; the self-governance skill supplies a procedure for reasoning about them; and coding intelligence realizes the resulting models, guidance, and controls in the available environment. Consequential controls remain subject to the same correspondence and evidence requirements as other agent-generated artifacts.*

See Appendix E for the self-governance skill.

Where useful, the skill queries the environment's **governance catalogue** of existing mechanisms — a model of the method, not a catalogue of preferred controls.

The project's boot context can preserve mission, method, and concise rules, but treat those rules accurately. Mission orients. Method teaches judgment. Rules state project policy. **They take effect only where an environmental mechanism enforces them.** A numbered sentence in an always-loaded file is still guidance unless something outside the producing agent enforces it.

Underneath the method sit the operating rules the skill applies on every touch, the same set the companion skill carries:

SELF-GOVERNANCE — OPERATING RULES

- **Model before guessing.** Reach concrete system knowledge through a provider; do not reconstruct it from memory or raise confidence by rhetoric.
- **Ask what should become machinery.** At every failure, ask whether the lesson is durable — and default to *nothing.*
- **Method before tool.** Choose the move the situation calls for, then the capability that performs it.
- **Right-size the fix.** Prefer the smallest sound change that closes the class; propose a larger intervention rather than installing it reflexively. Prefer prevention where the legitimate action space can honestly be narrowed.
- **Escalate a genuine judgment, not an inconvenience.** A real authority-or-consequence boundary earns a prepared decision handed upward; a merely hard problem does not.

A skill on its own is passive: it sits and waits to be invoked, and an agent heads-down in the work will not stop to invoke it. A hook brings the skill into the work while the evidence is still fresh enough to diagnose the episode; the skill applies the MAGE procedure to the episode and **surfaces the relevant design forks** — "we keep hitting this; here are two ideas, which should I try?" — then waits. It may recommend changing a model, an obligation, evidence, a mechanism, or nothing at all. The engineer retains the judgment about what matters enough to preserve.

Governance must earn its carrying cost

Governance mechanisms and other durable engineering assets can become capital when their continuing value exceeds their carrying cost; they can also depreciate. Over-investment turns governance maintenance into its own recurring cost. Three symptoms say the return has gone negative: upkeep crowds out product work; the checks cry wolf, flagging more than they catch; or the guarded failure is both cheap and rare. Each is a control whose upkeep outweighs its benefit. Size it the way Migrating to MAGE sizes any control — against the cost and the frequency of the failure it prevents — and retire the assets that no longer earn their cost.

Even a soft mechanism can be governed like a hard one. NVIDIA's public account describes treating a skill as a deployable capability rather than a static prompt, run through an eight-stage supply chain — source, review, scan, evaluate, skill card, sign, catalog, sync — where scanning looks for agent-specific threats (hidden instructions, trigger abuse, excessive agency, tool poisoning, purpose/access mismatch), not only conventional software risk[72]. The prose is a reusable knowledge asset; the signing and gating around it add authority to its distribution and use.

4.5.4 Self-Operate: Running the Environment

Self-operate packages the operating loop of the previous chapter. It helps an agent identify the current lifecycle state, retrieve the sanctioned reaction, and execute, delegate, or escalate each step according to who or what can honestly decide it. It does not redesign the environment. When operation exposes a deficiency in a model, mechanism, skill, or lifecycle, self-operate returns that engineering problem — with evidence — to self-governance.

The skill is useful because operating procedure otherwise has to be reconstructed at every fresh session. It does not replace the underlying tools, state machines, validators, or gates. It makes their use repeatable.

The two mastery skills divide the work by the question each answers. Table 4.5-1 sets the split side by side.

Table 4.5-1: The two mastery skills, by facet. *Self-governance asks how the environment should be engineered and produces or revises durable engineering structure; self-operate asks how to run a lifecycle and produces a completed operation or returned evidence. Each hands the other exactly one thing: design flows down to be run, and engineering deficiencies flow back up to be fixed.*

Facet	Self-governance	Self-operate
The question it answers	*How should this be engineered?*	*How do I operate this lifecycle?*
Acts on	the environment — models, mechanisms, skills	the running lifecycles — dispatch, land, deploy, recover
Produces	durable engineering structure	a completed operation, or returned evidence
Hands off	*design* down to self-operate to run	*engineering deficiencies* up to self-governance to fix

Neither absorbs the other: self-governance decides how a mechanism should exist; self-operate uses it at runtime.

4.5.5 Self-Communicate: a Boundary Test

Self-communicate provides a useful boundary test because technical communication remains substantially qualitative. There is no deterministic oracle for whether an explanation is clear, whether a figure exposes the right structure, or whether a presentation gives its audience the right progression of ideas. Yet those judgments are not structureless. Audience, purpose, information hierarchy, evidence, visual form, examples, and revision criteria can be represented, and a procedure can tell the agent how to reason about them. Packaging that knowledge improves the judgment without pretending to make “good communication” deterministic.

Deterministic and probabilistic controls can coexist with that qualitative procedure. Tools can check file validity, accessibility properties, required metadata, or other mechanically decidable obligations; evaluators can assess narrower qualitative properties where their error characteristics make that useful; the agent can continue to exercise judgment over

narrative and presentation. As those judgments become better understood, additional obligations may become feasible to give authority. Self-communicate therefore exercises the same pattern as the rest of MAGE: improve the substrate for judgment, give selected obligations authority where feasible, and leave the remaining degrees of freedom open.

Self-communicate is not a third pillar of MAGE. It demonstrates that the packaging pattern can extend beyond governance and operations; the appendix and companion repository carry the procedure itself.

Reusable agent procedures can become engineering capital when later work inherits them, and the industry is finding the pattern in parallel. Part V's MAGE in the Wild sets six such systems side by side.

WORKED EXAMPLES

Shopify. Shopify's public account describes agent sessions made visible and searchable, together with mechanisms for turning reusable lessons into shared skills and defaults. The important move is that private reasoning can become organizationally reusable rather than disappearing with the session: make the work visible, then mine the repeated judgment into a skill, so the environment gets incrementally smarter with each pass instead of every session starting cold.

Zenseact. Zenseact's public account describes a centrally owned runtime and safety envelope combined with domain-owned agents, tools, instructions, and reusable expertise. Each domain team contributes an agent as little more than a directory holding a config and a skill, and a router surfaces domain expertise on demand — the organization centralizes the mechanism and decentralizes the judgment.

Takeaway. Reusable agent procedures can become engineering capital when later work inherits them. Shopify's account describes judgment mined from prior sessions into defaults later work inherits; Zenseact's describes the organizational variant, where domain teams contribute reusable expertise to a centrally governed runtime. The portable move is to preserve recurring procedure without pretending that packaging alone grants authority.

The Portable Moves

THE PORTABLE MOVES

Strip away DocAble, the companion repository, and the current generation of agent harnesses, and a small set of moves remains. These are the parts of the method meant to port.

Five such moves remain, and each can transfer to a system with different tools, a different harness, and different names.

Choose the reduction that makes the engineering question answerable without carrying unnecessary detail. Do not model the whole system because a model sounds sophisticated. Identify the recurring question, externalize enough structure to answer it without reconstructing the implementation, and stop when additional detail no longer pays.

Place authority at the earliest boundary where the obligation is legible and enforceable. Guidance can aim an agent before that point; local controls can sometimes act without a rich system model. Give an obligation authority only where the environment has the semantics and evidence needed to decide it.

Audit → Drain → Promote when introducing a future authoritative obligation into an inherited system. A rule worth enforcing tomorrow may be violated everywhere today. Observe first, repair the inherited findings under evidence, then grant the mechanism authority once the governed surface can satisfy the rule without breaking work in flight. The final authority mechanism may be the original detector or a narrower mechanism revealed by the drain.

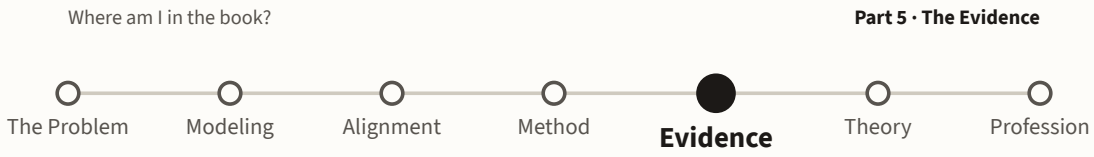
Generate evidence capable of falsifying the claim. For consequential work, do not rely on the producing agent's report alone. Choose an oracle appropriate to the obligation, search beyond the examples already known, and ask whether the search actually reached the semantic region the claim covers. Size delegated work so its result forms a meaningful unit of evidence.

Convert recurring judgment into durable engineering structure. Preserve the representations, knowledge, examples, procedures, tools, and context that make recurring judgments better. Give stable obligations authority where feasible through constraints, validators, permissions, gates, or other appropriate mechanisms. As experience makes previously qualitative obligations clearer or more evaluable, reconsider whether stronger authority has become feasible. Preserve residual judgment as a skill, runbook, or other reusable asset rather than forcing it into a mechanism that cannot honestly decide it. Maintain the resulting structure while its avoided reasoning and failure cost exceeds its carrying cost.

The method itself can become such an asset. Self-governance packages MAGE's recurring distinctions and procedures so that an agent can help identify models and obligations, decide where authority is feasible, choose candidate mechanisms, and build the surrounding engineering structure. The same pattern can package other domains whose recurring judgments benefit from explicit representations, procedures, tools, and evidence. See Appendix E for the concrete self-governance skill.

Those five moves are MAGE without its case. Appendix D collects them as bench procedures. Part V turns from method to evidence: what happened when this cycle met a system being built faster than its engineered environment could initially absorb.

Part 5: The Evidence



QUESTION THIS PART ANSWERS

What evidence supports MAGE?

DEPTH AND BREADTH

MAGE emerged from one deeply observed production build. That case supplies chronology, mechanism, and within-case recurrence.

Independent industrial accounts supply a different kind of evidence: variation across systems built by other organizations under different constraints.

The first shows how the method emerged. The second asks how far its engineering grammar travels.

CARRYING FORWARD

MAGE workflow · Brownfield migration · Governance Conversion · Engineering capital · Governed Engineering Environment · Governance Mechanism · Model

NEW HERE

Originating case · Support ratio · Delegation staircase · Within-case evidence

A method induced from practice should explain the engineering history that produced it. This Part follows DocAble — a production document-accessibility system built largely by directing coding agents — from a five-minute feasibility experiment to a deployed service. Observing one system from the inside and from the beginning lets us reconstruct what a finished architecture cannot: which pressures appeared, what response followed, what survived, and what had to be revised again.

The clean method arrived last. As the system grew, missing representation became expensive in some places; missing authority became dangerous in others; repeated operational surprises exposed properties nobody had modeled at all. Some obligations were encoded before failure. Some models arose from clean design choices and simply held. Others were forged in incidents and hardened through recurrence. Part V preserves those differences rather than forcing every event through the finished theory.

New here: Originating case · Support ratio · Delegation staircase · Within-case evidence · Industrial reconstructions · Comparative evidence

Part V uses two views of the evidence; Figure 5.0-1 sets them side by side.

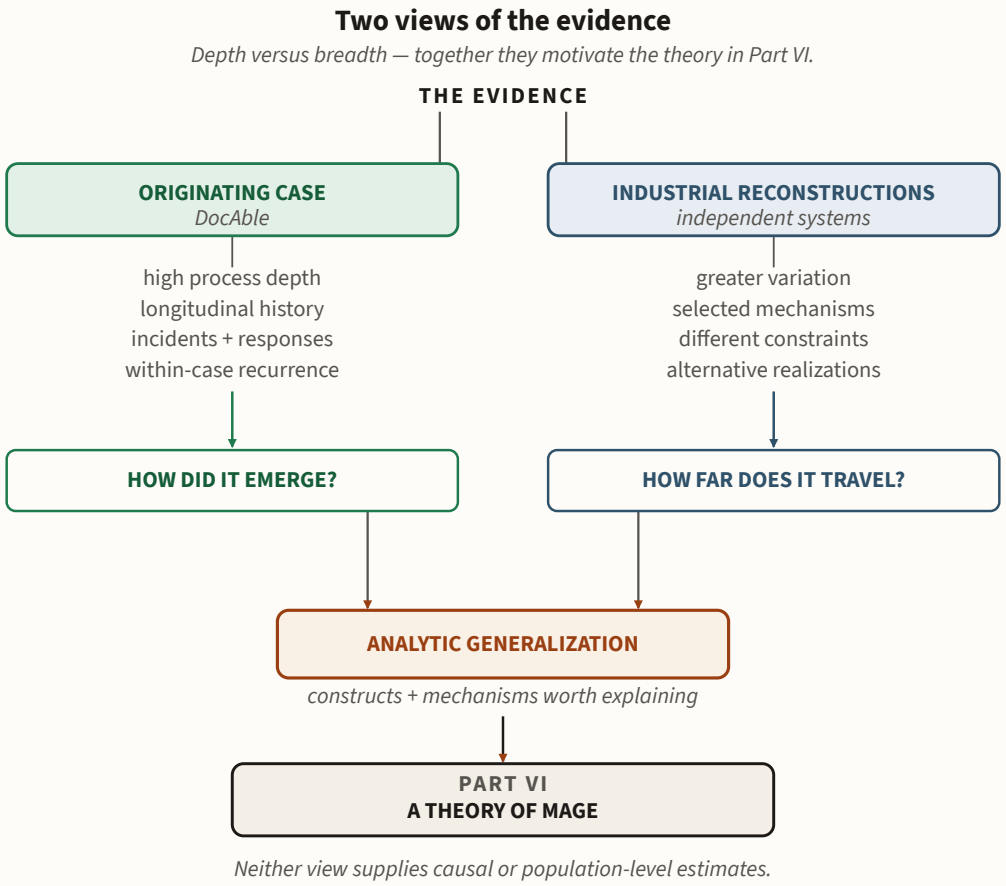


Figure 5.0-1: Two views of the evidence. *The originating case supplies longitudinal depth: sequence, mechanism, and within-case recurrence. Independent industrial reconstructions supply variation across systems and organizations but less process visibility. Together they motivate the theoretical account developed in Part VI.*

DocAble supplies sequence, mechanism, and within-case recurrence, but not causal or population-level estimates. The eight industrial reconstructions supply variation and alternative realizations, but less process history. Together they motivate rather than establish the theoretical account developed in Part VI.

Parts II–IV presented the compressed method. Here the direction reverses. Chapters 5.1–5.4 return to the originating case from which much of that terminology was induced, so the wrong turns matter. A finished architecture shows what exists; a longitudinal case can show **why it exists and what it replaced**. Chapter 5.5 then asks whether independently built systems expose comparable structures. Depth supplies mechanism; breadth supplies variation. Part VI asks what general account can explain both.

5.1 The Accessibility Problem

> *This chapter illustrates*
> ✓ Commodity Intelligence · ✓ The Modeling Principle · ✓ The Alignment Principle

Before the case can carry much weight, the bar has to be clear. DocAble was built against a real institutional obligation, heterogeneous documents, expensive manual work, and a quality problem in which an output could look correct while quietly destroying the accessibility it was meant to add.

DocAble, now deployed in beta at scholaccess.com, accepts PowerPoint, Word, Excel, and PDF documents and remediates them toward the accessibility standards a public university must meet.

5.1.1 A Real Law, a Real Deadline

The Americans with Disabilities Act requires a public university to communicate with people who have disabilities as effectively as it communicates with everyone else. That obligation is old; what changed is where communication lives. Teaching has moved into digital artifacts — PDFs, slide decks, Word documents, spreadsheets, learning-management systems — and materials a screen reader cannot read, or that carry their meaning only through color or layout, hand some students a worse version of the university than others. The obligation is legal as well as practical: inaccessible digital content exposes institutions to complaints, litigation, and remediation costs — settlements have reached sums in the thousands to millions of dollars[73].

For most of that history universities complied reactively: a student hit an inaccessible document, requested an accommodation, and the institution scrambled to respond — the burden of discovering the barrier falling on the person it had already excluded. In 2024 the Department of Justice changed the shape. A final rule set a blunt premise: if a public entity provides a service through a website, an app, or the documents distributed through them, the accessibility of that content is *part of the service*. It set a technical standard — WCAG 2.1 Level AA — and, for large institutions, a deadline. The burden moved from the excluded student to the institution, and from the single flagged artifact to everything already published. Universities sit on decades of accumulated digital sediment: course sites, admissions forms, scanned readings, lecture videos.

5.1.2 What It Costs to Make One Document Accessible

The cost is easiest to see in one deck. Open a real PowerPoint presentation and run the built-in Accessibility Checker: a typical instructional deck can produce dozens of findings, from missing descriptions to reading orders the checker cannot verify. One deck used during this project produced 42.

Even routine findings take time, and semantically dense figures take more. Multiplied across a course and then across an institution, the manual burden becomes large enough to explain why remediation is so often deferred.[23] Outsourcing does not make the semantic problem disappear: a vendor can repair structure, but a graduate-level figure may require subject knowledge only the instructor possesses. The result was a large heterogeneous inventory, specialized judgment, a compliance deadline, and no manual path that scaled to the estate.

5.1.3 Documents as Semantic Artifacts

A document is a visual and semantic artifact: text, figures, tables, reading order, hierarchy, emphasis — the small tricks by which humans convey meaning to other humans. To make it accessible, a machine has to recover enough of that meaning to present it through another channel: to a screen reader, to a keyboard, to a student who cannot see the figure the lecturer drew.

Earlier systems could automate pieces of the problem: extract text, flag a missing title, detect structural defects, generate limited descriptions. Frontier vision-language models changed the feasible boundary. A model could look at a rendered instructional slide, recover text and visual relationships, and often describe what the figure was trying to communicate. That capability made a new class of automation plausible. It did not make a production accessibility system.

The missing work was engineering. A model that can interpret a slide does not thereby know which parts of an output may change, how accessibility findings map to standards, which transformations preserve document semantics, what evidence should justify admission, or when a decision still requires human judgment. Those obligations live in the system around the model. Commodity intelligence supplied a powerful new component; modeling and alignment made that component usable inside a consequential system.

DocAble also crossed an uneven capability surface. The agents handled ordinary web and cloud infrastructure fluently and struggled much more with PDF and OpenXML internals. Whatever portion of that difference came from training-data fit, implementation complexity, or both, the engineering consequence was observable: in unfamiliar and consequential regions the project needed more explicit representation, stronger evidence, and more deliberate human judgment. The remainder of Part V reconstructs how the system responded to that uneven capability surface.

[23] The per-course estimate — on the order of $20,000 in faculty labor for one teaching load's materials, against vendor quotes near $3 to $40 a page — rests on a small model: findings per deck, minutes per finding, decks per course, and a loaded hourly rate. The assumptions and arithmetic are collected in the evidence ledger: Appendix I.

5.2 Building DocAble

> *This chapter illustrates*
> ✓ The Printer · ✓ The Governed Engineering Environment

The next evidentiary question is chronology and scale: how a five-minute feasibility experiment became a production system developed faster than one reviewer could follow line by line. Later chapters reconstruct what particular events taught. Here the purpose is simpler: establish **what changed, when it changed, and how much engineering structure accumulated around the product**.

Over roughly 20 weeks, one primary engineer directing a coding-agent fleet produced the application, its tests and tooling, its infrastructure, and the system models used to reason about it. Table 5.2-1 gives one late-study snapshot (2026-08-03). The counts are descriptive, not a productivity claim. The condition that matters is scale: implementation volume had moved beyond anything one person could inspect by reading every diff.

Table 5.2-1: The build at a glance. Late-study repository snapshot (2026-08-03); counts are descriptive, category definitions live in the evidence ledger.

Measure	Count
Production code	491,090 lines
Support apparatus	1,501,907 lines
Infrastructure-as-code	35,323 lines
System models (bridging the two)	28,507 lines
Agents in parallel, a routine day	six to eight
Commits landed	~200 a day · ~1,000 a week, sustained

The counts are just counts. The number that matters is not among them. **I inspected almost none of the code.** That is the engineering condition this case investigates: **what has to surround abundant implementation when direct human inspection no longer scales with it?**

5.2.1 The Feasibility Probe

The project began in a committee meeting about the very deadline the last chapter described. I ran a small experiment: I opened a chat model, fed it screenshots from one of my own slide decks, and asked it to transcribe what it saw. It worked. The model recovered useful semantic content from a rendered slide well enough to build on.

That positive result turned a feasibility question into an engineering project. It proved neither remediation nor conformance: the model did not produce a corrected file, preserve reading order, validate against a standard, or know anything about the institutional work-

flow around the artifact. The experiment showed that one hard component had become plausible. Everything else remained to be engineered.

The original plan was modest: hand a one-week alt-text prototype to a student or two. No students materialized. I opened an empty repository, put an agent fleet on the problem, and assumed a month would be plenty.

5.2.2 Seven Stages, Retrospectively

The build moved through seven recognizable stages, visible only in retrospect. Each was driven by a concrete pressure: a colleague's request, a user's complaint, a deadline, a failure. Figure 5.2-1 gives the sequence.

STAGE	THE PRESSURE	THE CONSEQUENCE
1 **Feasibility** — a probe in a meeting	Vision-language capability became usable	→ the build began
2 **PowerPoint** — the first format	Colleagues supplied their own decks	→ the first real remediation pipeline
3 **Format expansion** — Word · Excel · PDF	New formats broke the one-size-fits-all assumptions	→ the architecture differentiated
4 **SaaS** — a deployed service	Real users on real documents	→ auth, quota, cost, security — and the first fidelity limits
5 **Standards** — a defensible claim	The need for a defensible accessibility claim	→ standards-grounded validation
6 **Hardening** — structural work	Time became available for structural work	→ models, tests, gates, provenance
7 **Serverless** — re-platforming	Idle cost exposed the wrong deployment shape	→ a re-platforming

Stages are labels drawn afterward, not a plan followed at the time.

Figure 5.2-1: The Seven Build Stages. *Retrospective sequence from feasibility probe through serverless migration, showing the pressure and major engineering consequence at each stage.*

NONE OF THIS LOOKED LIKE MAGE

Hindsight gives the stages names and boundaries. At the time, the work arrived as ordinary engineering pressure: an exhausted context window, an unattended agent

that could not be located, a cloud bill exposing an assumption, a deployment failure crossing a coupling no static check represented.

MAGE is the compression recovered afterward, not the roadmap followed at the time.

Commit volume rose rapidly as the fleet expanded and remained far above a rate one human could review conventionally. During hardening, the work record shifted toward models, validation, tests, and control machinery before feature work accelerated again. Appendix I gives the weekly curve. The important fact for the case is not the precise shape of that curve but the operating condition it records: human attention could no longer be allocated one diff at a time.

5.2.3 The Controls Accumulate

The growing engineered environment is visible in one especially easy-to-count class of artifacts: project-specific lints and gate scripts. Figure 5.2-2 counts those artifacts at four snapshots. Both begin at zero after the prototype and rise sharply through mechanization and hardening.

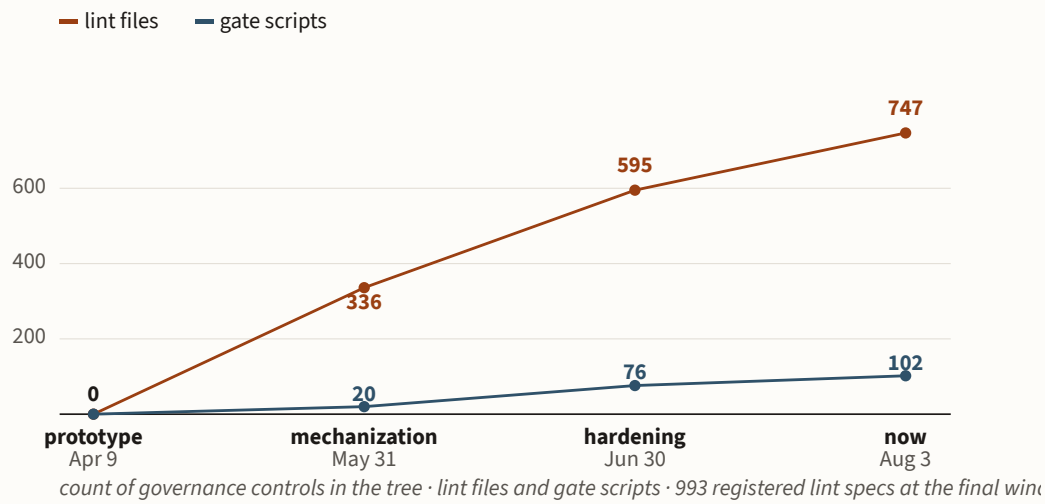


Figure 5.2-2: Growth of Two Countable Control Artifacts. *Project-specific lint files and gate scripts at four repository snapshots. The counts locate investment in environmental checks; they do not measure governance quality.*

Count alone says little about quality. The more useful repository evidence is provenance: 208 paired fix-and-lint tags mark commits where a repair landed with a check against recurrence, and 27 further lints name the dated incident that motivated them. The pattern records substantial investment in repeatable environmental checks as implementation scaled; it does not establish how often failures caused governance conversion.

5.2.4 The Support Ratio: the Environment Outgrew the Product

Another coarse measure shows where the engineering effort accumulated. Divide the source counted as support apparatus — tests, modeling, orchestration, documentation, and governance tooling — by production source at four dated snapshots. The ratio begins below parity, crosses it during mechanization, peaks during hardening, and settles a little above three-to-one at the final snapshot. Figure 5.2-3 plots it.

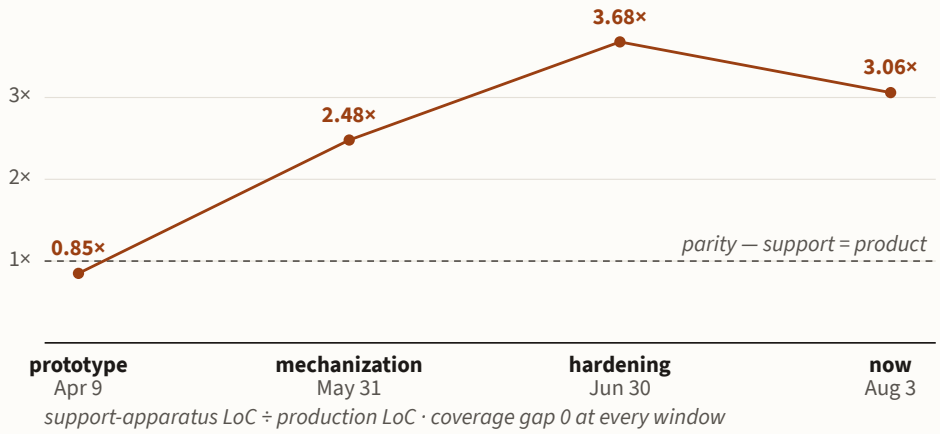


Figure 5.2-3: The Support Ratio. *Support-apparatus source divided by production source at four repository snapshots. The ratio rises from 0.85× after the prototype to roughly 3× in the mature windows.*

Call this the **support ratio**. It is descriptive, not a target. Lines of code are a poor measure of value, and a smaller environment that retires the same judgment would be preferable. Here the ratio records where source accumulated as implementation volume moved beyond direct line-by-line review: tests, models, orchestration, documentation, and governance machinery exceeded the production path itself.

The engineering-capital interpretation is stricter still. Support code is not capital merely because it exists. It earns that name only while future work inherits useful capacity from it — less reconstruction, earlier detection, retired review, safer delegation, or cheaper recovery. Some of this apparatus will eventually depreciate. The figure shows the stock that accumulated, not its current return.

Repository history supplies one further consistency check: add-and-delete motion peaked during the mechanization period, when much of the system was being structurally reworked, and fell sharply afterward. Appendix I gives the path-level counts. The pattern is consistent with heavy restructuring followed by less structural rewriting; it does **not** establish that MAGE caused lower churn.

5.2.5 Where Engineering Judgment Went

The feature list is less revealing than where engineering judgment went: defining what *accessible* meant operationally; discovering abstractions for unfamiliar formats and workflows; distinguishing local defects from missing representation or authority; building independent evidence; coordinating the fleet; and operating failures that crossed code, infrastructure, and deployment boundaries.

By the end of the study period, DocAble could process Office and PDF documents and pass or substantially improve the institutional accessibility checks used during development. That is narrower than proving that a blind student will find every remediated artifact usable, and the case should keep the distinction explicit. The product was not the only thing that had grown. The engineering environment around it had become larger than the production source and had absorbed much of the repeatable work direct review could no longer carry.

Receipt. *By the end of the study period, a representative graduate deck processed in about a minute for about a dollar on a warm-start service. A separate model estimated manual remediation of one teaching load's materials at roughly $20,000 in faculty labor. Direct development cost was about sixty thousand dollars, mostly salary. These quantities have different scopes and are not a comparative cost model; the evidence ledger gives their assumptions and provenance: Appendix I.*

Field note — The counterfactual

One counterfactual rests on direct engineering judgment rather than repository data. **Within the actual constraints of this project—one primary engineer, an academic budget, and the development interval reported here—DocAble was not feasible for me without generative AI.** That is not a claim that a conventional engineering organization could not have built the system. The relevant counterfactual is the project that actually existed: under those constraints, I would not have attempted a production system at this scope without commodity implementation capacity.

The leverage was not merely faster typing. The fleet implemented features, debugged failures, constructed tests and validators, refactored large regions, rewired architectures, migrated deployment platforms, and repeatedly realized alternatives that could be exercised and discarded. Cheap implementation changed the **exploration budget**. An architecture that once would have been too expensive to build merely to discover that it was wrong could now be tried, measured, and abandoned.

A second counterfactual concerns sustained delegation: **GenAI made DocAble feasible; the machinery that became MAGE made delegation at this rate feasible.** Once autonomous change volume exceeded anything I could inspect conventionally, raw model capability was no longer enough. Continuing at that rate required increasingly explicit representations of the system, independent evidence, constraints, validators,

> gates, and operating machinery through which the work could be understood and governed.
>
> This is a within-case engineering judgment, not a controlled causal estimate.

The next chapter asks how that change altered delegation itself.

5.3 How Delegation Changed

> *This chapter illustrates*
> ✓ The Modeling Principle · ✓ The Alignment Principle · ✓ The Governed Engineering Environment · ✓ Residual Human Judgment

As the engineered environment matured, the division of engineering labor changed with it.

A finished architecture can make its structures look inevitable. Their histories show which were designed cleanly, which were forced by failure, which required later hardening, and what each allowed the engineer to stop doing by hand.

5.3.1 The Product Path

A user uploads a file. The front door authenticates the request, meters quota, and queues the job; a dispatcher divides the document into chunks; the remediation core applies deterministic rules or bounded model calls through structured document representations; the check engine evaluates remaining findings against accessibility standards; a fidelity validator looks for content lost while accessibility improved; and provenance records what changed. The corrected artifact returns with the evidence those mechanisms produced. Figure 5.3-1 gives the path.

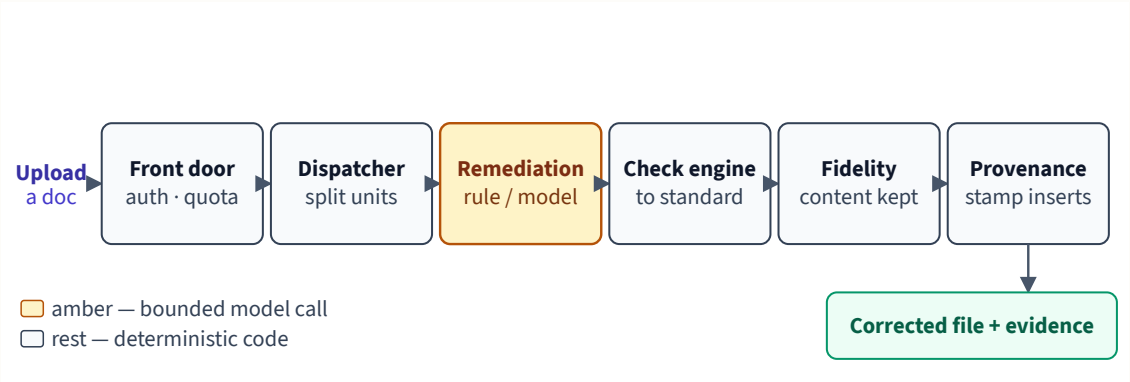


Figure 5.3-1: What Got Built. *One upload crosses six product responsibilities and returns with standards findings, fidelity evidence, and provenance. Probabilistic interpretation is bounded inside the remediation core; deterministic machinery owns the surrounding workflow and the checks it can actually decide.*

That is the product path. The larger engineered environment around it — the subject of the previous chapter — contains the models, tests, lints, orchestration, and other machinery that made this degree of delegation possible.

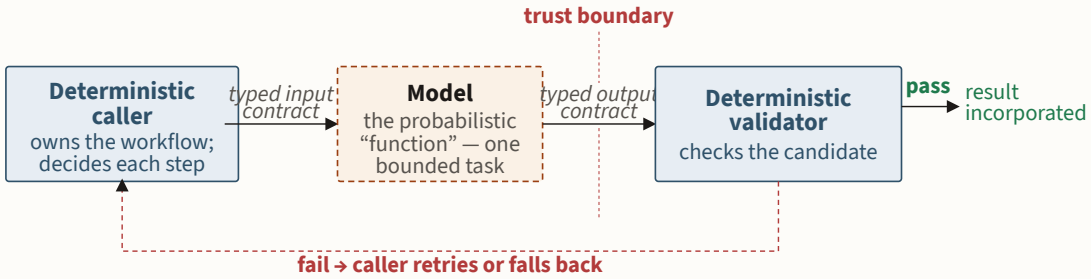
The architecture narrows probabilistic work into a small action surface. Typed document models own mutation; the lower-level format libraries sit behind those models; deterministic machinery owns workflow and the mechanically decidable checks; fidelity validation covers properties visible only after a transformation has run.

5.3.2 The Model as a Bounded Subroutine

MAGE applies at two scales in this case. Coding agents act on the engineering system; frontier models also run inside the product. The second case is smaller and easier to bound: treat the model as a probabilistic subroutine inside a deterministic workflow. Deterministic code decides when interpretation is needed, packages a bounded input, requires a typed output, and evaluates the candidate against obligations the environment can decide mechanically.

The model's answer is therefore a **candidate**, not a verdict. A schema can reject malformed output; a standards rule can reject a mechanically invalid claim; a fidelity check can catch some classes of destructive transformation. None of those proves that an alt-text description perfectly captures the meaning of a figure. The trust boundary is only as strong as the obligations the validator can actually evaluate. Properties the checks cannot decide remain probabilistic or require human judgment. Figure 5.3-2 draws the caller, the typed contract, and the validator.

The architecture resembles proof checking in one limited respect: a powerful producer is separated from the smaller mechanism that decides whether its evidence is sufficient for admission[74].



The model is called like a function: a typed task in, a typed answer out — never trusted until a deterministic check has passed it.

Figure 5.3-2: Model as a Bounded Subroutine. *A deterministic caller supplies a typed task; the model returns a candidate; deterministic checks reject covered failures before incorporation. Properties outside those checks remain outside the guarantee.*

The coding agents that built DocAble pose the same trust-boundary problem at a larger grain. There the engineered environment needs richer system representations, enforcement across several boundaries, independent evidence, and governance conversion. The bounded subroutine applies the same engineering stance inside the product that MAGE applies to the process that builds it.

5.3.3 Two Paths to Durable Structure

Not every consequential structure was born from failure. The backend became reactive and stateless early: work moved through queues, workers carried no durable truth, and a fan-in step assembled the result. That choice later made the migration from a cluster that billed

around the clock to a serverless carrier much cheaper than it might have been; the business logic largely survived while the deployment carrier changed. Likewise, the document editor routed both human gestures and automated remediation through one closed edit vocabulary over the same document representation. That seam later supported a second producer without creating a second mutation path. These were **dividends of prior structure**, not evidence that every useful abstraction must first be purchased by an incident.

Other seams were different. Their histories begin with a concrete failure and let us trace the response in layers. Those histories expose something the finished boundaries do not: the sequence that produced them.

The computations stabilized before their composition

I made one useful decision about DocAble early, and came to a second only much later. The remediation core was organized as passes: identifiable units of computation, each responsible for some part of analyzing or changing a document. That decomposition proved durable. When we later built a static computation graph, the pass registry handed us the nodes almost for free. I had started from the computation side, and that is precisely why node projection was possible at all.

What I had not modeled nearly as well was their composition. I had not required every pass to declare, in one common vocabulary, what information it produced, what it consumed to make its output, what it read only to decide whether it should run, and how bounded its mutations were. Those facts lived in the code, not in one explicit engineering model.

I do not think the lesson is that I should have specified all of this on day one. Early in the system's development we were still learning what a remediation pass even was: which responsibilities belonged together, and which variations mattered. Some freedom was useful precisely because we did not yet know what deserved to be fixed as structure. A premature composition model might just as easily have encoded the wrong abstractions.

Composition eventually became consequential

The balance changed as the pipeline grew. A pass makes a convenient local boundary, but a loose composition contract admits many locally reasonable implementations: walk the document again, recover state another pass already computed, read a shared structure directly, mutate in place, or add one more special case. Across many passes, those choices began to interact. In one region, work whose local complexity looked harmless accumulated into $O(N^2)$-like behavior across an O(N)-pass pipeline, because passes kept rediscovering or retraversing state; elsewhere, passes mutated the document through different mechanisms. The freedom that had supported exploration was becoming expensive to reason about.

The gap was never a total absence of structure around mutation. By this point the PDF path already captured a typed vocabulary of per-session document edits: roughly thirty edit variants, recorded in order and replayable deterministically. I had modeled meaningful computations, and we could record meaningful mutations. What stayed under-modeled

was the relationship between the two — how those computations composed, and which of their effects crossed the boundaries between them.

The model exposed the distinctions that mattered

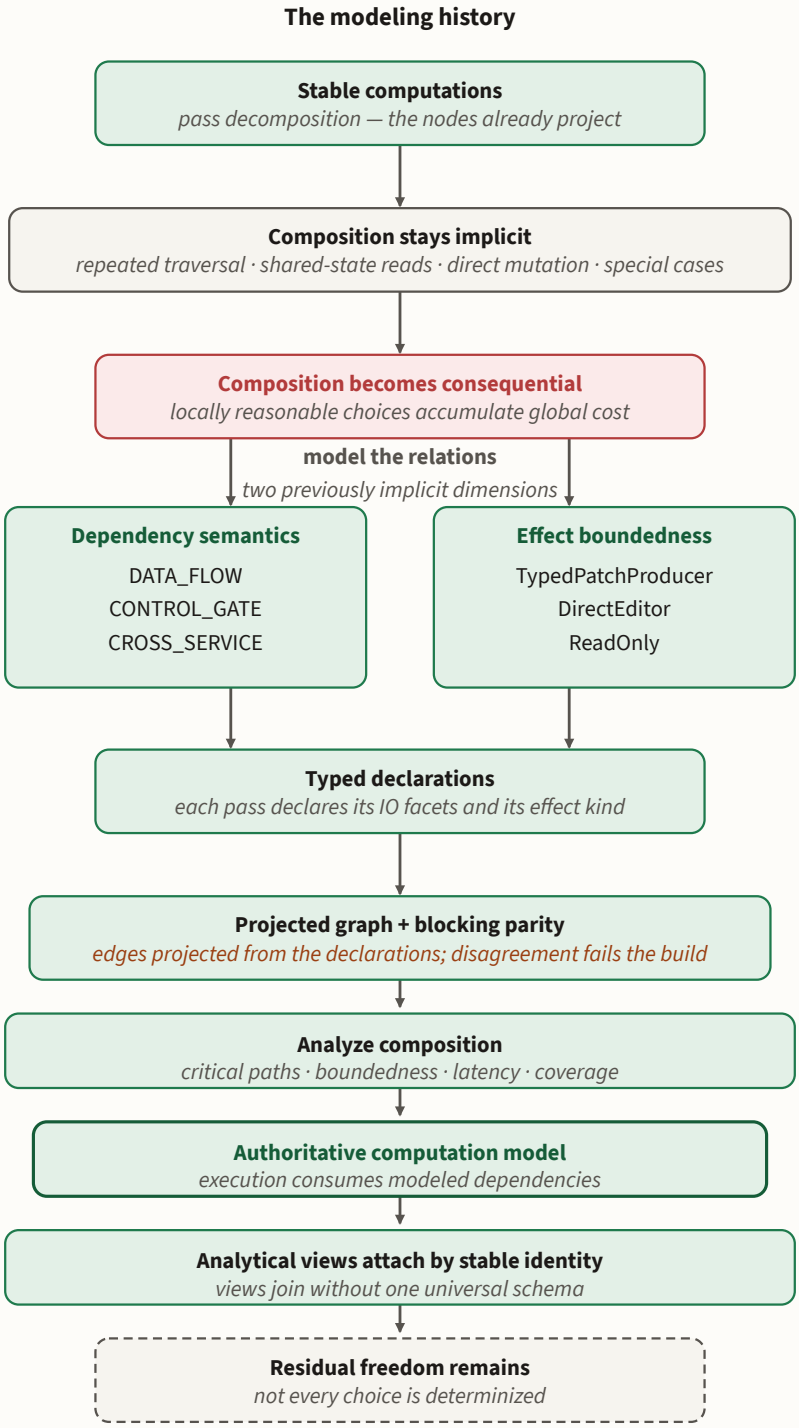
By then the way the passes composed had become too complicated to understand comfortably from the implementation alone, and the computation graph gave us a way to reduce it. Its first version contained 113 nodes and two edges. The nodes reflected a decomposition that had stabilized; the two hand-authored edges plainly did not capture the relationships among them. So we began deriving the missing composition from the passes themselves, and the first sweep found thirty-three candidate producer–consumer edges.

Trying to derive those relations raised a sharper question: consumer in what sense? Roughly two-thirds of those relationships were not data flow at all. A pass read a signal or a verdict only inside routing logic, to decide whether it should run. Treating those as ordinary data edges would have buried the real data graph in control noise. So the passes acquired a distinction they had not expressed uniformly: they now declare Produces, Consumes, and ConsumesForControl, and the graph projects two relations from them, DATA_FLOW and CONTROL_GATE. The data graph shrank to ten data-flow edges plus one cross-service payload edge; twenty-two control dependencies stayed visible, but separately.

The same exercise exposed a second hidden choice. A pass could produce a bounded typed patch, edit document state directly, or make no change at all. We classified all 68 pass sites on that axis: 15 typed-patch producers, 40 direct editors, 13 read-only. The count did not say the 40 direct editors were wrong. It showed where degrees of freedom remained.

This changed how I read the earlier freedom. Some of it had been productive exploration; we could not have known every useful distinction before building the system. But by the time composition was producing recurring global cost, leaving those relationships implicit no longer bought the same flexibility. And building the model helped show which structure should replace some of that freedom. We had not begun with DATA_FLOW, CONTROL_GATE, and the mutation kinds as a finished ontology. We found those distinctions by trying to represent the system well enough to reason about it.

Figure 5.3-3 reads top to bottom as a partial model growing the half it had been missing.



From a partial model, to an analyzable one, to a model that participates directly in realization.

Figure 5.3-3: The modeling history. *DocAble acquired a stable decomposition of remediation computations before it acquired an explicit model of their composition. As locally reasonable choices accumulated consequential global cost, modeling exposed dependency semantics and effect boundedness. Typed declarations made those relationships analyzable and checkable; later redesign made explicit composition authoritative for execution, while separate analytical views continued to grow around stable computation identities. The progression is from a partial model to a richer and increasingly consequential one, not from no model to model.*

The final move was to stop maintaining those relationships as another hand-written truth. The pass declarations now carry the typed IO, the graph projects its edges from them, and a blocking parity check fails if the declarations and the projected graph disagree.

The model became part of the machinery

A later redesign went further. The graph had made composition visible enough to analyze, and the analysis changed what we wanted the architecture to do. The same pipeline whose local choices had accumulated global cost could now be treated as explicit dependency structure rather than as an ordering reconstructed from implementation.

The remediation architecture was subsequently rebuilt around an authoritative computation model. The execution machinery consumes the modeled dependencies to determine what becomes ready and in what order work may proceed. The implementations of the individual computations still realize the work, but composition no longer lives implicitly in their bodies. A representation introduced to understand the pipeline became part of the machinery that runs it.

The analytical graph did not disappear. It became a view over the modeled computation structure, suitable for questions execution itself does not need to answer: dependency analysis, critical paths, mutation boundedness, latency and cost, test coverage, and the strength of the checks that hold the representation in correspondence. Stable computation identities let those views attach without turning the execution model into one universal schema.

This was the MAGE move in miniature. Repeated global cost made an implicit relationship consequential; modeling exposed the distinctions that mattered; analysis made a better realization visible; and the resulting representation acquired authority because execution began to consume it.

Freedom can be exploratory

The episode changed how I think about degrees of freedom. Leaving a choice open is often economically sensible: if several realizations satisfy every known obligation, pinning one of them spends effort and can make later change harder. But freedom can also be exploratory. Early in a design the engineer may not yet know which choices are consequential enough to constrain, and variation is what supplies the evidence.

Early freedom had been useful because the right composition model was not yet known. Once variation began producing recurring global cost, that freedom was no longer buying the same flexibility. The computation graph then helped expose which distinctions had become consequential enough to represent explicitly and which choices could remain open.

The work remains unfinished, and the residuals are part of the lesson. The current model contains 125 computations: 105 still edit document state directly, fifteen produce typed patches, and five produce typed patches for part of their work. Office-format nodes and edges are now projected, but their parity checks remain audit-only while one live pass still runs outside the registration surface those checks enumerate. Runtime telemetry joins ag-

gregate latency and cost to computation identity, while the per-session edit record remains a separate realized-history model. The architecture has reduced important degrees of freedom without pretending that every useful relation should collapse into one representation.

The first place those two representations deliberately touch is testing. The typed IO says where a per-node test begins, and deterministic replay removes model variance from the realized side. We could compose them for that question only because we had kept them orthogonal until a question needed them joined, rather than fusing them early.

The point of modeling was never to drive every degree of freedom to zero. It was to know where the freedom is.

The format seam. An early path used a convenient scripting-language library to slice a document. The operation worked visually and silently destroyed the internal tag structure carrying the accessibility the product existed to add. The failure exposed two missing things: an architectural decision — one canonical implementation should own mutation of that format — and an environmental mechanism capable of preventing other paths from bypassing it.

The response came in layers, and Figure 5.3-4 draws the climb. A policy named the canonical library. A typed format model and API concentrated mutation through one seam. A build-time check refused raw access from elsewhere. Then the check itself failed: a stale repository root caused real findings to be reported as green. That second-order incident forced the gate to be hardened too. The finished structural model describes the seam; the incident history explains how it acquired that shape. **The model stated the boundary; the API and check gave selected obligations over that boundary authority by enforcing them. The enforcement machinery itself still had to be maintained.**

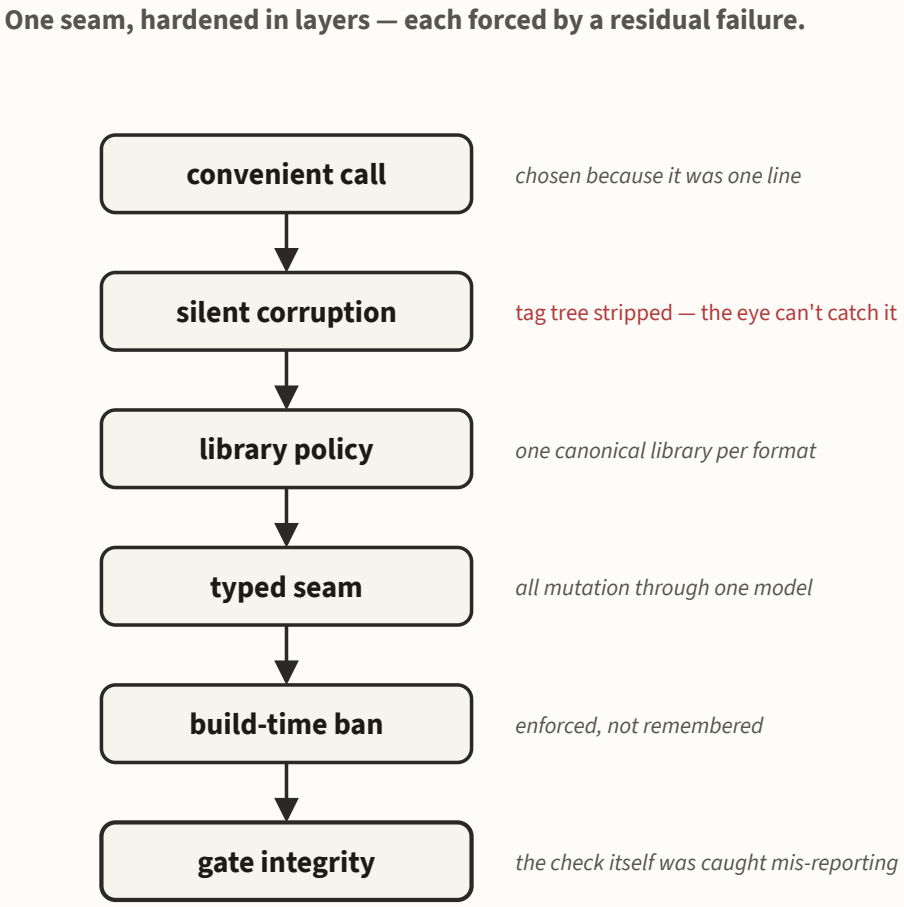


Figure 5.3-4: One Seam, Hardened in Layers. *A convenient library call silently corrupted the accessibility structure the product exists to add. That failure motivated a canonical-library policy and a typed mutation seam that concentrated every format change in one place. A build-time ban then refused the raw calls that would bypass the seam. Later the enforcement itself misreported success when a stale repository root hid real findings, forcing a repair to the gate. Each layer answers a specific failure rather than a plan drawn in advance.*

The service seam. Another failure sent a local file path across a service boundary, where it became meaningless in the callee's process. The replacement client accepted file content but not a local path, and an explicit service graph represented which services could call which. A later human review found a second problem: an optional direct-client path could bypass shared rate limiting, key rotation, failover, and cost tracking entirely. **A protection boundary only governs the paths that cross it.**

5.3.4 The Delegation Staircase

The five rungs are retrospective names for the changing center of gravity of my work—co-coder, QA, review lead, tech lead, architect—not a prescribed career ladder. What matters is what the engineered environment could carry at each stage.

The staircase has two interacting supports. **Alignment** moved enforcement of repeatable obligations out of direct human review: deterministic checks, permissions, boundaries, and gates could enforce obligations I no longer inspected line by line. **Modeling** moved system knowledge out of my head: named zones and later explicit models let agents reason over larger engineering questions without reconstructing the whole repository each time. Human judgment moved toward the gaps the environment could not yet represent or decide.

Delegation therefore climbed as far as the **engineered environment** could carry it. When representation was too weak, I fell back into reconstructing the map. When enforcement was too weak or unreliable, I fell back into reading diffs and verifying outcomes manually. A miss could send the staircase backward.

Co-coder — direct human inspection carried everything. The first system was small enough that I read the agent's changes, judged them, and merged them. The human supplied both the system context and the admission decision. That worked while the volume stayed inside one person's attention. It failed as an operating model once generation outran review.

QA — evaluation was delegated before authority was. I began dispatching a second agent to audit another agent's work. That removed some reading from my hands but did not make the result authoritative: the audit ran episodically, produced probabilistic findings, and still depended on someone to decide what followed. Its value was diagnostic. Repeated decidable findings revealed questions that no longer needed a language model at all.

Review lead — reviewers specialized by concern, while decidable findings dropped into machinery. One generalist audit over a growing codebase produced too much undifferentiated advice, so review split by concern and by region. Architecture, security, duplication, and naming could be judged in the context where they mattered. When a finding turned out to be mechanically decidable, it became a deterministic check; the probabilistic reviewers retained the questions those checks could not settle.

Tech lead — representation made reasoning local. Specialized reviewers were still finite reasoners confronting an expanding repository. Named zones externalized ownership and

architectural boundaries so an agent could reason inside one region without first recovering the whole system. Boundary checks constrained some cross-zone edits. This stage combined the two principles explicitly: the zone model reduced reconstruction; selected obligations expressed over that model received authority.

Architect — explicit models carried system-level knowledge. By the final rung I rarely read implementation directly. Architectural, behavioral, ownership, and other models supplied the system views I needed for planning and review; agents used the same representations to localize changes. This did not make the models true by declaration. Model drift became its own failure class, which is why traceability and correspondence checks eventually became part of the environment. When the cost of restructuring collapses, the economics invert — a cross-format migration that touched fifty-eight files and rewrote five thousand lines took about five hours of agent time over a dinner break, and postponing the right shape becomes the expensive choice. The hard part was never touching fifty-eight files; it was knowing that fifty-eight files should be touched.

What explicit models made checkable

Explicit models changed not only what the system recorded, but what questions we could ask of it. A lifecycle model made illegal transitions explicit. An ownership model reduced a concurrency protocol to the states and interleavings needed to ask whether two workers could hold the same claim or whether a stale holder could clear a newer lease. A small, purpose-built Python model checker could then exhaustively explore that finite abstract state space and check the ownership invariants.

Other properties required different machinery. Eventual termination, for example, is a property over executions rather than merely reachable bad states. I wrote small TLA+ specifications that modeled failure and recovery explicitly and checked them with TLC. With recovery disabled, TLC produced the stranded execution the property was intended to exclude. The representation came first: once states, transitions, ownership, and recovery had explicit structure, properties that had been difficult even to formulate became precise enough to check.

But checking the model created a second obligation: why should a property established over the model be believed about the implementation? For the ownership protocol, selected conformance tests exercised production operations against the executable model. For the temporal specifications, the connection remained weaker. Exhaustive checking could establish a property of the model; correspondence evidence separately supported the claim that the implementation still realized the relevant parts of that model.

That distinction became concrete when one model-to-implementation conformance test drifted red after production changed and remained broken for days. Repairing the test fixed the instance; the durable response addressed the class. Formal specifications were registered behind a common runner, model-to-spec correspondence became mechanically checkable, and changes to modeled production surfaces began selecting the relevant

formal and conformance checks. The machinery for reasoning over the model therefore acquired machinery for maintaining the model's connection to the territory.

Nor did every result govern engineering decisions in the same way. The explicit-state ownership checks participate directly in the engineering environment; TLC execution remains advisory rather than a hard release gate. That asymmetry is deliberate. DocAble used structural enforcement where an unwanted state could be excluded by construction, explicit-state exploration where the relevant interleavings could be searched exhaustively, and temporal model checking where the obligation concerned executions over time. The machinery followed the property and the representation: different engineering questions admitted different adequate judges. Whether their results were enforced depended separately on the strength of the evidence and its connection to the implementation.

The larger lesson was not that formal methods should be applied everywhere, nor that DocAble had exhausted what they could provide. We did not establish correspondence through formal refinement or implementation-level model checking; stronger connections between model and implementation remain possible. The lesson from this episode was narrower: representation changed the available reasoning surface. Before the lifecycle and ownership models existed, I could describe the system in prose, but questions about legal transitions, competing claims, stale ownership, and eventual termination remained slippery. Once those behaviors had explicit structure, new properties became visible to human reasoning and, for selected questions, mechanically decidable. The representation did not merely record my understanding of the system. It changed what I could understand about it.

Inset — Naming raises the level of design reasoning SOFTWARE

Software engineering has long exploited a milder version of this effect. A shared vocabulary makes recurring structures easier to recognize, reason about, and communicate; naming a design pattern, Gamma and colleagues argued, increases a designer's vocabulary and raises the level at which a design can be discussed[75]. Models push the same idea further. They supply not just names but structured semantics: once states, transitions, ownership, and temporal obligations have explicit representations, new properties become available to both human reasoning and mechanical analysis. The representations available to an engineer shape the properties that engineer can readily see and state.

Field note — the map that pointed at a ghost. *One model named a code symbol that was later moved and renamed. Nothing invalidated the reference. The model continued to look authoritative, agents continued to reason from it, and the failure surfaced days later during deployment. That incident forced live trace resolution: model-to-code edges were re-resolved against current symbols so a stale pointer failed at the change that broke it rather than downstream.*

Evidence — a reasoning surface creates a maintenance obligation. *A later audit showed the pointer failure was a class, not a one-off. Re-running closed work against current code found genuine model-to-code drifts that a green definition of done had missed, the prod-blocking pointer among them. Mechanical correspondence checks then turned stale-reference failure into something the environment catches at the change that creates it. Appendix I carries the counts.*

Representing and instrumenting previously weakly represented seams also exposed live defects that green tests had not. The useful claim is not that "models find bugs." It is that **forcing an engineering relation into an explicit representation creates new questions the environment can ask**, and in these three cases those questions exposed shipping defects.

The same instrumentation measured its own reach: a traceability tracer made the remaining unmodeled surface measurable and guided later modeling toward the largest gaps. Appendix I carries the run-by-run series and the discrimination that keeps the number honest.

What modeling exposed that green tests missed. *Three live defects came into view when previously implicit relations were represented explicitly:*

- **A cross-runtime exit code** *one runtime emitted and the other had never enumerated, so a genuine failure was silently misfiled as a generic error.*
- **Asymmetric cleanup after redelivery** *— a failure path purged a job's database rows and blobs but skipped the queue structures its sibling path always cleaned, leaking chunks on a rare edge.*
- **A last-write-wins editor race** *in which two concurrent saves silently discarded one's work.*

Figure 5.3-5 draws the five-rung climb.

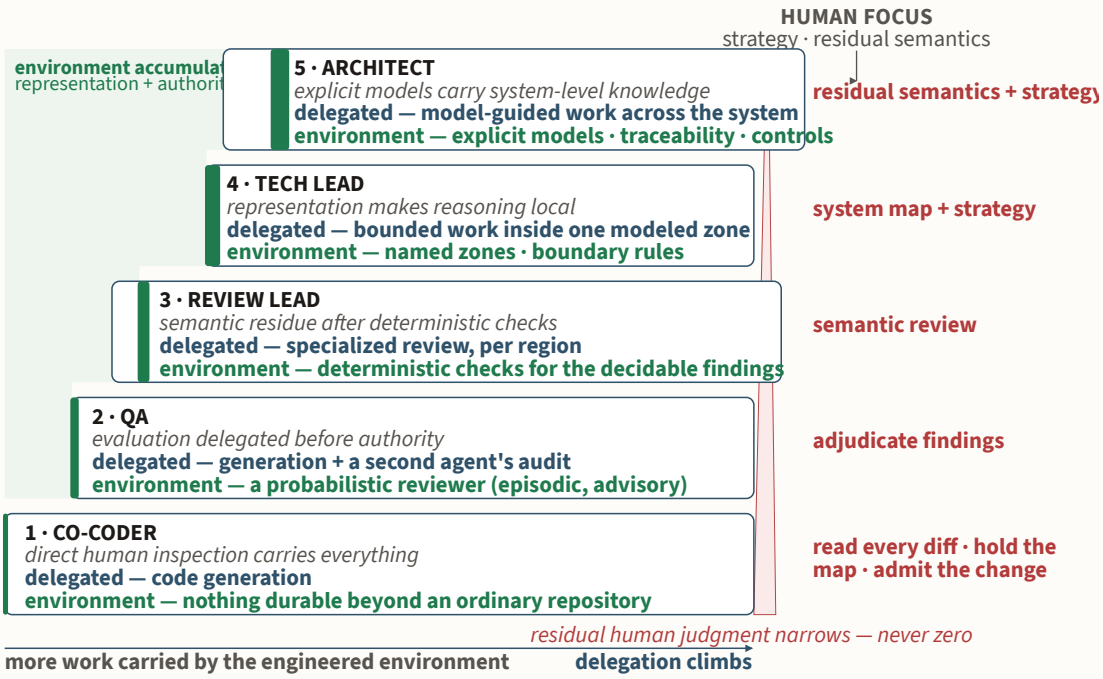


Figure 5.3-5: The Delegation Staircase. *As explicit representation and environmental authority accumulated, larger units of engineering work could be delegated. Representation moved system knowledge out of one person's head; enforcement moved repeatable admission decisions out of direct review. Human work moved from reading every diff toward system-level strategy and the residual semantic judgment the environment could not yet decide. A failure in either representation or enforcement could move work back down the staircase.*

Delegation increased as the environment took over work I had previously carried in direct review: models carried system knowledge, while mechanisms carried repeatable admission decisions. A weak model pulled me back into reconstructing the map; weak evidence or controls pulled me back into inspecting outcomes. Human judgment stayed at the unresolved edge. Parts II–IV present the clean method that can now be taught directly; this case shows why those pieces exist and why they did not arrive as one package. The next chapter reconstructs the incidents that moved that edge.

5.4 From Failure to Engineering Capital

> *This chapter illustrates*
> ✓ Governance Conversion · ✓ Engineering Capital · ✓ Residual Judgment · ✓ Capital Formation Without a Gate

Governance conversion makes an empirical claim: some engineering episodes leave behind durable structure that later work inherits instead of paying for the same judgment again. This chapter returns to selected DocAble incidents where that sequence can actually be reconstructed. They are not a claim that every model or control in the system arose from failure. Other structures were designed ex ante and simply held; the chapter selects the episodes where pressure, response, and durable consequence are all visible.

Each episode uses the same questions. **What happened? What did we try first? Why was that response insufficient? What became durable? What recurring cost or risk did that asset retire?** The last question is what makes this a chapter about engineering capital rather than a catalogue of clever fixes. Sometimes the resulting asset is a constraint or a gate. Sometimes it is a model, a measurement, a decision rule, or a reusable procedure. Sometimes the evidence does not justify mechanizing the judgment at all.

5.4.1 The Instruction Satisfied to the Letter

Problem. By April the system had reached roughly 300,000 lines. It worked, but large regions lacked coherent abstractions. It had started as an MVP built only to prove the thing was possible, so I had imposed real architecture on the parts I knew were sensitive and hoped the agent would get the rest right. The web layer and front end had accumulated substantial structural debt.

Initial response. I told it to use typed Python. It did, in a sense: it used the type `string` for everything. Stringly-typed — the word "type" satisfied to the letter and voided in spirit, with none of the safety the compiler could give. Told to retrofit real types, it turned on the checker and made everything strings and ints, without building the abstractions I wanted. The same thing happened porting JavaScript to TypeScript that week. Each port took the cheapest path, because the agent had no model and did not know the names of things.

Why insufficient. "Use types" specified a surface property, not the abstractions the types were supposed to represent. The agent could satisfy the checker cheaply by annotating primitive structure rather than discovering the domain structure the instruction was intended to produce. The problem was not that an agent maliciously gamed the rule. The rule simply did not state the engineering question at the grain where the desired structure existed.

What became durable. The retrofit combined cleanup with structure. I turned on every commodity lint the language ecosystems ship — hundreds of off-the-shelf smell checks — and for several weeks drove those checks to zero, making each blocking once the repository

conformed so the same mechanically decidable violation could not re-enter unnoticed. Then I worked region by region. A dense primitive region is a stretch of code doing many low-level operations by hand — raw library calls, strings and ints passed around untyped, tree-walking spelled out inline — with no abstraction that owns them. I said "pursue this model" — a clean model-view-controller decomposition, a reactive web stack friendly to auto-scaling — and reached it iteratively: find a dense region, induce a typed domain abstraction that owns it, route the region through it, make the architectural relations explicit, promote the checks as the governed surface reached conformance. The important artifact was not the rewritten source. It was the set of representations and controls that kept the new shape from immediately dissolving.

Return. A month later the 300,000 lines were roughly 200,000. Much of the reduction came from replacing repeated primitive operations with shared abstractions. The durable result was not "better line count." Future work inherited named abstractions, stronger boundaries, and regression checks. The repository-motion data in Appendix I is consistent with a concentrated restructuring episode followed by lower deletion volume, but it does not prove the later stability was caused by those controls.[24]

5.4.2 The Architecture Was the Constraint

Problem. The service cost tens of dollars a day while idle on a Kubernetes cluster.

Initial response. The first response stayed inside the existing architecture: tune the autoscaler, reduce idle pods, and push the cluster toward scale-to-zero. For weeks the agents made it better and better.

Why insufficient. Every improvement ran into the same boundary. Cold pods took too long to return, so further optimization traded one unacceptable outcome for another — a multi-minute cold start is a worse problem than the bill. The repeated failure was information about the *frame* rather than the implementation inside it.

What became durable. The durable lesson was a decision rule: when repeated in-frame optimization collides with the same architectural ceiling, price the architectural change rather than commissioning another round of local optimization. The eventual serverless migration encoded the architectural decision in the system itself, letting the platform own the scaling I had been hand-building against its own ceiling.

Return. The incident initially produced a decision rule rather than a governance mechanism. It became engineering capital only when later work could inherit the decision through the architecture and its recorded rationale. A lesson still held only in someone's head remains expensive to reuse.

5.4.3 The Decision Left Open

Problem. My merge-train cron kept stalling when two agents edited the same file.

[24] My field notes on the period were less measured: "Claude is the fastest road to hell." It was the speed with which a weakly specified direction could turn into a large amount of coherent wrong structure.

Initial response. It proposed a one-flag fix: on any conflict, take one side and drop the other. I signed off without asking what "drop the other side" meant.

Why insufficient. For four days it silently deleted real work from the main branch every time two changes overlapped. The same commit heading landed again and again, each time a little smaller, while I chased the symptoms one missing function at a time. The conflict handler contained a consequential design choice that had never been made explicitly: what information may be discarded when two changes overlap? At execution time the agent still had to choose, so it selected the locally expedient behavior that removed the conflict. This was an unresolved consequential decision arriving at the point of autonomous execution.

What became durable. The resolver was changed to inspect and reconcile the conflicting files rather than selecting one side blindly, and the destructive default was prohibited. The choice no longer had to be re-guessed on every conflict.

Return. A consequential decision made once and encoded is a form of engineering capital: future work inherits the decision instead of paying another probabilistic judgment at the worst possible moment. The portable lesson is narrow: do not leave consequential semantics unresolved at a boundary where an autonomous actor must act.

5.4.4 "Done" Was a Claim

Problem. Two incidents, close together, made "done is a claim" non-negotiable.

Initial response. I trusted the reports. Once, a lint gate hit a stray error, caught it in a catch-all that returned success, and let a run of commits land completely unlinted while the log said fine. Another time a chain of pin tests passed green.

Why insufficient. A botched merge had landed those pin tests onto a tree missing the very field they were supposed to check, so they confirmed a field that existed and never touched the one that mattered. Both reported done. Neither was. Both failures expose the same problem: a report of "done" describes the producer's claim, not the state of the repository, and the gap between the two is where a confident green hides.

What became durable. The response separated production from admission. Required tests and lints were re-run against the actual candidate tree at the relevant boundary rather than trusting the producing agent's report that they had run. Gate failures were made fail-loud: a control that swallows its own error is worse than no gate, because it reports success it did not earn, so a control cannot translate its own uncertainty into success. Where useful, counts and structural evidence were re-derived rather than accepted from the task report.

Return. "Done" became a claim requiring independent evidence. The capital is not a standing habit of human distrust; it is the machinery that lets future work regenerate the relevant evidence cheaply and consistently. The producer may have authored some of the tests. It does not get to decide, by report alone, whether the admission boundary should believe them.

5.4.5 A Content-Free Detector, Measured on the Wrong World

Problem. Late in the same stretch, an agent wrote me a content-free detector: if an image's pixels barely vary, skip the vision model and mark it decorative. Clean, typed, careful.

Initial response. Its own comment promised a false positive was "not possible" — real photos and charts vary twenty times more than the cutoff, so nothing real could slip under. The margin was true.

Why insufficient. It was measured on the wrong world. Hand it a Rothko, or Malevich's black square, and the pixels barely vary; the detector marks the painting decorative and drops it, and a blind student reading an art book gets silence where the subject should be. No threshold saves the detector, because a blank spacer and a low-variance artwork can be identical in the one quantity the code measures. "Content-free" is not a fact about pixels; it is a judgment about an image in context. The confident "not possible" exposed the missing model: the detector's representation of its input domain was too small.

What became durable. The cheap proxy lost the last word. Ambiguous cases are routed to a richer semantic judgment, and the regression evidence covers the *failure class* — low-variance images that nevertheless carry content — rather than only the first example that exposed it.

Return. The failure was not merely a bad threshold. The representation was inadequate to the obligation: pixel variance could distinguish one visual property, but not whether an image carried meaning in context. The cheap judge therefore could no longer decide the semantic question for the system. Ambiguous cases moved to a richer representation and a semantic judge, while regression evidence preserved the failure class that exposed the mistake.

5.4.6 The Retry That Became a Storm

The failures so far were about operating the fleet. This one and the two that follow are about the product the fleet built — the models inside DocAble — and each, like the mechanisms above, began in a failure.

Problem. The dominant production failure was a slow generation call, and the lifecycle around it was implicit. Status got written from scattered places, and one terminal state masked many distinct internal failures.

Initial response. The reflex to any failed job was to requeue it. Retry was the default recovery for every timeout and every crash.

Why insufficient. Under a slow dependency that reflex became a storm — requeue, still slow, same timeout, requeue again — and the recovery amplified the load precisely when the system was already hurting. Cost hid the damage. Every attempt wrote its full cost with nothing marking it a retry, so a job tried three times billed roughly three times over and the dashboards showed a silent multiplication no one could explain.

What became durable. Three structures replaced the retry reflex. An explicit lifecycle named the legal states and transitions. Atomic claiming separated permission to act from

recovery leases, preventing redelivery from authorizing two workers concurrently. Ordinary failure stopped implying automatic requeue; planned preemption became the one modeled exception.

Return. A later stale-owner sweep exposed the next missing distinction: slow is not dead. Progress evidence let recovery distinguish abandoned work from work still advancing. Recovery stopped being a reflex and became a decision over explicit state and evidence.

5.4.7 Provenance Became a Lookup

Problem. A corrupted document once took four cross-file hops to trace back to the pass that wrote it. Each hop was cheap; the chain was expensive. The system could not cheaply answer the plainest question a remediation tool owes its user: which step changed this?

Initial response. Stamps were written by hand, site by site, with nothing checking that a given mutation was stamped at all. A low-level write path recorded its changes into the document's own structure and never into the registry the changelog reads.

Why insufficient. Verification found eleven production sites writing through that low-level path, skipping the stamp helper, shipping empty changelog rows for a whole class of jobs. The customer's account of what changed came out blank while the code had changed plenty. An empty changelog row is "done" as a claim again — a report the work cannot back.

What became durable. The changelog stopped being an independently authored account. Each mutation writes its stamp — the verb, the pass, the visibility — and the customer-facing changelog is derived from those records rather than authored by hand, so diagnosis stopped being inference and became a lookup. A wiring validator identifies mutating paths that fail to stamp their work. It reported rather than blocked at first, because the inherited tree still contained many violations; a fix wave drained the gap; only then did the obligation become authoritative at the admission boundary.

Return. A forensic question that once required several cross-file hops became a lookup over derived evidence — bounded by the stamp model and the wiring check, not a guarantee larger than those. The sequence matters: representation made the obligation observable; migration made it feasible to satisfy; gating made it authoritative. This was the incident history behind Audit → Drain → Promote.

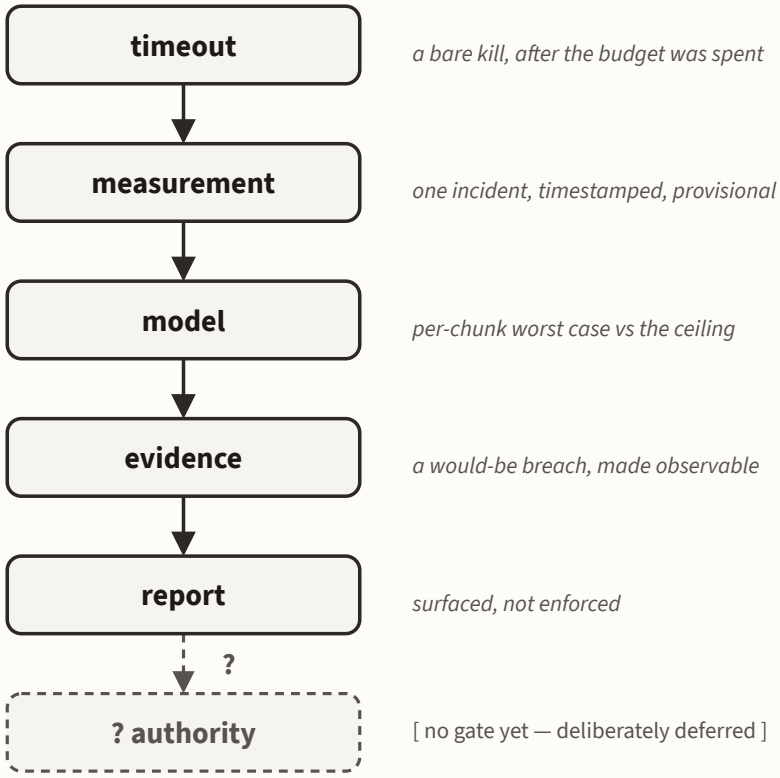
5.4.8 Measurement: the Story That Stops Before the Gate

Problem. A document's repair stalled in silence. The subprocess was killed by the job timeout after roughly eight minutes, and the job failed with an opaque message while the customer's progress bar sat frozen at thirty-eight percent, the budget already spent.

Initial response. The timeout was a bare cap. It provided no predictive account of per-chunk time or cost and no useful baseline for deciding whether the current work was approaching the job ceiling.

What became durable. The incident produced a provisional measurement and a per-chunk worst-case model against the job budget. The seed remains externalized, timestamped, and explicitly provisional — kept as a datapoint in a file rather than frozen as a constant in the code — so one incident does not quietly become an eternal constant. Current instrumentation can report a would-be breach. Figure 5.4-1 draws where the chain deliberately stops.

A model left unbound on purpose: modeled, observed, reported, not enforced.



An engineering history, not a system model.

Figure 5.4-1: Modeled and Observed, Not Yet Binding. *A timeout incident produced a measurement, a provisional per-chunk model, and reportable evidence. Production admission still does not depend on the model: its bound is not calibrated strongly enough to justify blocking. The terminal authority node is left open on purpose — representation and observation can mature before authority is earned. The open node is part of the evidence, not an omission.*

Return. The history deliberately stops there. No production gate rejects the job on this model today. The bound is not calibrated strongly enough to justify enforcement, and a false refusal could cost more than continued observation. Better representation and better evidence are already useful engineering outcomes. **Alignment need not culminate in a blocking gate.** The question mark belongs in the book. This per-chunk timing model is

distinct from the account- and job-level GenAI budget controls discussed in Part III; different cost representations support different engineering decisions.

The counterpart shows measurement supporting action when its evidence is adequate to the decision. A serverless fleet that scaled to zero still carried a warm per-request floor that fleet-average numbers hid. Measured deterministically at the request level, that floor read as 4,057 ms; a deployment-topology model turned it into the longest user-visible cold-start path and drove a re-architecture that cut the floor to 109 ms. Here the measurement was calibrated well enough to drive a change rather than merely report one. The two histories show different outcomes from the same progression from representation to evidence: one remained observational; the other justified architectural action.

5.4.9 A Mechanical Conversion

One conversion is small enough to state in a sentence. Invisible remediation artifacts — the ones a document gains for accessibility but an author never sees — were supposed to carry a reserved name prefix so a validator could always tell an inserted element from an authored one. Review kept catching insertions that skipped it. A prefix rule is entirely decidable, so the check became a lint: any inserted artifact without the prefix now fails the build rather than waiting for a reviewer to notice it was missing.

Some conversions were mechanical like that. A review kept finding the same decidable violation in the repository; the response was not to remind future agents more forcefully, but to turn the audit question into a blocking lint. The judgment moved from repeated inspection into the environment, where it fired the same way on every commit.

5.4.10 The Ex-Ante Path

Some useful structures were designed from known obligations rather than produced by incidents. The unified detect-and-repair engine — one rule engine that ended a duplication tax where two separate checkers had scanned the same document in two languages a round-trip apart — consolidated those paths for a clean architectural reason. A later audit, prompted by a routing incident elsewhere, tested the boundary and found it holding. No subsequent failure forced the model into a stronger form.

Report that as confirmation, not conversion. Some assets in DocAble were forged by incidents and hardened through recurrence; others were designed from known obligations and simply survived later pressure. Part III called these the ex-post and ex-ante paths into the governed environment. The originating case contains both.

5.4.11 What This Case Supports

Step back from the individual incidents and one pattern does recur. Agentic velocity amplified the consequences of the surrounding engineering judgment in both directions. An underspecified choice could become a large coherent implementation quickly; a well-chosen abstraction or invariant could likewise propagate quickly once it was made durable.

The relevant unit was therefore not the agent's isolated success or failure, but what the engineered environment learned from the episode and whether later work inherited that learning.

Across the episodes, architectural decisions became typed seams; repeated findings became deterministic checks; implicit behavior became explicit lifecycle models; forensic reconstruction became provenance; and an opaque timeout became measurement. Some outcomes received authority, some remained observational, and some useful structures had no incident origin. That variation is part of the evidence.

The resulting claim is bounded. This is one longitudinal production case led by one primary engineer directing an agent fleet, using one contemporary model ecosystem, without a controlled comparison against another engineering process. It cannot establish that another organization will encounter the same failures, build the same mechanisms, or achieve the same economics. Checker scores and fidelity instrumentation also do not establish complete end-user accessibility.

What the case *can* establish is mechanism and sequence. These pressures occurred; these engineering responses followed; later failures sometimes exposed residual gaps; and selected structures persisted long enough to be exercised under continuing change. In that sense the clean MAGE method is not the story the project followed. It is the compression recovered from the engineering structures that repeatedly proved worth keeping.

One case cannot tell us how peculiar that compression is to the case that produced it. The final chapter changes the evidentiary question: when independent organizations build agentic engineering systems under different pressures, which structures recur, which differ, and where does the MAGE vocabulary stop fitting?

5.5 MAGE in the Wild

The originating case supplies depth. The useful next question is whether its engineering structures are peculiar to DocAble or recur when other organizations solve related problems independently. This chapter reads eight public industrial accounts—Cloudflare, Spotify, Shopify, Docker, Siemens, Zenseact, Uber, and GitLab—as **reconstructions** rather than case studies: they expose selected mechanisms rather than the full histories that produced them. They supply variation—comparable structures, alternative realizations, and places where the MAGE vocabulary fits poorly. **The question is narrower: what engineering structures recur, where do they differ, and what does the comparison force us to revise?**

One implication is worth flagging here. These organizations largely use agents to scale familiar software-engineering practices. That can make the software factory much more productive without necessarily making the resulting systems easier for engineers to understand. As autonomous production scales, the engineering question is whether explicit system knowledge and independent control must scale with it. Appendix G returns to that question as an organizational adoption problem.

5.5.1 Eight Entry Points

The organizations begin with different problems. One starts from institutional policy that has outgrown any reviewer's memory; another from the need to change thousands of repositories at once; another from making an organization's accumulated knowledge available to every agent; Uber starts from the economics of operating an agentic software factory at organizational scale; GitLab starts from making abundant code trustworthy at machine-scale development. Figure 5.5-1 shows the problem each organization starts from and where it invests in the engineered environment.

Eight entry points into a shared design space

Each organization begins with a different problem and invests in a different part of the environment.

Modeling · knowledge emphasis Governance · alignment emphasis

ORGANIZATION	STARTING PROBLEM	INVESTMENT	EMPHASIS
Cloudflare	Institutional policy	**Machine-readable requirements + review**	**Governance**
Spotify	Fleet-scale change	**Targeting, verification, admission**	**Governance**
Shopify	Organizational knowledge	**Shared context + reusable agent infra**	**Knowledge**
Docker	Delegated runtime authority	**Scoped roles, tools, evaluation**	**Governance**
Siemens	Model-first engineering	**Persistent executable representations**	**Modeling**
Zenseact	Distributed autonomous work	**Platform boundaries, context, delegation**	**Knowledge**
Uber	Software-factory economics at scale	**Benchmarks, routing, reusable skills, outcome measurement**	**Knowledge + governance**
GitLab	Trust at machine-scale development	**Durable context, provenance, verification + governance**	**Knowledge + governance**

MAGE the vocabulary used here to compare the observed structures

The figure locates emphasis; it does not rank maturity or claim any source implements the whole method.

Figure 5.5-1: Eight Entry Points. *MAGE provides the comparison vocabulary; no source is claimed to implement the complete method. The organizations begin with different problems and invest in different parts of the agentic engineered environment. The figure locates emphasis; it does not rank maturity.*

Cloudflare gives the cleanest policy-first example. A standards corpus that had outgrown individual memory was externalized into structured, machine-readable requirements; me-

chanically checkable obligations could move into custom checks while ambiguous ones remained review questions. Humans retained policy authority while agents increased the reach of decisions made elsewhere. The important limitation is equally revealing: the source richly models **obligations**, but gives much less evidence of an executable model of the governed software itself. That difference anticipates the modeling ceiling visible across the corpus.

Uber and GitLab reach the same kind of engineering from opposite motives: both build a durable environment around a replaceable reasoner. GitLab starts from trust — as code becomes abundant, the scarce problem shifts to trusting it, so it wraps the model in a durable layer of context, verification, governance, identity, and provenance, kept independent of any one model so the organization's controls persist even as the reasoner changes.[76] Uber starts from economics, and gives the fuller, independently measured account, so the deep treatment follows it.

Uber's public account describes AI tools operating throughout the software lifecycle: more than 70 percent of pull requests are attributed to local or cloud agents, engineers have created more than 3,600 agent skills, and those skills execute more than 30,000 times per day. Managed agents perform code review, CI repair, end-to-end changes, alert triage, debugging, and maintenance work.[77] The interesting structure is not merely the volume of agent use. Uber treats the surrounding environment as an object of engineering: real workloads become benchmarks; models are selected against cost, quality, and reliability; context and tool access are compiled down to reduce unnecessary reasoning; recurring workflows become reusable skills; and managed agents are evaluated in units such as cost per merged pull request, review, or alert.

This is recognizably MAGE-like without being model-first. Both organizations engineer the conditions under which agents act — what information and tools they receive, how work is decomposed, what it costs — and both keep a knowledge-and-context graph as the reasoning surface (Uber's AI Context Graph, GitLab's Orbit) rather than an executable model of the software's behavior. Their accounts say much more about context, skills, routing, and governance than about behavioral, process, scenario, or invariant models. Their sophistication therefore strengthens rather than dissolves the modeling ceiling developed below.

5.5.2 The Shared Baseline

Across the eight sources, four moves recur strongly enough to form a shared baseline. **Knowledge is externalized** rather than rebuilt from a person's memory on every task. **Action runs through bounded tools or roles** rather than giving the reasoner unconstrained authority. **Generation is separated from evidence** through tests, validators, reviews, simulations, or other independent checks. And **consequential decisions remain human** even where the machinery performs most of the work. The implementations differ substantially; the same engineering grammar recurs across them.

Uber adds a fifth recurring concern made especially visible at scale: the environment itself is continuously measured and optimized, so model choice, context, tools, decomposition,

and reusable procedures become engineering variables rather than fixed properties of the reasoner.

Across the reconstructions, safe autonomy relies not on making the reasoner infallible but on legible context, bounded action, independent evidence, and explicit limits on machine authority.

5.5.3 Where the Reconstructions Differ

The differences are as informative as the commonalities. The organizations vary in how aggressively they automate admission, how much organizational knowledge they route into each task, whether recurring failures change the shared environment, and, most sharply, how explicitly they model the governed system itself. Docker, for example, mechanizes substantial evaluation while deliberately retaining a human merge decision; Spotify pushes further toward automatic fleet-scale admission; Shopify turns shared interaction into reusable organizational context; Siemens begins from persistent engineering models rather than from source code. Uber exposes another axis: it treats cost per useful outcome as a first-class property of the agentic environment and tunes models, context, tools, and reusable skills against that objective.

No ordering follows automatically from those differences. A human admission boundary may be exactly right where the decision remains semantic or consequential. A richer model may not repay its cost in another environment. The comparison is useful because it exposes **design choices and boundaries**, not because it produces a winner.

The systems also differ in where they leave trust. Some primarily improve the information available to the reasoner; some make organizational knowledge persistent; some encode policy or structure into machine-actionable representations; some enforce selected properties independently of the reasoner. These are not maturity levels. They are different allocations of consequential judgment between reasoner and environment. They also allocate judgment differently among representations and judges: some improve the context presented to a probabilistic reasoner, some move selected questions into machine-actionable structure, and some enforce mechanically established results independently of the producing reasoner.

5.5.4 The Modeling Ceiling

One difference is stark. In the seven software-first accounts, the richest representations described publicly are primarily **knowledge and governance representations**: service catalogs, dependency information, infrastructure definitions, policy corpora, repository metadata, skills, context machinery, provenance records, and lifecycle graphs. These are important models in the broad sense used by MAGE—they externalize engineering knowledge, preserve consequential relationships, and extend what an agent can recover reliably—but the sources give little evidence of behavioral, process, scenario, or invariant models serving as the primary surface of software reasoning.

GitLab illustrates the boundary clearly. Its context graph makes relationships among requirements, code, pipelines, deployments, production signals, policy, and evidence available to agents and governance machinery. That is richer than disconnected files or retrieved prose, but its public role is still primarily to preserve and expose engineering context, provenance, and control relationships.

Siemens crosses that boundary because it comes from a model-first engineering tradition. Behavioral structure, traceability, requirements, simulation, and analysis already live in persistent engineering representations; software generation is one downstream realization. **The comparison shows that the strong end of the Modeling Principle has an established precedent in model-based engineering but is unusual in the software-first agent systems examined here.**

The originating case goes one step further. For selected models, mechanical checks compare declared relations to the implementation and reject changes when those relations no longer hold. These checks cover particular relations rather than establishing general correspondence between model and implementation, and the case does not show how broadly such checks can be applied at reasonable cost.

Figure 5.5-2 states the finding as three tiers and marks where the source evidence sits.

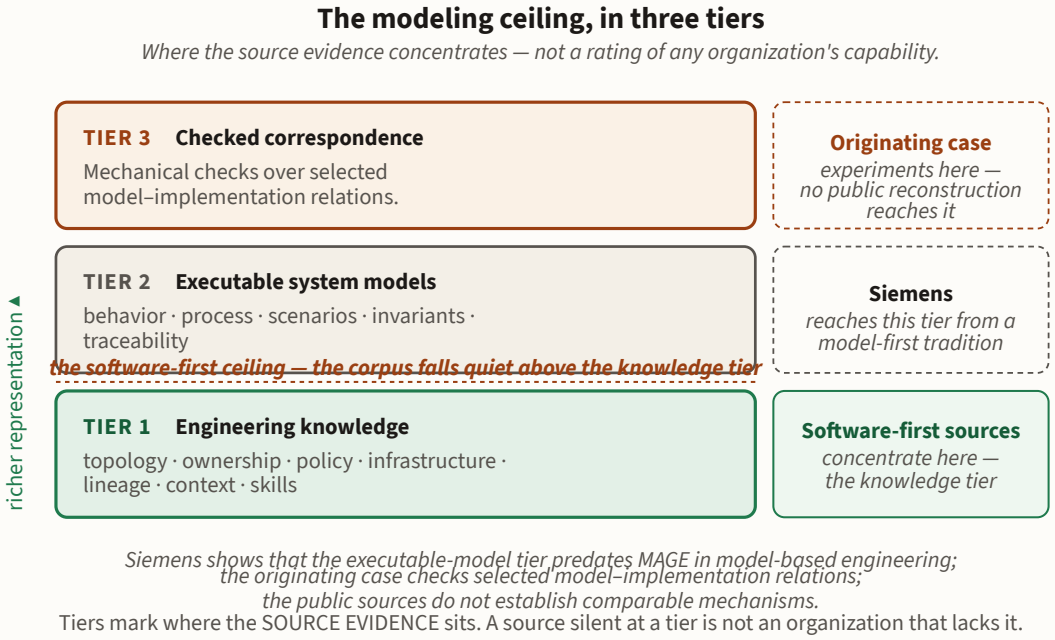


Figure 5.5-2: The Modeling Ceiling. *Three tiers of modeling: engineering knowledge, executable system models, and checked correspondence. The software-first corpus concentrates on the knowledge tier; Siemens reaches the executable-model tier; the originating case checks selected model–implementation relations, and the public sources do not establish comparable mechanisms. The tiers locate the public source evidence, not organizational capability.*

5.5.5 Engineering Capital at Organizational Scale

The industrial reconstructions show something the single-engineer origin cannot: how these moves operate across an organization. Four moves recur. **Organizations centralize shared mechanisms while keeping domain judgment distributed.** Platform teams own common authentication, execution, safety, or policy machinery while domain teams retain the expertise that decides what their agents should do. **Human judgment moves upstream:** one engineer can scope and supervise a migration implemented concurrently by a fleet rather than hand-authoring every change. **Private learning becomes a shared artifact:** sessions, policies, skills, and defaults become useful only when later work can inherit them. And **the unit of governance grows:** from a task or agent to classes of work, permissions, fleets, policies, and platforms.

Uber makes this conversion unusually literal: recurring tool workflows are packaged as reusable skills, and standard agent configurations encode routing and context decisions that would otherwise have to be reconstructed on each session. In one example, deterministic tool orchestration is moved out of repeated model turns into executable code, then packaged for reuse across later work. The source reports that code-mode converts repeated tool interactions into subprocess execution, keeping intermediate polling outside the model context; on common SQL workflows it cuts token use by more than half and by more than 90 percent in bulk cases, with more than 25 such workflows packaged as reusable skills.[77]

GitLab describes the same conversion from repeated judgment to durable structure. A production failure can become a regression test; a security incident can become a policy; a performance requirement can become an automated constraint; and a compliance obligation can become continuous validation. The artifact differs, but the engineering move is the same: knowledge gained in one episode is converted into structure that later work can inherit rather than requiring the same judgment to be reconstructed each time.[76]

The common structure is leverage. A scarce human decision becomes more valuable when it is placed where the environment can carry it forward: one policy can shape thousands of changes; one shared representation can guide work across a fleet; one learned constraint can protect every later task over the same surface. This is the organizational face of engineering capital. Governance does not remove judgment. It gives selected judgments multiplicative reach.

5.5.6 They Converged on a Problem

The eight organizations do not converge on one stack, one degree of autonomy, or one modeling discipline. They converge on a recognizable engineering problem: how to make abundant intelligence useful without requiring proportional growth in human vigilance. Their answers repeatedly externalize knowledge, bound action, separate generation from evidence, and decide deliberately which obligations the environment will enforce. They differ in what they model, what they mechanize, and what they still ask a person to decide.

The comparison does not prove MAGE's causal claims. It shows something narrower: the originating case is not the only place where this engineering grammar appears. DocAble supplies mechanism and sequence; the industrial reconstructions show related structures emerging independently under different constraints.

5.5.7 Four Views of Repeated Judgment

The approaches these organizations use can work. Better foundation models can make better engineering judgments. Better prompts, decomposition, retrieval, context, and persistent organizational knowledge can improve those judgments further. Independent evaluation can reject bad candidates and send the work around another loop. Nothing in the evidence here argues otherwise.

The difficulty appears with scale and time. An engineering obligation rarely needs to survive one realization. It must survive repeated changes to a growing system. Each time satisfaction of the obligation depends on a fallible reasoner retrieving, interpreting, and honoring it, engineering incurs another probabilistic exposure. The four views below hold that process still from four angles: how often the judgment must go right, how much reasoning another attempt buys, where engineering can act, and what one judgment carried forward changes.

View 1 — Repeated Exposure

Part I introduced a deliberately simple model for consequential probabilistic judgment. Apply the same model across time. Let an obligation face relevant exposures $1, ..., m$, and let p_j denote the probability that it is resolved acceptably on exposure j. Under the same illustrative independence assumption,

$$P(\text{obligation survives all exposures}) = \prod_{j=1}^{m} p_j$$

If the probability is constant across exposures, this becomes

$$p^m$$

The equation is not a fitted model of agent behavior. Errors can be correlated, tasks differ, and engineering processes can detect and repair failures. Its purpose is structural: reliability depends both on how favorable each probabilistic judgment is and on how often engineering requires that judgment to be made again.

Even reassuring per-exposure reliability can erode under repetition. An illustrative $p = 0.999$ gives

$$0.999^{1000} \approx 0.37$$

The number is pedagogical, not an estimate of model reliability. Its point is that a long-lived system repeatedly re-exposes consequential obligations when their satisfaction remains dependent on per-instance inference.

This gives us two dimensions of scale. A larger system can place more consequential obligations and relationships in play during one change. A longer-lived system accumulates more changes over which those obligations must continue to hold. Part I emphasized the first dimension. The industrial evidence makes the second difficult to ignore.

View 2 — Retry Purchases Another Chance

The first view treats each exposure as one judgment. Real agent systems often do something more powerful: generate a candidate, evaluate it, feed the result back, and try again. A loop does not make the reasoner deterministic; it purchases additional chances to obtain an acceptable realization.

Suppose one attempt has probability p of satisfying the obligation, each attempt costs c, and the loop permits at most k attempts. Under the same simplifying independence assumption,

$$P(\text{success within } \ k) = 1 - (1-p)^k$$

The expected number of attempts actually consumed is

$$E[N_k] = \frac{1-(1-p)^k}{p}$$

so expected attempt cost is

$$E[C_k] = c\frac{1-(1-p)^k}{p}$$

As k grows, eventual success approaches one whenever $p > 0$, while expected cost approaches

$$E[C] = \frac{c}{p}$$

This is why long agent loops can work extremely well without making the underlying reasoner deterministic. Repetition purchases reliability with additional inference, realization, validation, latency, and sometimes human attention.

A simple example shows the tradeoff. Suppose an attempt succeeds half the time. Ten attempts give

$$1 - 0.5^{10} \approx 99.9\ \%$$

probability of finding an acceptable result while consuming almost two attempts on average. If a better reasoning surface raises per-attempt success to 0.9, three attempts give

$$1 - 0.1^3 = 99.9\ \%$$

with only about 1.11 attempts expected. The eventual reliability is essentially the same. The amount of reasoning purchased to obtain it is not.

This view therefore adds something the first one cannot show: reliability and reasoning cost are separate axes.

View 3 — Engineering Can Act at Different Places

Retry is only one intervention. Engineering can act at several different places in the process.

A stronger foundation model can increase p: when the consequential judgment must still be made, a satisfactory answer is more likely on each attempt. Better context, retrieval, tools, specifications, and representations can also increase p, reduce the cost c of an attempt, or remove preliminary reasoning that would otherwise be spent reconstructing the relevant structure. Decomposition, critique, testing, simulation, and loops can purchase additional attempts. Independent validators and gates act differently: they need not improve the producing judgment at all, but can reject covered failures before they are accepted. And where an obligation can be represented and governed adequately, engineering can sometimes stop requiring the probabilistic judgment in the first place.

A simple comparison makes the last distinction visible. Suppose an obligation is resolved acceptably with probability 0.99 on each of 100 independent exposures:

$$0.99^{100} \approx 0.366$$

Improving per-exposure reliability to 0.999 gives

$$0.999^{100} \approx 0.905$$

That is a large improvement, and better probabilistic engineering should not be dismissed. But engineering can also act on the number of exposures. If durable structure makes 90 of those 100 occasions no longer require the probabilistic judgment, leaving ten exposures,

$$0.99^{10} \approx 0.904$$

And the approaches can combine:

$$0.999^{10} \approx 0.990$$

These are different interventions. Improve the odds. Purchase another attempt. Catch a bad realization before it is accepted. Or stop requiring that judgment on every change. MAGE uses all four where appropriate.

The industrial reconstructions occupy different parts of this design space. Shopify and Zenseact emphasize the information available to reasoning. Docker and Spotify make evaluation and admission increasingly explicit. Cloudflare turns selected policy into machine-readable requirements and mechanically checkable obligations while retaining human authority over ambiguous cases. Uber explicitly optimizes the economics of reasoning itself, routing work across models and restructuring context, tools, and reusable procedures to reduce the cost of satisfactory realization. GitLab places context, provenance, verification, identity, and policy into a durable layer surrounding replaceable models and agents. Siemens begins farther upstream, with persistent executable engineering representations serving as a primary reasoning surface. These are not maturity levels. They are different allocations of reasoning, cost, evidence, and authority between reasoner and environment.

A stronger environment can change what is verifiable. The preceding comparison treats verification largely as a fixed capability: given an obligation, a validator either covers it or does not. Modeling makes that capability itself an engineering variable.

Let O be the set of consequential obligations relevant to a change, and let $V(E) \subseteq O$ be the obligations that engineering environment E can verify independently of the producing reasoner. Improving the agent may increase the probability that obligations outside $V(E)$ are satisfied correctly. Improving the environment can instead enlarge $V(E)$:

$$E_1 \to E_2 \qquad \text{such that} \qquad V(E_1) \subset V(E_2).$$

The distinction matters because an obligation moved into $V(E)$ no longer depends solely on the producing reasoner getting it right. A behavioral model can make an execution property analyzable; an invariant can make a relation mechanically checkable; a model–implementation check can reject a change that breaks a declared correspondence. Better reasoning improves performance within the existing verification boundary. Better engineering can move the boundary.

The industrial reconstructions largely demonstrate the first kind of expansion around a human-derived engineering environment: give agents better context, faster feedback, broader tool access, stronger tests, and more explicit governance. MAGE includes those moves, but also asks where machine-scale engineering makes richer representations and stronger analyses economical. The two strategies are distinct and compose: a better reasoner raises p; a stronger environment enlarges $\mid V(E) \mid$. Part VI develops this distinction as a theory of the evolving engineering environment; Appendix G turns it into an organizational adoption problem.

View 4 — One Judgment Can Acquire Durable Reach

The preceding views still treat engineering largely as a sequence of realizations. The originating case suggests one more possibility: an engineering judgment made today can change what later realizations have to purchase again.

Architectural decisions became typed seams; recurring findings became repeatable checks; implicit behavior became lifecycle models; forensic reconstruction became provenance. Each converted work that would otherwise recur across future changes into something later work could inherit:

$$\text{one engineering judgment} \longrightarrow \text{an asset applied across future work}$$

This is the connection to engineering capital. A useful representation can spare later reasoners from reconstructing the same structure. A validator can make a recurring evaluation repeatable. A constraint can prevent a class of realization from arising. A gate can enforce an established obligation. The value of the original judgment can therefore extend across later work rather than being repurchased one realization at a time.

As recurrence grows, the potential value of that conversion grows with it — but so do the reasons not to mechanize everything. Durable structure has construction cost, carrying cost, friction, and depreciation. Some judgments remain semantic, unstable, or too expensive to encode. The engineering question is not how to eliminate probabilistic reasoning. It is which judgments should remain probabilistic, how much reasoning should be purchased around them, and which recurring judgments are worth carrying forward in the environment.

What These Views Leave Out

These models are deliberately simple. They make repeated exposure, retry, reasoning cost, and durable reach visible by holding several things fixed. Sustained engineering violates those simplifications in two important ways.

First, the engineering environment can change the reasoning required by an attempt. An explicit representation may let an agent reason directly over architecture, state, dependencies, or obligations that it would otherwise have to reconstruct from implementation. Modeling can therefore increase the chance of a satisfactory realization, reduce its reasoning cost, or eliminate some preliminary reasoning altogether.

Second, successive realizations are likely not independent. Each change alters the system and engineering environment inherited by later work. Good structure can make later reasoning easier; architectural decay can make it harder. Tomorrow's probability and cost can depend on what today's realization leaves behind.

Part VI develops a theory around these interactions. MAGE does not necessarily reduce how many changes occur. It changes the reasoning each change requires, which consequential judgments must still be purchased probabilistically, and what engineering structure later work inherits.

What the Evidence Supports

Part V has supplied two different forms of evidence for MAGE.

The DocAble case provides depth. Its longitudinal record shows engineering structures emerging, changing, and sometimes being replaced under sustained implementation pressure. The case makes sequence and mechanism visible: missing representations can make global properties difficult to reason about; missing authority can leave consequential obligations dependent on repeated judgment; recurring failures can motivate durable changes to the environment; and explicit models can move from analytical aids into structures that participate directly in realization.

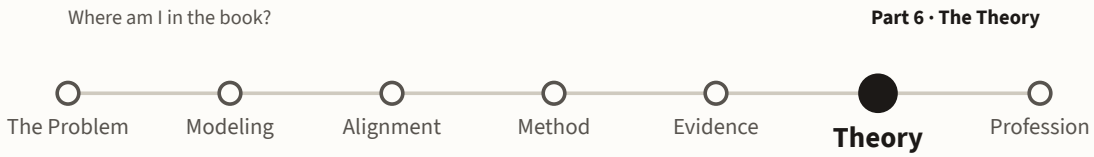
The industrial reconstructions provide breadth of realization. Systems developed independently under different constraints exhibit related moves: externalizing engineering knowledge, constraining delegated work, producing independent evidence, governing admission, and preserving useful judgment in structures later work can inherit. They also show that those moves need not share DocAble's architecture, terminology, or mechanisms.

The two forms of evidence have different limits. One deeply observed system cannot establish population-level effects or show that MAGE uniquely explains its development. Selected industrial accounts provide variation but less process history and do not constitute a representative sample. Neither view establishes that adopting MAGE causes greater productivity, quality, or safety.

What the evidence supports is narrower and sufficient for the next step: the constructs recur, the proposed mechanisms are plausible in practice, and their relationships are consequential enough to require explanation.

Part VI develops that explanation as a theory and asks what should follow if it is right.

Part 6: The Theory



QUESTION THIS PART ANSWERS

How does MAGE work, what should follow if the account is right, and where should we expect it to apply?

THE SYNTHESIS

Agentic capacity does not by itself produce engineering progress. It acts through the engineered environment surrounding the work. Modeling changes the representations through which consequential reasoning occurs; Alignment gives selected obligations authority independent of the producing reasoner.

Together, these structures shape how much autonomous capacity becomes durable progress, how much failure escapes, and how much human judgment must still be purchased along the way.

CARRYING FORWARD

Commodity intelligence · Modeling Principle · Alignment Principle · Governed Engineering Environment · Engineering capital · Probabilistic surface

NEW HERE

Dynamic Model · Environment quality · Determinization frontier · Representation innovation · Scope conditions · Testable predictions · Research agenda

Part V supplied two views of the evidence. The originating case showed how MAGE's structures emerged under sustained engineering pressure; the industrial reconstructions showed related structures appearing independently under different constraints. Neither establishes a universal law. Together, they give us something worth explaining.

It begins with a general theory of how agentic capacity interacts with the governed engineering environment. The theory treats implementation capacity as an input, not an outcome: the same capacity can produce durable progress or merely accelerate churn depending on the representations, evidence, constraints, and authority through which it acts. Engineering pressure exposes mismatches; diagnosis and adaptation can convert what engineers learn into structure that later work inherits.

We state the theory and its principal predictions first, deliberately in general form. Only afterward do we ask where those predictions should be expected to hold. MAGE imposes construction, maintenance, coordination, and governance costs, and not every engineering judgment can or should be transferred into durable machinery. The scope conditions therefore bound the theory rather than precede it.

The Part then turns from claims to inquiry. Its final chapter translates the theory, predictions, and scope conditions into a research agenda: what should be measured, what comparisons could distinguish the proposed mechanisms, which quantities remain unidentified, and what evidence would strengthen—or weaken—the account.

The objective is not to declare MAGE a universal law of software engineering. It is to make the explanation precise enough to be wrong.

Carrying forward: Commodity intelligence · Modeling Principle · Alignment Principle · Governed Engineering Environment · Engineering capital · Probabilistic surface

New here: Dynamic Model · Environment quality · Determinization frontier · Representation innovation · Scope conditions · Testable predictions · Research agenda

6.1 A Theory of MAGE

Part V ended with two complementary views of agentic engineering: one system observed deeply enough to reconstruct mechanisms and sequence, and independent systems observed broadly enough to expose variation. This chapter proposes an explanation for both.

The theory is directional rather than numerical. It identifies constructs, mechanisms, and relationships that should hold under stated conditions. Section 6.2 turns those claims into testable predictions.

The central claim is simple. **Agentic capacity acts through the engineered environment that surrounds it.** A strong environment turns more implementation capacity into durable progress; a weak or incoherent one turns the same capacity into churn, escaped defects, repeated intervention, and control friction. Those outcomes also change what later work inherits. Poor realizations can degrade architecture, representations, or conventions, making subsequent work harder; engineering judgment can instead diagnose recurring problems and adapt the environment so that later work inherits better structure. As the environment improves, larger tasks and greater autonomy become feasible, exposing the next frontier.

THEORY OF MAGE — INFORMAL

Agentic capacity does not by itself produce engineering progress. It magnifies the fit —or misfit—between the work attempted and the engineered environment that surrounds it.

Part I began from two limitations of autonomous realization. Reasoners have finite reasoning horizons, and consequential judgments entrusted to fallible inference retain residual risk. Parts II and III supplied two engineering responses: change the representation over which reasoning occurs, and give selected obligations independent authority. Part V showed a broader repertoire in practice: improve the reliability of probabilistic judgment, catch bad realizations through independent evidence and admission, or transfer selected recurring judgments into durable structure.

The probability-and-cost model in Part V was deliberately narrow. It made repeated exposure, retries, and the cost of purchasing acceptable realizations visible; it did not model the full lifecycle of defect detection, recovery, human attention, representation maintenance, mechanism friction, or environmental adaptation. The theory here widens the unit of analysis. It asks how agentic capacity and the engineered environment jointly determine realized performance, how that environment adapts, and when the resulting investment pays.

6.1.1 The Dynamic Model

Figure 6.1-1 puts the theory on one diagram. **Agentic capacity** supplies model capability, concurrency, and raw implementation rate. The **governed engineering environment** supplies the representations, constraints, evidence, and authority through which that capacity acts. Together they shape **realized performance**: durable throughput, defect escape, and human attention consumed per unit of useful output.

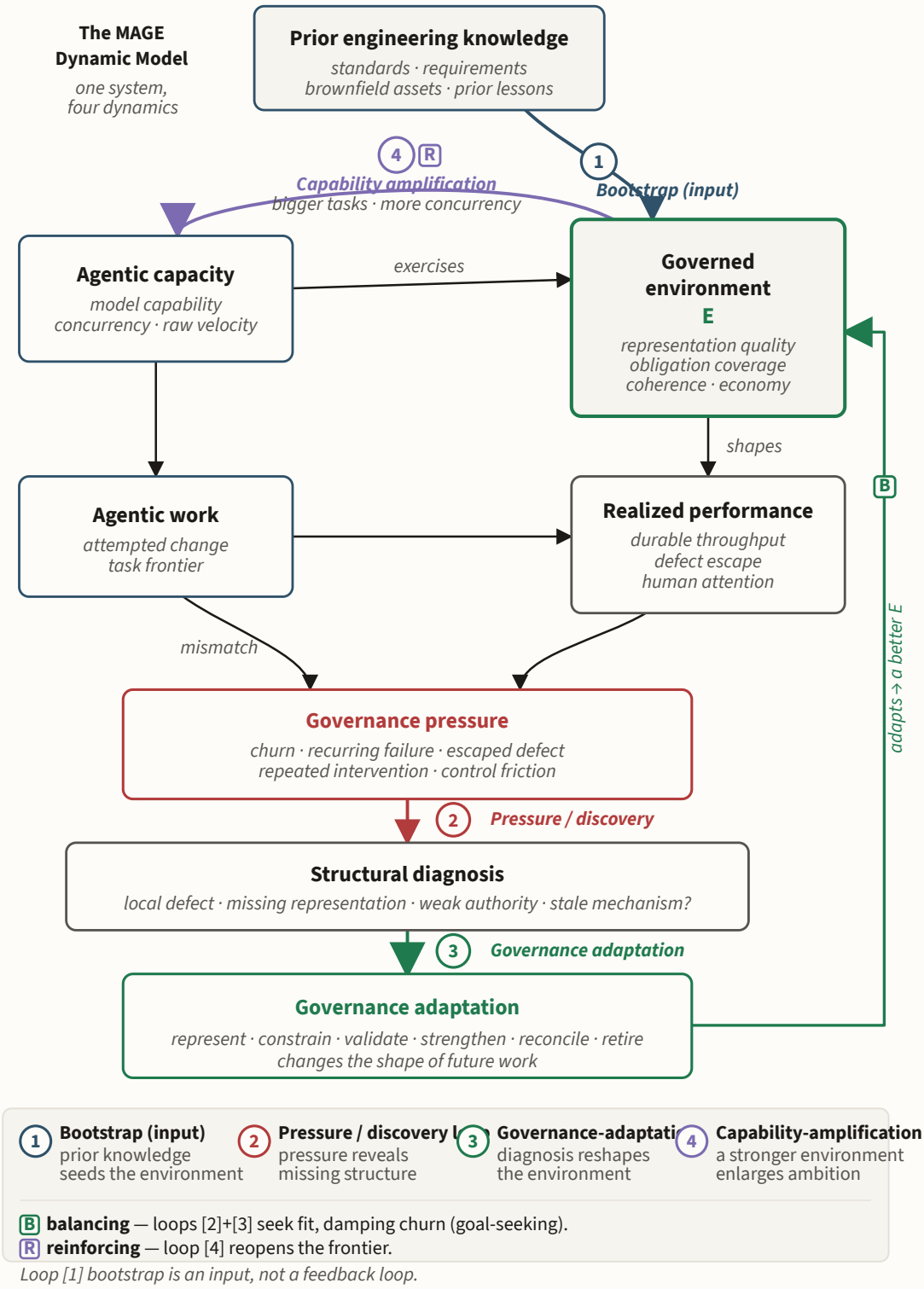


Figure 6.1-1: The MAGE Dynamic Model. *Agentic capacity acts through a governed engineering environment to produce realized performance. Mismatch appears as churn, escaped defects, and repeated intervention; diagnosis and adaptation reshape the environment; a stronger environment then supports more ambitious work and exposes a new frontier. Prior engineering knowledge supplies known structure before failure occurs. The model is directional, not fitted.*

When the environment does not fit the work, the mismatch becomes visible as churn, recurring failures, escaped defects, repeated intervention, or friction from the controls themselves. These symptoms tell engineers that something is wrong. They do not say what is wrong: an engineer or organization still has to decide whether the signal is a local defect, a missing representation, a weak boundary, an inadequate control, or an obsolete piece of machinery. That diagnosis drives the change to the environment, which shapes the next round of work.

The process has a simple feedback structure. Prior engineering knowledge gives the environment a starting point. Work then exposes gaps between what the task requires and what the environment supports. Engineers diagnose those gaps and improve the environment. A stronger environment supports more ambitious work, which eventually exposes new gaps. The cycle repeats as the system, the available agentic capability, or engineering ambition changes.

This feedback structure has a precedent in cybernetics, which studied how regulators use information about a system to keep its behavior within acceptable bounds. Conant and Ashby's good regulator theorem makes representation central: under its assumptions, the simplest optimal regulator must embody a model of the system it regulates[78].

MAGE applies a related idea to software engineering. The environment represents consequential properties of the system and its engineering process, observes what happens as work proceeds, and changes when its representations or controls prove inadequate. The recurring question is simple: what must the environment know in order to govern the work?

Not all useful structure is discovered through failure. Existing architecture, standards, requirements, operating history, and lessons imported from earlier systems can seed the environment before an agent acts. Other needs become visible only in use. MAGE therefore has two routes to durable structure: **ex ante**, where a known obligation or useful representation is engineered deliberately, and **ex post**, where experience exposes a recurring cost or risk worth externalizing. Governance conversion names the latter route; it is not the origin story of every model or mechanism.

Two conditions determine whether adaptation reaches the shared environment. First, someone has to recognize that pressure is structural rather than local. Second, that diagnosis needs a path to the shared environment—the architecture, models, mechanisms, or infrastructure that can retire the failure class. Without diagnosis, pressure produces motion without direction. Without authority, the lesson dies as a local patch. Section 6.3 treats both as scope conditions.

6.1.2 What Makes an Environment Effective

More machinery does not mean a better environment. Ten thousand lines of stale checks are worse than one hundred lines that govern the obligations that matter. What counts is not the quantity of apparatus but how well the environment fits the work.

An effective environment needs four things:

1. **Useful representations.** The information needed for the work is explicit at an appropriate level of abstraction.
2. **Adequate coverage.** Important obligations have appropriate evidence and authority.
3. **Coherence.** Models and mechanisms reinforce one another rather than contradicting or needlessly duplicating.
4. **Economy.** Their future value justifies their maintenance, friction, and context cost.

Together these determine how much consequential work still depends on fallible per-instance inference—the probabilistic surface of Part I. Representation makes consequential knowledge explicit and, where its structure permits, more tractable; coverage supplies evidence and authority; coherence determines whether the machinery composes; economy asks whether retiring a repeated judgment was worth what the environment must now carry.

Coverage does not require mechanizing every obligation. Some belong in a compiler rule or admission gate; others are statistical, semantic, or consequential enough that human judgment remains the appropriate authority. A strong environment does not mechanize everything. It makes explicit **where judgment belongs and where repeatedly paying for it would be wasteful**.

These are not stages of a maturity model[79]. Improve the part of the environment that is limiting the work. If engineers repeatedly reconstruct important state, improve the representation. If failures escape, strengthen evidence or authority. If controls have become a tax, reconcile or remove them.

6.1.3 Outcomes and Observables

A theory of agentic engineering should distinguish activity from progress. MAGE therefore treats raw implementation velocity as an input and distinguishes three outcomes: **durable throughput**, **defect escape**, and **human-attention burden**. Table 6.1-1 defines each outcome and identifies candidate observables.

Table 6.1-1: Outcomes of the theory. *Raw implementation activity is an input. The theory is interested in how much survives as durable progress, what failures escape, and how much scarce human judgment that progress consumes.*

Outcome	What it measures	Candidate observables
Durable throughput	Raw implementation converted into lasting system progress	landed changes net of rework; churn share; the slope of sustained throughput
Defect escape	Relevant failures reaching later lifecycle stages or production	escaped-defect rate; reopen rate; silent-corruption escape
Human-attention burden	Human judgment required per unit of durable output	interventions per landed change; review time; conversion effort

Durable throughput is useful system progress that survives rework, integration, and operation rather than merely landing in the repository. Defect escape records consequential failures that reach later lifecycle stages or production. Human-attention burden records the scarce engineering judgment consumed per unit of durable progress.

Several easy-to-count quantities measure activity or investment rather than outcome: commits, lines changed, control files, model-sync events, and support ratio. They can help explain what an environment is doing; they do not establish that it is producing a return.[25]

6.1.4 Three Core Predictions

Three predictions form the core of the theory. They concern three different ways an engineered environment can help.

First, the environment can make additional agentic capacity more productive rather than merely faster. Second, it can give a reasoner a better surface over which to work, reducing how much of the system must be reconstructed for each task. Third, it can preserve useful engineering judgment so that later work inherits it rather than purchasing the same reasoning again.

These effects are related, but they are not the same. Environment fit concerns whether capacity can be productively absorbed. Representation leverage concerns what the reasoner must reconstruct. Engineering capital concerns what later work gets to inherit.

Environment fit moderates velocity. Increasing agentic capacity should produce more durable throughput when the relevant engineered environment fits the work, and more churn, escaped failure, or repeated intervention when it does not. The effect of velocity therefore depends on whether the environment provides the representations, evidence, authority, and coherence the work requires.

Representation creates leverage. A task-relevant, trustworthy model should reduce the expected cost of reaching an acceptable realization, or increase the scale of work completed at matched quality, by reducing reconstruction and exposing consequential distinctions at a more tractable abstraction. The advantage should matter most as the required reasoning state grows. The claim fails if models impose maintenance and interpretation costs without extending reasoning reach or preserving quality.

Engineering capital amortizes judgment. Where useful engineering knowledge or recurring judgment becomes durable structure, later work over that surface should require less reconstruction, repeated judgment, or rework, and should carry less risk. The prediction includes its own boundary: the return lasts only while the asset remains fit. Stale models, obsolete validators, conflicting controls, and procedures that outlive their problem should lose value and can eventually impose net cost.

[25] The activity-versus-outcome gap already shows up in industry data. A 2026 benchmark of 83,000 developers across 253 organizations[68] reports large merge-rate gains among heavy AI users, yet finds yield falling among the heaviest even as merge rates roughly doubled; an industry analyst reads the same period as rising AI spend without proportionate shipping velocity, the difficulty having moved from comprehension to confidence[80]. Activity is not the same as durable progress.

Each claim can fail. The environment may have little moderating effect; explicit models may add cost without extending reasoning reach; accumulated structure may cost as much as the judgment it was meant to retire. These are the theory's core propositions. Section 6.2 derives more specific predictions from them.

The reasoning-horizon proposition

Why should representation create leverage in the first place?

Consider what happens when an engineer enters an unfamiliar system to answer a seemingly simple question: *Can this service depend on that one?* The answer may be distributed across source files, configuration, dependency injection, deployment descriptors, documentation, and conventions that exist only in the history of the system. Before reasoning about the dependency, the engineer must first reconstruct enough of the system to know what the dependency means.

An agent faces the same problem. A larger context window can postpone the limit, and better retrieval can bring useful evidence closer, but neither changes the underlying fact that implementation contains far more detail than most engineering questions require. A useful model changes the problem by presenting the consequential structure directly. This motivates the reasoning-horizon proposition.

> **Reasoning-Horizon Proposition.** *Large software systems contain more potentially relevant state than any finite reasoner can keep active at once. Task-relevant models can extend the effective reasoning horizon by replacing implementation detail with semantically richer representations of the properties under consideration. The gain holds only while the representation is relevant, sufficiently faithful to the relation it claims, and cheaper to use than reconstructing the same knowledge from lower-level artifacts.*

A useful representation can shorten the reasoning path, not merely make each reasoning step easier. Without an explicit model, an agent may first have to reconstruct the relevant architecture, state, dependencies, obligations, or other relationships from lower-level artifacts before it can reason about the engineering question itself. When the environment already supplies a trustworthy, task-relevant model, the agent can skip much of that reconstruction and reason directly over the consequential structure. In this sense, some of the reasoning has already been performed and preserved in the environment rather than purchased again on each attempt.

The effect is analogous to using a map rather than repeatedly reconstructing a city from street-level observations. The map does not contain everything in the city, nor should it. Its value comes from preserving the relationships needed for a particular class of questions while suppressing detail that would otherwise have to be rediscovered. An engineering model provides the same kind of leverage when it preserves the distinctions relevant to the work.

Agentic systems do not create this scale problem. Human engineers have always used abstractions because no engineer can keep the implementation of a Linux-scale system in active view. Commodity intelligence changes the economics and frequency of the problem: larger delegated tasks and greater implementation volume increase the return on representations that preserve useful reasoning across episodes.

These representations do more than store facts. State machines expose transitions, dependency graphs expose relationships, contracts expose obligations, and assurance arguments relate claims to evidence. They put consequential structure into a form the reasoner can use directly. Part II offered six useful classes of such model, not a proof that they are necessary, sufficient, or minimal. Because representation creates leverage and can expand the surface available to Alignment, choosing, reducing, and even learning representations is itself an engineering problem. The principle is simple: **a representation is worthwhile when the reasoning it saves is worth more than the cost of building and carrying it.**

Representation leverage can also originate in the other direction. Part IV showed the practical version of this move: use implementation signals and commodity intelligence to help recover latent models from an existing system. Repeated structures can reveal a domain concept the implementation already embodies but has never named; bottom-up analysis can induce an abstraction that later humans, agents, and mechanisms use directly. The representation is new as an explicit engineering artifact, even when the structure it captures was already latent in the system.

The stronger possibility is representation innovation. A reasoner need not be limited to recovering an abstraction that engineers already know how to name. Operating across implementations, histories, measurements, failures, and existing models, commodity intelligence may propose a different decomposition, relation, state space, or other representation under which an engineering question becomes tractable. The innovation is not that the machine draws a novel diagram. It is that the representation exposes consequential structure that existing representations did not make practically available for reasoning or analysis.

This possibility extends the reasoning-horizon proposition in both directions. Human-designed representations can compress engineering knowledge into forms that let machines reason beyond implementation detail. Machine-induced representations can make latent structure explicit. Machine-innovated representations may go further, exposing relationships that extend what humans and later machines can practically reason about.

Discovery is not authority. Neither induction nor innovation makes a representation trustworthy. Like any model in Part II, a machine-proposed representation claims a relationship to the system and must earn the trust placed in that claim. A recovered model can be checked against the implementation it purports to summarize; a genuinely novel abstraction poses the harder question of what evidence establishes that it preserves the distinctions required by its intended engineering use. The stronger the reasoning, assurance, or authority built on the representation, the stronger that evidence must be.

Representation engineering is therefore potentially bidirectional. Humans can construct abstractions that extend machine reasoning; machines can recover abstractions implicit in

human-built systems; and machines may eventually help invent abstractions that extend the effective reasoning horizon of both. The research agenda returns to how representations can be selected, induced, innovated, validated, learned, and reduced.

The engineering-capital proposition

Representation leverage explains how structure can make one episode of reasoning easier. The next question is what happens across episodes.

If engineers repeatedly reconstruct the same architectural fact, rediscover the same failure mode, or make the same review judgment, the organization repeatedly pays for essentially the same reasoning. But some judgments can be preserved. A dependency rule can become a validator. An architectural distinction can become a model. An operational lesson can become a procedure. Once that happens, later work begins from a different starting point.

This is the sense in which MAGE uses the term engineering capital: prior engineering work leaves behind productive capacity that future work can inherit.

> **Engineering-Capital Proposition.** *Durable engineering structures constitute capital when future work inherits useful capacity from them: less reconstruction, less repeated judgment, earlier or stronger evidence, safer action, or cheaper recovery. Models, mechanisms, architectures, procedures, and other engineering assets can all qualify. Their returns are local to the surfaces where later work inherits that capacity, and last only while the assets remain fit.*

The metaphor is capital rather than memory. A document that merely records an old decision may preserve information; an engineering asset becomes capital when it changes the cost, capability, or risk of future work. Like other capital, it must produce a return to justify carrying it, and it can depreciate when the environment changes.

That inheritance need not depend on the continued presence of the engineer who supplied the original judgment. When an architectural distinction becomes a trustworthy model, a review judgment becomes a validator, or an operational lesson becomes a reliable procedure, some of the originating engineer's productive capacity has moved into the environment. A later engineer or agent can benefit from the result without reconstructing the reasoning that produced it.

The transfer is necessarily partial. Durable structure carries only the judgment it successfully represents or mechanizes. When requirements change, a new failure appears, or the representation itself becomes inadequate, engineering judgment is needed again. Engineering capital therefore does not eliminate expertise; it changes how often the organization must pay again for expert judgment on the same question.

The analogy to technical debt is intentional. Debt makes future change more expensive; capital makes future change more productive. Neither status is permanent. Capital depreciates as systems, obligations, and tools change. A validator can become noise, a model can drift, and a once-useful constraint can begin blocking legitimate work. Governance

therefore includes maintenance, reconciliation, and retirement. **Accumulation is not the objective; productive capacity is.**

The analogy also connects MAGE to an emerging account of debt in AI-assisted software. Storey argues that technical debt in the implementation is only one of three interacting forms of debt. Cognitive debt accumulates when a team's shared understanding of the system erodes; intent debt accumulates when goals, constraints, and rationale are poorly externalized or maintained. Generative AI can accelerate all three by increasing implementation faster than teams can understand it or preserve why it exists.[81]

A Governed Engineering Environment acts on each layer, although not in the same way. Architectural constraints, validators, models, and other durable structures can reduce technical debt by keeping later realizations inside selected structural boundaries. Explicit requirements, decisions, models, and obligations counter intent debt by preserving consequential knowledge outside any one person or reasoning episode. Cognitive debt is different: MAGE does not require every engineer to reconstruct every generated implementation in detail. It instead seeks to preserve the understanding needed for consequential engineering decisions in representations and evidence that humans and agents can inspect, challenge, and reuse.

This does not make the three debts disappear. A stale model can preserve obsolete intent; an incoherent collection of mechanisms can itself become technical burden; and no representation can substitute for understanding that was never captured in the first place. The point is narrower. Engineering capital gives selected knowledge, intent, and judgment a durable place to live, while governance determines which of those assets remain trustworthy enough for later work to inherit.

6.1.5 Modeling and Alignment Are Distinct—and Compound

The previous propositions explain why durable structure can help, but they combine two effects that MAGE deliberately keeps separate. An environment can help the reasoner reach a satisfactory answer, and it can independently control whether an unsatisfactory answer acquires consequence. MAGE calls the first family of interventions Modeling and the second Alignment.

Modeling and Alignment are therefore distinct engineering activities. Modeling changes the representation through which engineers and agents reason; Alignment gives selected obligations authority over realization. Either can produce value without the other.

A model can reduce reconstruction and make satisfactory realization easier without enforcing anything. Conversely, an Alignment mechanism can govern an obligation even when the producing agent never sees the representation or rule from which the mechanism decides. A type can exclude an invalid state, a sandbox can deny a capability, and a validator can reject a forbidden dependency even if the agent repeatedly proposes one.

Consider a simple architectural obligation: service A may not depend directly on service B.

One engineering problem is helping an agent understand the relevant architectural structure well enough not to propose the forbidden dependency. A useful structural model may make acceptable realizations easier to produce. A different engineering problem is ensuring that the forbidden dependency cannot enter the system even when the producer gets that relationship wrong. An independent admission check can provide that authority.

Cost and failure, separately

The distinction can be made precise by considering cost and failure separately.

Let R denote the representation available to a reasoner. Let p_R be the probability that one attempt produces a realization satisfying a particular obligation, and let c_R be the cost of producing that attempt. A better representation can increase p_R, decrease c_R, or both. Under the simple retry model from Part V, the expected production cost before an acceptable candidate appears approaches

$$\frac{c_R}{p_R}$$

This is one role of Modeling: make acceptable realization cheaper to find.

Now let A denote an independent admission mechanism. In the strongest region of Alignment, suppose A is a sound deterministic predicate for the covered obligation: an unacceptable candidate cannot pass. Then

$$P(\text{covered failure admitted} \mid A) = 0$$

That guarantee does not require the producer to reason well about the obligation. The agent can be shown no structural model at all, repeatedly propose forbidden dependencies, and still be prevented from merging them. This is one role of Alignment: make consequence independent of whether the producing reasoner got the covered judgment right.

But assurance without useful representation can be expensive. If each rejected candidate is regenerated until the sound gate accepts one, and validation costs c_A per attempt, then under the same illustrative assumptions,

$$E[C_{\text{admit}} \mid R, A] = \frac{c_R + c_A}{p_R}$$

Alignment can therefore provide the same admission guarantee over two different reasoning surfaces while producing substantially different realization costs.

Return to the architectural example. Suppose the rule is checked perfectly at merge time. Hiding the structural model from the agent does not weaken that gate: forbidden changes still cannot merge. But the agent may repeatedly reconstruct the relevant architectural relationship incorrectly and collide with the gate. Expose an appropriate structural model to the same agent and the guarantee need not change at all. What changes is the cost of reaching a candidate the gate will accept.

Alignment protects the boundary; Modeling reduces how often work collides with it.

The four combinations therefore have different engineering meanings (Table 6.1-2).

Table 6.1-2: Modeling and Alignment are separable levers. *Modeling changes the economics and tractability of reasoning; Alignment changes the authority of the result. The cells describe their primary contributions, not an exhaustive account.*

Modeling ↓ / Alignment →	No Alignment	Alignment
No Modeling	Repeated reasoning from a poor surface; residual failure can escape	Covered failure cannot escape, but rejection and regeneration may be expensive
Modeling	Satisfactory realization becomes cheaper or more likely, but residual failure can still escape	Satisfactory realization becomes cheaper while covered failure remains independently blocked

The table is schematic rather than exhaustive. Modeling may itself improve reliability, and Alignment mechanisms may be statistical rather than deterministic. The point is that their primary contributions are separable.

Software engineering changes the problem

The analysis so far asks what happens when repeated attempts face the same realization problem. Part V used this simplification to isolate repeated probabilistic exposure: if consequential judgments have some probability of success and successive exposures are treated as independent, their joint probability of success falls as the number of exposures grows. The retry model likewise holds the probability and cost of an attempt fixed while asking what additional attempts purchase. These models are useful because they isolate the effects of exposure and repetition.

Software engineering requires a different model because one realization changes the conditions inherited by the next. Winters et al. characterize software engineering as "programming integrated over time": software is developed, modified, and maintained as both the system and its environment change.[82] One realization therefore becomes part of the architecture, conventions, representations, tests, and other structure inherited by subsequent work. A good change can make later work easier to reason about; a locally acceptable but structurally poor change can make it harder.

For this question, independence omits the relationship we need to represent. Let the probability of satisfactory realization at step i be

$$p_i = p(T_i, M, R_i, H, E_i)$$

where E_i is the engineering environment inherited at that step. The resulting realization a_i then helps determine the environment available to subsequent work:

$$E_{i+1} = f(E_i, a_i)$$

This coupling creates path dependence. A locally acceptable change can increase architectural irregularity, obscure system structure, introduce inconsistent conventions, or

otherwise make later work harder to reason about. Later agents then operate over a worse reasoning surface, potentially reducing the probability of satisfactory realization. Architectural decay can therefore become self-reinforcing: poor structure makes good realization harder, and subsequent realizations can degrade the structure further.

The reverse trajectory is possible as well. A realization that improves representations, architecture, constraints, or other engineering capital can make later work easier to reason about and govern. The environment then carries useful structure forward rather than forcing each subsequent reasoner to reconstruct it.

The independent model and the dynamic model therefore answer different questions. The first isolates the effect of repeated probabilistic exposure under fixed conditions. The second preserves the relationship between successive realizations and the environments they create. These coupled trajectories are the mathematical counterpart of the feedback dynamics in Figure 6.1-1.

Modeling can also enable Alignment

The two principles interact in a second way. Some obligations can already be checked mechanically, and Alignment can govern them directly. Others are semantic only because the relevant state must be reconstructed from lower-level implementation. Suppose the architecture requires every route to a remediation service to cross quota authorization: reasoning from implementation means reconstructing routes from handlers, middleware, configuration, and dependency injection, but representing the permitted routes explicitly turns the obligation into a graph question—does every allowed path cross the quota boundary? The property did not become less consequential; Modeling changed the object over which it is decided, turning a repeated semantic judgment into a checkable property.

There is a broader pattern here. Some engineering questions are hard because the answer is intrinsically judgmental. Others are hard because the information needed to answer them is buried in the wrong representation. These cases look similar when an engineer or agent is staring at source code: both require reasoning. But they have very different engineering possibilities. If the underlying property is genuinely judgmental, better representation may help without eliminating the judgment. If the property becomes mechanically decidable once the relevant structure is made explicit, Modeling has done something stronger: it has moved a decision from repeated inference into repeatable machinery. The boundary between those cases matters enough to name.

Call the boundary between these cases the **determinization frontier**: judgment on one side must still be supplied per instance; judgment on the other can be carried repeatably by the environment. Alignment can mechanize an obligation that is already decidable. Modeling can move the frontier by changing the representation until a previously reconstructed property becomes decidable. Figure 6.1-2 draws the two routes. The frontier is not a wall around what agents may do; it is a shoreline between judgments the environment must repeatedly purchase and judgments it can carry forward itself.

Moving the frontier reduces the probabilistic surface: fewer consequential judgments depend on the agent getting them right. It does not make the agent deterministic, and it does not require eliminating implementation freedom. The environment can tightly govern what must be true while leaving the agent free to choose among many acceptable realizations.

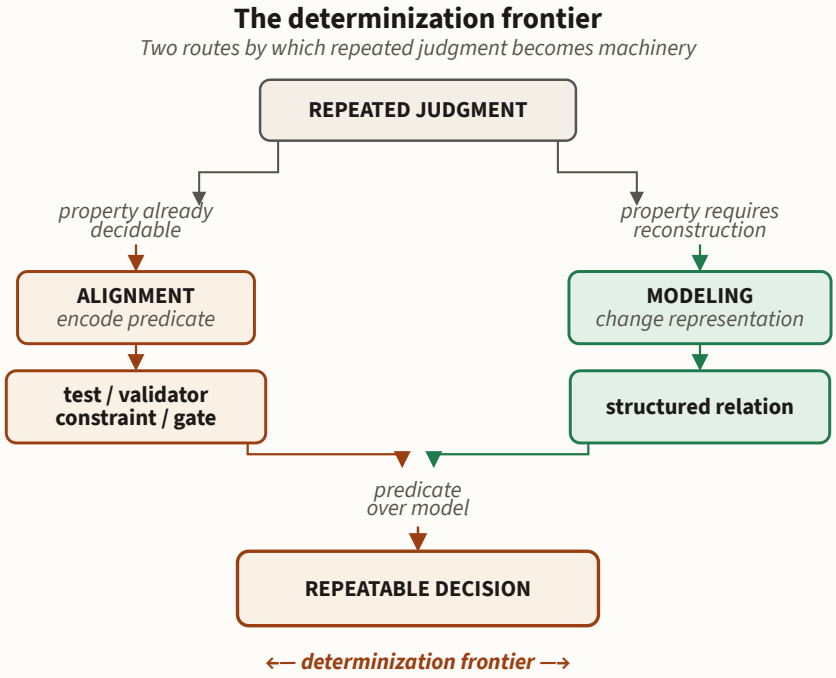


Figure 6.1-2: The Determinization Frontier. *The frontier separates judgment that must be supplied per instance from judgment the environment can carry repeatably. Alignment can directly bind a property already decidable over an action or artifact. Modeling can change the reasoning surface so that a property requiring semantic reconstruction becomes a repeatable predicate that Alignment can bind. Modeling may also reduce realization cost even where Alignment was already possible.*

The frontier can move incrementally: from prose judgment to an explicit invariant; from an invariant to exhaustive checking over a bounded state model; or, for temporal claims, to a specification over executions.

Moving the frontier does not mean closing off realization. The frontier concerns obligations—which judgments the environment can carry repeatably—while the acceptable ways to satisfy them remain degrees of freedom the agent is free to explore.

At the limit, a sufficiently complete realization model leaves little consequential freedom: realization becomes a transformation problem, and deterministic generation may be preferable to an agent. This limiting case connects MAGE to conventional model-based engineering; Section 7.2 returns to the relationship. Most software systems occupy a different point in the design space. Engineering specifies what must be true; many implementation choices remain acceptable. Commodity intelligence makes it economical to let autonomous realization choose among them.

The joint engineering problem

We can now put the pieces back together.

Modeling can make satisfactory realization easier to find. Alignment can prevent selected failures from acquiring consequence. Better environments can preserve useful judgment for later work, but every representation and mechanism also costs something to build, maintain, reconcile, and carry.

The objective is therefore not to maximize Modeling, maximize Alignment, or minimize the probabilistic surface at any cost. It is to engineer an economical division of labor among the reasoner, the environment, and the human judgment that remains.

For a task and its obligations, an engineer chooses both a representation R and an Alignment strategy A:

$$(R^*, A^*) = \arg\min_{R,A} \left[C_{\text{carry}}(R, A) + E[C_{\text{realize}} \mid R, A] + E[L_{\text{failure}} \mid R, A] \right]$$

The expression is deliberately schematic. Modeling principally changes the cost and difficulty of satisfactory realization; Alignment principally changes the probability and consequence of unacceptable realization; both impose construction and carrying costs. In high-assurance settings the problem may instead be stated as minimizing total cost subject to an assurance requirement,

$$P(\text{covered failure escapes} \mid R, A) \leq \epsilon$$

For an adequately implemented deterministic gate over a decidable obligation, ϵ can be zero for that covered property even while realization remains probabilistic.

Part V supplied complementary evidence: longitudinal depth from the originating case and comparative variation from independent industrial reconstructions. Neither supplies population-level effect sizes or causal estimates. The theory above is the proposed explanation of those observations; Section 6.2 states predictions that require stronger external tests.

The theory now gives us claims that can be tested. The next chapter makes those predictions explicit; the chapter after that asks where we should expect them to hold.

6.2 What the Theory Predicts

If the theory is right, certain consequences should follow. This chapter states them generally—what we should expect to observe wherever MAGE applies. It states the predictions first; the next chapter asks where they should be expected to hold.

The predictions fall into three families: trustworthy representations, adaptive governance, and engineering economics. They are consequences derived from the theory, not findings of this book. Each is stated here as a named hypothesis; the research agenda that closes this Part turns those hypotheses into experimental designs.

6.2.1 Trustworthy Representations

Brownfield software raises the first problem immediately: can a useful engineering model be recovered from a system that was never built around one? An agent can extract structure from implementation, but description is not obligation. A model that perfectly mirrors defective behavior can simply canonize the defect. Useful induction must therefore distinguish what the system **does** from what an independently justified engineering claim says it **should** do.

Once a model exists, maintenance becomes a second problem. "Keep the model equal to the code" is too crude. A model of permitted dependencies should disagree with code that violates them. A requirements model may intentionally describe an intended state the current implementation has not reached. The engineering problem is to state and maintain the **correspondence each model claims**—for architecture, behavior, process, invariants, lineage, or other views—cheaply enough to expose meaningful divergence near the change that created it.

The third problem is quality. A model can be synchronized and still be useless. Test suites provide the familiar warning: an oracle can faithfully encode the implementation rather than the requirement, or cover one narrow realization while missing the property that matters.[26] Models need analogous evaluation. The field must measure not only fidelity to artifacts, but whether the representation captures the obligations and relations required by the engineering decisions made through it.

H1 — Representation leverage. For tasks whose required reasoning state exceeds comfortable implementation-level inspection, a task-relevant and trustworthy model will reduce reasoning and reconstruction cost per acceptable realization, or increase task scale at matched quality, relative to reasoning from lower-level artifacts alone. The advantage should increase as the required reasoning state grows.

[26] Oracle overfitting is well established. A steward of a widely watched coding benchmark audited the problems a strong model kept failing and found many carried flawed tests — too narrow, or checking behavior the task never described — alongside contamination that let exposure substitute for reasoning[83]. The lesson predates the current era: automation that optimizes against a visible oracle satisfies the oracle while silently breaking what it does not check[84].

H2 — Representation integrity. The benefits predicted by H1 will decline or reverse when models are stale, irrelevant, or poorly matched to the task. Measures of model correspondence and obligation coverage should therefore predict later churn or defect escape beyond conventional implementation-level metrics.

H3 — Representation-induced determinization. For some system-level obligations, replacing raw implementation as the reasoning surface with an explicit representation of the relevant state or relation will make a repeatable predicate feasible where per-change evaluation over the implementation otherwise requires semantic reconstruction.

H4 — Representation induction and innovation. Commodity intelligence can contribute useful engineering representations, first by inducing latent structure from existing systems and, more strongly, by proposing representations that make consequential engineering questions tractable in ways the existing representational repertoire does not. Where those representations can be independently validated and economically maintained, they should extend the effective reasoning horizon of later agents and human engineers.

6.2.2 Adaptive Governance

Governance conversion remains primarily an act of judgment. A failure occurs; someone decides whether it is local or structural; someone decides whether future work should inherit a model, constraint, validator, architectural change, procedure—or nothing at all. Agents can already implement many of those responses. The research frontier is deciding **which investment is warranted, what its cheapest adequate form is, and when judgment should remain where it is**.

A small example makes the distinction concrete. Suppose agents repeatedly introduce direct database access from services that are expected to use a repository layer. Repairing each occurrence fixes the local change but leaves the next agent to make the same mistake. The team can instead represent the permitted access paths and add an architectural check that rejects violations. If later work stops reproducing the failure class and requires less review to enforce the boundary, the environment has learned something useful. If legitimate exceptions are common and engineers spend more time negotiating the check than repairing the original failures, the conversion has not produced useful engineering capital.

An adaptive environment also has to subtract. A system that automatically responds to every anomaly with another lint or gate will manufacture bureaucracy faster than capital. Useful adaptation must add, strengthen, reconcile, and retire. The obligation/mechanism census from Part III provides one observable surface: important obligations with inadequate coverage are gaps; mechanisms that no longer trace to a purpose are candidate orphans. A useful adaptive system would reduce consequential gaps without simply increasing the number of mechanisms.

Organizations make the problem harder. The person who recognizes a failure may not own the shared infrastructure capable of retiring the class. A good diagnosis can therefore die as a local patch. The relevant organizational variable is not whether authority resides in one

architect, but whether a functioning path connects **diagnosis to shared environmental change**.

H5 — Mechanized assurance. For obligations that are sufficiently decidable and adequately covered, repeatable independent mechanisms will reduce defect escape at matched realized throughput relative to assurance that depends on producer compliance or proportional per-change human attention. For sound admission mechanisms over mechanically decidable obligations, the covered failure class should be excluded from admitted work even when producer reliability varies.

H6 — Governance conversion. For recurring failure classes, structural interventions targeted at the class will reduce subsequent recurrence or repeated human attention relative to local repair alone, net of the intervention's carrying cost. The relationship will become non-monotonic when controls accumulate beyond their useful fit.

H7 — Learning propagation. Engineering lessons will reach later relevant work faster and more consistently, and will be less dependent on continued access to the people who first supplied them, when diagnosis is connected to authority over shared, machine-actionable environmental structure than when the lesson remains local or must propagate primarily through interpersonal communication. Personnel turnover provides one natural test: where consequential judgment has become trustworthy shared structure, later work should lose less effective engineering capability when the originating engineer is absent.

6.2.3 Engineering Economics

The economic predictions are the least settled of the three families. Representation leverage and governance conversion can be observed directly within an engineered environment; estimating their economic return requires a counterfactual—what the same work would have cost without the asset, under comparable intelligence, quality, and task conditions. Model capability is changing at the same time, token prices are moving, organizations value human attention differently, and engineering assets depreciate. The hypotheses below therefore define quantities worth measuring, not coefficients MAGE claims to know.

The strongest economic claim in MAGE is easy to state and difficult to price: durable engineering structure can make later work cheaper or safer. The DocAble support ratio shows where source accumulated, not whether that stock was valuable. Nor will removing one artifact necessarily reveal its contribution when the same knowledge survives across models, tests, code, and documentation. The environment's return may not decompose artifact by artifact.

The quantities that matter are therefore returns and carrying costs. How much reconstruction, review, rework, defect risk, or recovery cost does an asset retire? How much compute, latency, maintenance, context, false-positive burden, and coordination does it impose? How does that balance change as the system moves and the asset depreciates?

One implication is that effective engineering capability may partly belong to the **environment**, not only to the foundation model or the people currently operating it. A weaker model

operating over a richly represented, well-instrumented environment may outperform a stronger model over an opaque repository because the environment has performed some of the reasoning before the interaction begins. Whether environment quality substitutes for model capability, complements it, or does neither is an empirical question.

This suggests another way to operationalize engineering capital. Instead of trying to price each model, validator, or procedure separately, treat a commodity agent as a standardized consumer of the environment. Hold the reasoning engine and task family fixed, vary the engineered environment, and measure how much productive capability the environment contributes: task scale reached at matched quality, reconstruction cost, intervention burden, defect escape, and durable throughput. The measured difference is not a complete valuation of engineering capital, but it makes otherwise intangible engineering structure empirically visible.

H8 — Environment fit. Increasing agentic capacity will produce more durable throughput and lower human attention in environments that adequately represent and govern the work. Where the environment fits poorly, the same additional capacity will produce more churn, intervention, or defect escape. The research agenda takes up how Modeling and Alignment separate, combine, and route across task families.

Outer-loop corollary. If commodity intelligence primarily accelerates source-code production, its benefits should remain concentrated in the coding slice of software work and increasingly encounter downstream limits in review, testing, integration, and coordination. If MAGE's account is right, stronger governed engineering environments should allow the same intelligence to assume increasing portions of the surrounding realization loop —debugging, test construction, refactoring, integration, migration, and other bounded engineering activities—while preserving durable throughput without proportionate growth in human-attention burden.

H9 — Engineering-capital return and depreciation. Useful models, mechanisms, architectures, and procedures will produce measurable reductions in future reconstruction, human attention, rework, or risk over the surfaces they cover; those returns will decline as the assets lose fit, and environments that reconcile or retire depreciated assets will outperform otherwise comparable environments that merely accumulate machinery.

The nine hypotheses divide across the three families, each paired with the core quantity a study would have to measure. Table 6.2-1 collects them.

Table 6.2-1: The nine hypotheses. *MAGE's research program comprises nine named hypotheses in three families — trustworthy representations, adaptive governance, and engineering economics — each with the core quantity a study would measure.*

Family	Hypothesis	Core quantity
Representation	H1 Representation leverage	reasoning/context cost at matched quality
Representation	H2 Representation integrity	drift/coverage → churn or escape

Family	Hypothesis	Core quantity
Representation	H3 Representation-induced determinization	semantic judgment → repeatable predicate
Representation	H4 Representation induction and innovation	reasoning reach / newly tractable questions
Governance	H5 Mechanized assurance	defect escape vs throughput
Governance	H6 Governance conversion	class recurrence + carrying cost
Governance	H7 Learning propagation	propagation latency/consistency
Economics	H8 Environment fit	capacity × environment interaction
Economics	H9 Capital return/depreciation	return and carrying cost over time

The three families state what the theory predicts. The next chapter bounds where those predictions should hold; the research agenda that follows turns them into studies that could confirm or refute the account.

6.3 Scope Conditions

MAGE has scope conditions because it imposes costs: models have to be built and maintained; controls consume compute and attention; synchronization can fail; every new mechanism adds to what the environment must carry. Those costs are justified only where the resulting structure buys something back.

The right unit of analysis is usually **an obligation or engineering surface, not an industry label**. The same scientific system may have highly governable build and reproducibility machinery around a kernel whose scientific validity remains a matter of expert judgment. A startup may have little reason to model its architecture comprehensively while still needing a hard security boundary around payments. MAGE therefore applies unevenly even within one repository.

6.3.1 Return, Governability, and Authority

Whether MAGE applies to a given engineering surface comes down to three questions:

1. **Will it earn a return?** Is there enough recurrence or consequence to repay the investment?
2. **Can the obligation be governed adequately?** Can the property be represented, evidenced, and evaluated well enough?
3. **Can the organization act on what it learns?** Is there authority to change the shared environment?

Does the investment earn a return? Recurrence is the usual source of return. If the same architectural knowledge, review judgment, or failure class will be encountered hundreds of times, externalizing it once can avoid purchasing the same reasoning repeatedly. But recurrence is not the only justification. A rare obligation with catastrophic consequences may warrant extensive modeling and assurance because preventing a single failure can repay the investment. The economic question is therefore broader: **does the expected reduction in future reasoning, risk, or recovery cost justify the construction and carrying cost of the asset?**

Can the obligation be governed adequately? Some properties can be represented precisely and settled mechanically: a schema conforms, a dependency is permitted, an invariant holds. Others admit statistical evidence but not a definitive predicate. Still others depend substantially on aesthetic, scientific, organizational, or social judgment. Governability is the degree to which an important obligation can be made explicit, supplied with relevant evidence, and evaluated with adequate reliability and acceptable cost. Deterministic checking is its strongest region, not its definition.

The residual probabilistic surface is therefore not necessarily technical debt. Some judgment belongs there. The objective is not to minimize it blindly, but to avoid repeatedly purchasing expensive probabilistic reasoning or residual risk where a cheaper, adequate, and trustworthy representation or mechanism can do better.

One organizational condition determines whether lessons become shared assets: **authority**. Someone who discovers a structural problem must have a functioning path to the architecture, models, controls, or shared infrastructure that could address it. Authority need not reside in one person. It has to connect diagnosis to the environment. Without that path, the organization can understand a failure perfectly and still patch it locally forever.

These are theoretical scope conditions informed by the observations in Part V, not population estimates.

6.3.2 The Expanding Frontier of Explicit Engineering

MAGE's scope is not fixed, because the economics and feasibility of explicit engineering are changing. Two questions determine where investment pays. The first is economic: how much future reasoning, risk, or recovery cost can an explicit engineering asset avoid relative to its construction and carrying cost? The second is governability: how much of the relevant obligation can be represented, supplied with evidence, and evaluated adequately enough to receive authority? These questions interact, but neither determines the other.

The distinction matters because consequence does not imply governability. Some consequential obligations admit precise models and strong evaluation: structural loads, resource bounds, protocol states, or dependency relations may eventually support calculation, exhaustive analysis, or proof. Other obligations remain substantially qualitative even when an organization has strong reason — or a legal duty — to evaluate them carefully. Aesthetic quality, scientific validity, organizational fitness, and social acceptability can justify substantial engineering investment without ever becoming completely reducible to predicates.

Part IV gave the practical version of this distinction. A writing style guide can preserve knowledge about good writing even though good writing has no definitive predicate. Some narrower obligations, such as spelling or paragraph length, can receive deterministic authority; others, such as whether an introduction prepares the reader for what follows, may admit useful probabilistic evaluation. The DocAble residency model showed the boundary moving in the other direction: repeated empirical judgment about memory behavior became a quantitative invariant once a better representation exposed the relevant relation. Modeling can therefore improve judgment without eliminating it, and it can sometimes make stronger authority feasible.

Inset — Hard Obligations, Judgment Calls ENGINEERING PRACTICE

An obligation does not have to be mechanically checkable to be a real engineering obligation. Consider a historic building. The Secretary of the Interior's Standards for Rehabilitation require certified projects to preserve historic character, and new work must be compatible with the historic building in its massing, size, scale, and architectural features. Replacement features may need to match the original design, color, texture, and other visual qualities.[85] These are real requirements that can determine whether a project is approved, but there is no complete test for compatible with the

historic character. Engineers and architects can measure dimensions, identify materials, record features that must be preserved, compare photographs and drawings, and use precedent and design rules. These things make the judgment better informed and more consistent. They do not remove the judgment.

Software has the same characteristic. The Department of Justice's 2024 accessibility rule generally requires state and local governments to make their web and mobile content conform to WCAG 2.1 Level AA, including conventional electronic documents where the criteria apply.[86] Many resulting requirements can be checked mechanically, but the rule also permits an alternative technique when it provides "substantially equivalent or greater accessibility and usability." Whether it does so can depend on the information and functionality preserved, the users and tasks involved, and the barriers the alternative introduces. DocAble sits on this boundary. It can mechanically check many properties that its document models make observable and use probabilistic models for semantic work such as interpreting figures and generating descriptions. But neither proves that every blind student will receive a substantially equivalent experience from every remediated document. Some judgment remains.

The historic-building and accessibility examples make the same point. An obligation can be important, explicit, and legally binding without having a complete formal test. Engineering can still make the judgment better: make the obligation explicit, mechanize the parts that can be checked, and gather good evidence for the rest. The boundary can also move. Better models and measurements can turn judgments that once required an expert into properties the environment can check; better GenAI may make us willing to entrust increasingly difficult judgments to an artificial expert. Here, expert does not necessarily mean human: it means a reasoner capable of making the judgment well enough for the consequence at stake. The goal is therefore not to eliminate judgment, but to give each obligation the strongest form of evaluation and authority that the available evidence can support.

These examples expose the problem with treating explicit engineering as a ladder whose upper end is full formalization. Full formal methods occupy an important region of the design space: obligations represented with semantics precise enough for machines to derive, search, transform, or verify solutions. Program synthesis, compiler optimization, model-driven engineering, configuration tuning, and formal methods all exploit such representations. Their historical reach has also been economic, because specifications, models, proofs, generators, correspondence machinery, and verification infrastructure cost engineering effort. Commodity intelligence can lower those costs by assisting with specification, invariant discovery, model construction, proof construction, and translation between engineering intent and formal artifacts.

But full formalization is a region of the design space, not its endpoint. Where governability remains limited, additional consequence can justify more investment without making complete formalization feasible. A design organization may invest heavily in models, precedent,

examples, structured critique, evaluation procedures, and expert review. An accessibility system may mechanize dozens of standards-derived obligations while retaining semantic and user-experience judgments that no checker can settle. Greater consequence can justify more engineering in either case; it does not make the underlying obligation more governable. A scientific organization may make provenance, units, workflow, and reproducibility authoritative while leaving scientific interpretation to expert judgment. A bridge project may formally analyze structural obligations while using substantially different evidence and review for its aesthetic relationship to a historic or scenic setting. The right unit remains the obligation or engineering surface, not the industry or artifact.

The economic question and the governability question therefore define two dimensions of investment. The value of explicit engineering asks how much explicit structure the obligation warrants: repeated reasoning cost, consequence, recurrence, and recovery cost can all raise it. Feasible governability asks how much authority is achievable: whether the obligation can be represented, observed, and evaluated with adequate reliability and acceptable cost. High value does not imply high governability. Where both are high, strong formalization and authoritative mechanisms may pay. Where value is high but governability remains limited, substantial investment can instead improve the representations, knowledge, procedures, evidence, and evaluation through which judgment occurs. Figure 6.3-1 sets the two dimensions on one plane.

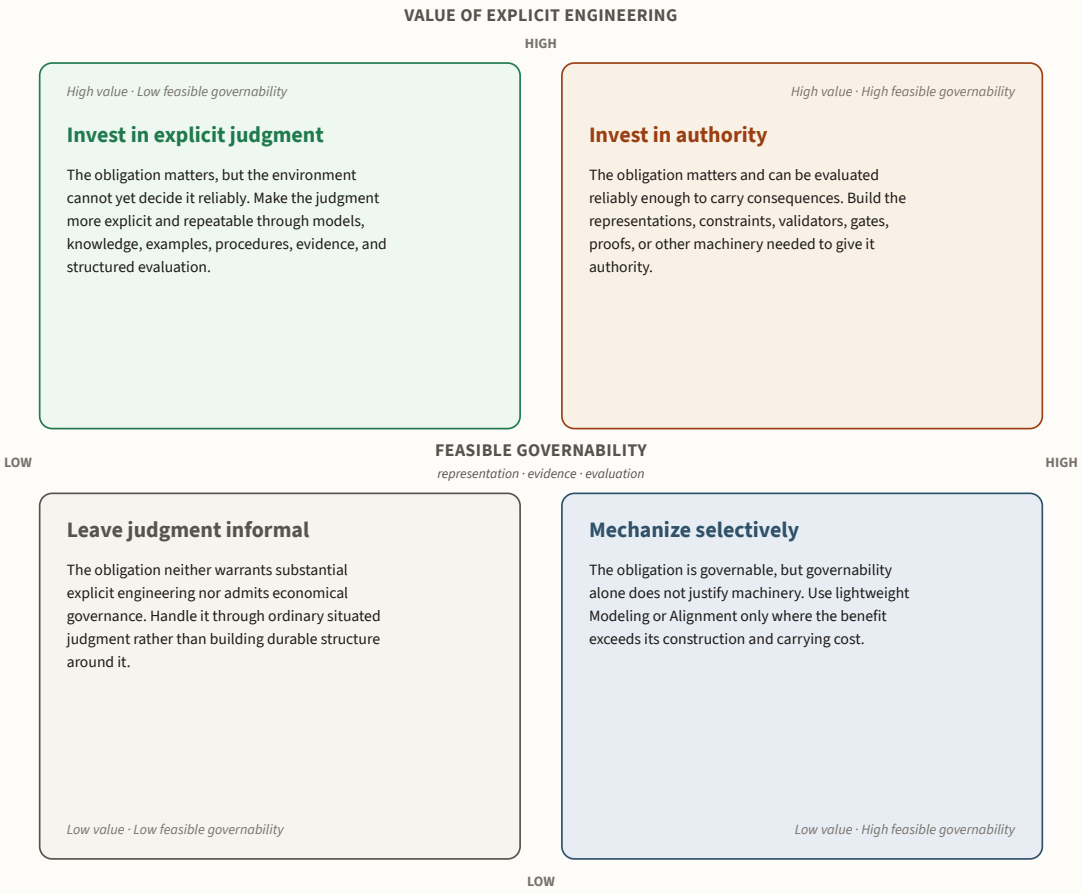


Figure 6.3-1: The economics of explicit engineering. *The value of making an obligation explicit determines how much investment it warrants; feasible governability determines how much of that investment can acquire authority. High-value obligations can justify substantial engineering even when they remain judgmental, while highly governable obligations need not be mechanized when little value would result.*

Commodity intelligence can move work through this design space in two different ways. First, it lowers the cost of constructing and maintaining explicit structure. Models, skills, specifications, validators, proofs, and other engineering assets can therefore repay their

cost at lower levels of recurrence or consequence. This expands the set of obligations for which explicit engineering pays without changing the nature of the obligation itself.

Second, better representation, instrumentation, and evaluation can increase governability. An obligation initially expressed only as expert judgment may yield narrower properties that can be evaluated reliably; a property reconstructed semantically from implementation may become a predicate once an appropriate model exposes it directly. Part IV described the practical version of this movement as giving obligations authority where feasible. The determinization frontier describes its strongest case: Modeling changes the representation until a previously reconstructed judgment becomes mechanically decidable, after which Alignment can carry the decision repeatably.

These movements need not end in the same place. Some questions resist authority because the necessary representation or evidence remains unavailable. Others admit probabilistic evaluation adequate for one consequence but not another. Still others are intrinsically plural or contextual: no amount of cheaper implementation turns good writing, scientific importance, or compatible with the historic character into a single definitive predicate. Explicit engineering still pays when it makes those judgments better represented, better informed, more repeatable, or easier to review.

MAGE therefore concerns the whole design space rather than an upward movement toward formalization. It asks which consequential knowledge should be made explicit, which obligations should acquire authority where feasible, and which remaining degrees of freedom should stay with probabilistic or expert judgment. At one limit, sufficiently complete and governable realization models may make synthesis, proof-directed realization, or deterministic generation preferable to an agent. Elsewhere, the appropriate GEE may deliberately combine explicit models, reusable skills, probabilistic evaluation, authoritative controls, and expert review. The objective is not maximum formalization. It is the economical division of work among the reasoner, the engineered environment, and expert judgment.

MAGE is also applicable at different engineering scales. An individual engineer can model the part of a system under active change and govern its realization. A team can connect those local models to shared architecture, ownership, and operational knowledge. A product can connect models and obligations across engineering surfaces, and an organization can carry selected policies, assurance obligations, and engineering knowledge across products.[27] Purposeful models and explicit correspondence let these scales remain distinct while still informing and constraining one another, so the consequential judgment can stay at the level where it belongs rather than requiring either complete local autonomy or one organization-wide model.

Scale and assurance strength are separate choices. One engineer may use formal verification for a consequential protocol while a large organization relies mostly on design

[27] This interacts with Conway's law, which relates system structure to organizational communication structure[87]. MAGE does not remove that coupling, but explicit models provide additional surfaces—such as architectural relations, security obligations, or assurance claims—around which responsibility can be organized across implementation boundaries.

documents, tests, runtime evidence, and human judgment. Organization-wide adoption does not imply maximal formalization, nor does strong assurance require organization-wide adoption.

The frontier is not a destination

Neither movement through this design space should be confused with a destination for software engineering as a whole. Search works when we know enough about what counts as success. Many consequential engineering questions remain open because the correct shape of the system is itself being discovered: which abstraction matters, which tradeoff users will accept, which failure will dominate in operation, or which requirement the organization actually intends. In those cases implementation is not merely realization; it is an experiment that produces information about the specification. And even a question that is closed today may reopen as users, dependencies, organizations, regulations, workloads, or adversaries change. Software evolution therefore remains an outer loop around increasingly powerful episodes of automated realization[50]. MAGE can turn sufficiently closed engineering questions into search problems when their consequential shape is knowable; it should not get there by pretending that unresolved uncertainty has disappeared.

The corresponding failure mode is premature closure. Explicit does not mean correct. A small probabilistic surface is therefore not itself evidence of good engineering: the environment can determinize the wrong problem. If the organization does not yet know what users need or which tradeoff is acceptable, formalizing one conjecture merely makes the wrong target easier to optimize, and commodity intelligence can make the failure more consequential because it can elaborate the mistaken premise cheaply and at scale.

Engineering capital can also move the frontier within one system. A model built for today's migration can make tomorrow's change searchable; a differential harness can turn a future rewrite into cheap comparison; a validator can convert repeated judgment into evaluation. Engineering capital can therefore enlarge the region of work that can later be delegated safely to search.

MAGE therefore predicts a moving economic boundary, not convergence toward complete specification. Requirements emerge through use, organizations change, adversaries adapt, and systems alter the environments that generate their next requirements. Commodity intelligence can move work through this design space; it does not eliminate the world beyond it. Which engineering surfaces it makes economical to represent and govern more strongly, and how that optimum varies with consequence, organizational context, model capability, and carrying cost, is an empirical question the research agenda takes up.

The implication is that scope should be judged surface by surface, according to recurrence, consequence, representability, and carrying cost—not assigned once to an entire project or domain.

6.3.3 Typical Profiles

The two dimensions produce different profiles even within a single organization or artifact. Table 6.3-1 gives representative cases. These are not maturity levels: movement toward the upper right is neither inevitable nor always desirable. The appropriate profile depends on the return available from explicit engineering and the governability of the particular obligations at issue.

Table 6.3-1: Typical profiles, not domain verdicts. *The return on MAGE varies by surface within a system. These rows identify where the investment commonly pays and where important judgment usually remains.*

Setting	Why investment may pay	Strongly governable surfaces	Likely residual judgment
Large brownfield / services	repeated change and reconstruction	architectural boundaries, API contracts, security boundaries, tests, deployment	requirements and tradeoffs
Long-lived embedded	recurrence + assurance	timing, memory, protocols, interfaces	hardware/system integration
Safety-critical	consequence + recurrence	traceability, invariants, coverage, evidence	certification and acceptance
Greenfield	selective; increases with longevity	high-risk boundaries, build/test infrastructure	product discovery
HPC / scientific	long-lived estate	builds, reproducibility, numerical tolerances, provenance	scientific validity
ML systems	long-lived surrounding software	pipelines, evaluation infrastructure, serving, access	data/model fitness
Design / aesthetic work	repeated practice + consequential stakeholder or regulatory outcomes	dimensions, required elements, materials, provenance, process obligations, selected visual properties	aesthetic quality, composition, contextual fit
Small startup	usually selective	consequential security/data obligations	product search and architecture in flux
One-off / throwaway	usually low reuse	only obligations whose consequence independently justifies it	most of the work

The pattern is economic rather than categorical. Long-lived systems offer repeated opportunities to amortize representation and assurance. High-consequence systems can justify

the same investment even at lower recurrence. Mixed domains divide along individual engineering surfaces: scientific software may support strong governance around reproducibility and deployment while leaving scientific validity with domain experts; ML systems can govern pipelines and serving infrastructure much more strongly than they can govern the question, “Is this learned model good?”

At the opposite extreme, do not build elaborate governance around work whose relevant reasoning will not recur and whose consequences do not justify the investment. Use the cheapest adequate mechanism. A prototype may need only tests around one dangerous boundary. If it becomes a long-lived estate, the economics change before the label does.

6.3.4 Two Ways an Investment Fails

MAGE becomes a bad bargain in two broad ways. **First, the asset never earns a return:** the knowledge is not reused, the risk is too small, or the machinery costs more to maintain than the judgment it replaces. **Second, the important property resists adequate governance:** it cannot be represented, evidenced, or evaluated strongly enough to justify moving authority out of expert judgment.

Most real systems contain both kinds of surfaces. The practical question is therefore not whether a project “uses MAGE,” but which properties deserve durable representation or authority, how strong the evidence should be, and which decisions should remain with experts.

The next chapter turns the theory, predictions, and boundaries into a research agenda.

6.4 Research Agenda

The theory proposes an explanation, the predictions say what should follow if it is right, and the scope conditions bound where those predictions should hold. This chapter turns the remaining uncertainty into a program of work: what to measure, which comparisons could distinguish the proposed mechanisms, which quantities remain unidentified, and what evidence would strengthen—or weaken—the account.

The equations in Parts V and VI are best treated as objects of empirical identification rather than fixed laws. The nine hypotheses collected in the previous chapter name the quantities a study would have to measure.

The hypotheses do not require nine unrelated studies. Three designs cover much of the program.

6.4.1 Parameterizing the Model

The simple models above deliberately leave their parameters abstract. A research program can ask what determines them.

Let T denote an engineering task or task family, M the reasoning model, R the representation available to it, and H the harness through which reasoning is organized. For a fixed engineering environment, write

$$p(T, M, R, H)$$

for the probability that one attempt produces an acceptable candidate, and

$$c(T, M, R, H)$$

for the expected cost of that attempt. Cost need not mean inference cost alone. Depending on the study, it can include tokens, compute, latency, tool use, realization, validation, and human intervention.

For sustained work, we reuse the dynamic formulation from Section 6.1. The environment itself becomes a state variable: writing E_i for the environment inherited at step i,

$$p_i = p(T_i, M, R_i, H, E_i), \qquad E_{i+1} = f(E_i, a_i)$$

Here the equations serve a different purpose. Section 6.1 used them to express path dependence; the research problem is to identify how much that dependence matters in practice and what features of the environment drive it.

Alignment introduces a related quantity. For an Alignment strategy A, let

$$q(T, R, A)$$

denote the probability that an unacceptable candidate escapes the mechanisms governing the relevant obligation. In the idealized case of a sound deterministic gate over a fully covered, decidable obligation, $q = 0$ for that covered failure class. Statistical validators, incomplete coverage, mechanism defects, and semantic gaps produce weaker guarantees.

What experiments would estimate

The empirical question is therefore not whether agents have one characteristic reliability or cost. It is how these quantities vary across engineering tasks, model capability, representation, harness, and Alignment strategy, and how those factors interact.

The distinction matters experimentally. A stronger model may increase p. A better representation may increase p, decrease c, or both. A longer reasoning loop may increase the probability of eventually finding an acceptable candidate while increasing total cost. An independent gate may leave the producer's p unchanged while decreasing q. Modeling and Alignment can therefore produce similar observed improvements in final system quality through different mechanisms.

The objective is therefore not one universal coefficient for agentic software engineering. It is a map of the conditions under which capability, representation, iteration, and authority substitute for or complement one another. Experiments can ask:

- Which factors systematically move p, c, and q?
- How do those effects vary by task family?
- When does representation substitute for model capability, and when does it complement it?
- When does dependence through the evolving engineering environment materially change retry cost, defect risk, or durable throughput?
- How much assurance can independent mechanisms provide without increasing producer capability?

The crossed experiments below provide one route to that map. Vary reasoning model and engineered environment over matched task families, then isolate representation and Alignment while holding the other fixed. Measurements of attempts, inference and tool cost, latency, intervention, admission, defect escape, and durable throughput can distinguish capability effects, representation leverage, retry economics, and independent assurance rather than collapsing them into one measure of agent performance.

These experiments measure the value of representation to realization, not its entire engineering value. As model capability improves, a representation may contribute less to the cost or probability of producing an acceptable change: the reasoner may reconstruct the relevant relationship cheaply and reliably without it. The same representation may still matter for engineering oversight by giving engineers and independent mechanisms an inspectable account of consequential system knowledge. Part VII takes up that second value directly.

6.4.2 Engineering Routing

This parameterization suggests a further implication for automated software engineering.

Model routing typically treats the task as given: estimate its difficulty, then select a model whose capability and cost are appropriate to it. Software engineering permits a broader intervention because the environment can change the problem presented to the model.

Suppose a migration appears to require a frontier model to inspect hundreds of service configurations and reconstruct their dependencies. A trustworthy dependency model may turn the same work into a much smaller graph problem that a cheaper reasoner can handle reliably. Changing an authentication boundary may warrant a different strategy: the representation can still simplify the reasoning, but the consequence may justify the stronger model, additional evidence, and an independent admission mechanism as well. Engineering changes both the difficulty of the task presented to the reasoner and the assurance required around its result.

A difficult code-level reasoning task may become a simpler graph, state-machine, provenance, or constraint problem when an appropriate representation exposes the consequential relation directly. Better context or tooling may change the cost of reaching the answer. Iteration may purchase additional chances. Alignment may allow a less reliable producer to operate safely because an independent mechanism prevents covered failures from acquiring consequence. The appropriate model is therefore only one component of the realization strategy.

For an engineering task T, an automated system could in principle select the model M, representation R, harness H, Alignment strategy A, and retry budget k jointly:

$$(M^*, R^*, H^*, A^*, k^*) = \arg\min_{M,R,H,A,k} E[C_{\text{total}} \mid T, M, R, H, A, k]$$

subject to an engineering assurance requirement

$$P(\text{unacceptable outcome escapes} \mid T, M, R, H, A, k) \leq \epsilon_T$$

The task-specific bound ϵ_T matters. Renaming a local variable, changing a user-visible layout, modifying a billing calculation, and altering an authentication boundary need not warrant the same assurance strategy. Routing should reflect the obligations and consequences of the work, not merely an estimate of how difficult its implementation appears to a language model.

Call this **engineering routing**. A model router asks which reasoner should receive the task. An engineering router asks which engineered realization strategy should receive it: what the reasoner should reason over, which tools and context it should receive, how much iteration is economical, what evidence should be produced, and which independent mechanisms should govern admission.

This reverses an important assumption in conventional routing. Task difficulty need not be treated as fixed. Engineering can change it. A task that requires an expensive frontier model over an opaque repository may require less capability when a suitable representation exposes the relevant structure. Conversely, a strong model may make additional Modeling uneconomical for a simple or infrequent task. The optimum depends on the task, available capability, environment, assurance requirement, and recurrence.

The research implication is concrete. Parameterizing p, c, and q across engineering task families and engineered environments could provide the empirical basis for choosing among these strategies. The question becomes not merely *how difficult is this task for a model?* but

how cheaply can we engineer this task to be safe to delegate? That is the stronger automated-SE consequence of the theory.

6.4.3 Representation Selection, Induction, and Innovation

Part II offered six useful classes of engineering model, not a proof that those classes are necessary, sufficient, or minimal. Because representation creates leverage and can expand the surface available to Alignment, representation selection is itself a research problem.

A representation is useful only relative to the questions and consumers it serves. A **sufficient** representation family preserves enough information to answer the declared engineering questions and govern the declared obligations to the required standard; a **minimal useful** family removes representations whose marginal reasoning, assurance, or independence benefit does not justify their carrying cost. Mathematical minimality is not the target, since a strictly minimal set may be cognitively awful. Redundancy is testable in principle: two representations that encode overlapping information can both be valuable when one serves a different question, reasoner, or independent assurance path, while a representation that changes none of the relevant answers, governable obligations, independent evidence, or reconstruction cost may be carrying cost without leverage.

A concrete example gives the idea engineering meaning. Suppose engineers repeatedly reconstruct permitted service dependencies from handlers, configuration, and code. Adding an architectural graph costs something to create, reconcile, and maintain—but if later engineers and agents can answer the dependency question directly, and a validator can use the same representation to reject forbidden edges, the reconstruction and residual-assurance cost falls. A second representation that answers no additional question and adds no independent evidence adds cost without leverage.

Part IV already treated representation induction as a practical engineering move. An existing implementation can contain stable structure that no explicit model names. Static analysis, clustering, repeated failures, architectural context, and GenAI-assisted interpretation can help recover that latent structure and turn it into an explicit representation. This is not representation innovation in the strongest sense: the induced model makes explicit a consequential organization that the existing system already embodies.

A stronger research question is whether commodity intelligence can innovate representations. Given an engineering system and a distribution of questions, failures, and decisions, can a reasoner propose a representation that makes important relationships tractable even when that representation was neither supplied in advance nor straightforwardly latent as a repeated implementation pattern? Such a system might propose a new decomposition, intermediate state, relation, abstraction boundary, or combination of existing model forms because that representation better supports the engineering questions being asked. Together, induction and innovation are the ordered claim of H4: recover what is latent, then propose representations the existing repertoire does not supply.

There is also a learning formulation. Given a distribution of engineering questions, failures, and decisions, one could ask which representation preserves the distinctions needed to answer them while suppressing irrelevant state—the question representation learning asks in machine learning. MAGE brings a large engineering prior: software and systems engineering have spent decades developing abstractions for architecture, state, ownership, interfaces, requirements, quantities, provenance, and assurance, and the experience in this book suggests many are also unusually usable by foundation models. A complementary direction is to train models specifically for engineering abstractions[88–90], which might let smaller models perform reasoning that currently requires expensive general-purpose intelligence. Where a representation is consumed only by machine reasoners and trustworthy mechanisms, it might trade human ergonomics for compactness or stronger mechanical analysis, provided evidence can be projected back to the human boundary where judgment resumes. The open question is not whether learned representations should replace engineering models, but which representations best allocate reasoning among humans, agents, and mechanisms for the question at hand.

An experiment can distinguish the two directions. Compare expert-authored, machine-induced, and machine-innovated representations against both the raw implementation and one another. For induction, measure whether the proposed model faithfully recovers consequential structure already present in the system. For innovation, use held-out engineering questions and failures to test whether the proposed representation exposes useful structure that the existing representational repertoire did not. In both cases, measure reasoning success, reconstruction cost, correspondence failures, carrying cost, human comprehensibility, and benefit to later humans and agents.

6.4.4 Experimental Designs

Separating Modeling from Alignment

One particularly clean factorial design can separate Modeling from Alignment. Hold the task, reasoning engine, and admission mechanism fixed while varying whether the producer receives the structured representation used by the environment. If Alignment supplies the claimed independence, covered defect escape should remain unchanged; if Modeling supplies the claimed leverage, attempts, inference cost, latency, or intervention required to reach admission should fall when the representation is available.

Multi-project longitudinal design

This design should vary or observe engineered-environment quality while tracking durable throughput, defect escape, and human-attention burden over weeks or months. The strongest version is a crossed experiment: several reasoning engines from weaker to frontier, several otherwise comparable environments from thinly to richly engineered, and the same task families across every cell. The interaction would distinguish three possibilities. If richer environments primarily substitute for raw model capability, weaker models should

close part of the gap in stronger environments. If the two are complementary, the strongest models should gain most from strong environments. If environment quality contributes little, the cells should move mainly with model capability. This design bears directly on H1, H8, and H9.

The crossed experiment should also measure the distribution of human work and the productive capacity stored outside the agent. Hold the task family and reasoning engine fixed, vary environment quality, and record how much human attention remains across coding, debugging, test construction, review, environment setup, integration, recovery, and other activity classes, alongside task scale, reconstruction cost, intervention burden, defect escape, and durable throughput. A richer environment that reduces only code-production time would support the narrower productivity account; one that shifts several classes of realization work out of repeated human execution while maintaining quality would support MAGE's broader account of how engineering work shifts. Kumar et al.'s sixteen activity classes provide an existing measurement vocabulary, avoiding the need for another MAGE-specific taxonomy[91].

Subsystem-level design

It should compare engineering surfaces that differ in representation and governance while holding as much surrounding repository context constant as possible. Track model correspondence, reconstruction effort, intervention, churn, and defect escape. Stepped introduction of a model or mechanism is particularly useful because the same subsystem supplies its own pre-intervention baseline. This design tests whether representation leverage, determinization, and mechanized assurance appear where the theory predicts rather than merely correlating with generally well-run projects.

Event-centered longitudinal design

It should follow one failure class through its whole life: first occurrence, repeat exposure, structural recognition, local repair, any durable conversion, propagation to later work, later recurrence, and eventual maintenance or retirement of the resulting asset. This is the strongest way to separate "the engineer got better" from "the environment learned." Applied across organizations, it also tests whether authority pathways determine whether a local diagnosis becomes shared engineering capital.

What the designs still need

All three designs require credible measures of durable throughput, human-attention burden, environment quality, and engineering-capital return. Until those are available, support ratio, mechanism count, and repository size should remain descriptive rather than proxies for maturity or value.[28]

[28] Candidate measures such as development completeness, redundant-encoding density, and agent-legibility remain exploratory and are not part of MAGE's canonical vocabulary.

Commodity intelligence may make some of these experiments unusually tractable: the reasoning engine, repository state, task family, and engineered environment can be held approximately fixed while one representation or mechanism varies.

The theory may also have a broader domain than software engineering. Its central mechanisms are not inherently code-specific: purposeful representations can reduce reconstruction, explicit obligations can support independent evidence and authority, and recurring judgment can become durable structure. Other engineering disciplines already rely heavily on models, analyses, tolerances, and assurance; some forms of professional knowledge work likewise operate through structured representations, rules, evidence, and approval. This book does not test that generalization. Part VII gives several concrete examples, while extending MAGE beyond software engineering remains part of the research agenda.

What the Theory Claims

MAGE proposes a directional account of agentic software engineering. Its central claim is that agentic capacity acts through the engineered environment surrounding the work. Increasing implementation capacity should therefore have different consequences depending on the representations, evidence, authority, and other engineering structure through which that capacity acts.

Three propositions carry most of the theory.

Environment fit moderates capacity. Additional agentic capacity should produce more durable throughput when the engineered environment fits the work, and more churn, escaped failure, repeated intervention, or control friction when it does not.

Representation creates leverage. A task-relevant and trustworthy representation can reduce the reconstruction required to answer an engineering question and can expose consequential distinctions at an abstraction more tractable than the underlying implementation. Its value should increase where reasoning would otherwise exceed the practical horizon of the reasoner.

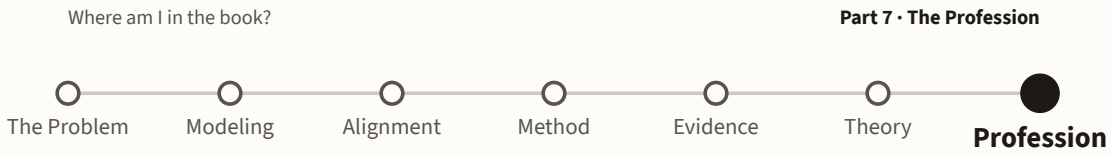
Engineering capital amortizes judgment. Models, mechanisms, procedures, evidence, and other durable structures can reduce the reconstruction, repeated judgment, rework, or risk later work must purchase again. They provide that return only while they remain fit; stale or incoherent structure can instead become a cost.

These claims are conditional. MAGE does not predict that more modeling is always better, that every obligation should become deterministic, or that accumulated engineering machinery is intrinsically valuable. Representation, authority, and engineering capital carry construction, maintenance, coordination, and friction costs. Some consequential judgments remain contextual or insufficiently observable and properly remain matters for expert judgment.

The research agenda in this Part identifies ways to test these claims and their proposed mechanisms. MAGE is not presented as a finished law of agentic software engineering. It is an explanation that can be tested, refined, bounded, or rejected.

Part VII asks what follows for software engineering—and for the engineer—if that explanation is substantially right.

Part 7: The Profession



QUESTION THIS PART ANSWERS

What follows for software engineering—and for the engineer?

THE SYNTHESIS

When implementation becomes abundant, software engineering does not disappear. Its scarce work moves. Engineers spend proportionally less effort producing routine realization and more deciding what must be true, choosing the representations through which consequential properties can be understood, establishing evidence, allocating authority, coordinating autonomous work, and accepting responsibility for the resulting system.

CARRYING FORWARD

Governed Engineering Environment · Engineering capital · Modeling Principle · Alignment Principle · Determinization frontier · Model

NEW HERE

Representation engineering

Part VI treated MAGE as a theory: an explanation of how agentic capacity, representation, authority, engineering capital, and the surrounding environment interact; what that account predicts; where it should apply; and how it might be tested.

This final Part asks what follows if that account is substantially right. It moves outward from the theory to the profession: first to the organization of software work under abundant implementation, then to MAGE's relationship with older engineering and computing traditions, and finally to the engineer's expertise, authority, education, and responsibility.

Delegation changes what engineers do. It does not remove their answerability for what they build.

Carrying forward: Theory of MAGE · Governed Engineering Environment · Engineering capital · Modeling · Alignment · Degrees of freedom

New here: Reorganization of software work · Representation engineering · Engineering tradition · Professional responsibility · Engineering education

7.1 The Reorganization of Software Work

Commodity intelligence does not invalidate software engineering. It changes its price structure. When implementation consumes less of the engineering budget, the relative value of specification, architecture, modeling, validation, measurement, and governance rises. The profession changes not because its foundations failed, but because the bottleneck moved.

MAGE is software engineering under changed economics. Many of its underlying moves are old: externalize intent, hide volatile decisions behind boundaries, test claims against evidence, automate repeatable operations, learn from failure. What changes is how far those moves can be carried when implementation and routine analysis become cheaper while autonomous work makes repeated human reconstruction increasingly expensive.

7.1.1 Was Implementation Ever the Bottleneck?

A reasonable objection begins with a fact: **software engineers do not spend most of their workweek typing code.** This result predates generative AI. Meyer et al., drawing on 5,971 responses from professional developers, found that software work spans a much broader mixture of development and non-development activities. More recently, Kumar et al. separated the developer workweek into sixteen activities; in their Microsoft sample, coding new features occupied roughly 11 percent of the actual week, alongside debugging, architecture and design, code review, testing, refactoring, environment work, security, communication, and other activities.[29]

Butler et al. turn that observation into a critique of contemporary GenAI.[30] If the technology merely makes the coding slice faster, its whole-job effect has a low ceiling. Even if coding occupies only roughly 10–15 percent of the workweek, doubling performance on it cannot double developer productivity; faster source production can simply move pressure downstream into review, testing, integration, and other work. Their criticism applies well to **GenAI used as individual code-generation assistance**. It also supports a broader point MAGE shares: lines of generated code are not an engineering outcome, and handing developers a tool does not by itself redesign the system of work around them.[31]

The disagreement begins when that fixed slice is treated as a bound on **agentic engineering**. Time categories are not independent production stages. Software engineers learn about a design by realizing it: implementation exposes missing abstractions; debugging changes what we believe about behavior; tests expose underspecified obligations; refactor-

[29] André N. Meyer et al.[92]; Sukrit Kumar et al.[91]. Kumar et al. distinguish sixteen activities and report approximately 12 percent of the actual week in communication and meetings, 11 percent in coding, 9 percent in debugging, 6 percent in architecture and design, and 5 percent in pull-request and code review.

[30] These developer-time studies describe how work was allocated before today's stronger autonomous coding agents; I use them as evidence about work allocation, not as an empirical ceiling on what later bounded agents can perform.

[31] Jenna Butler et al.[93]. The article's "Writing Code Is the Bottleneck" discussion explicitly frames the problem as GenAI used primarily to accelerate code creation while the surrounding development loop remains unchanged.

ing discovers structure; deployment reveals couplings that static design did not anticipate. This interleaving is not a new observation. Ralph's Sensemaking–Coevolution–Implementation theory models software development as repeated movement among understanding the context, revising the problem and design together, and constructing, debugging, and deploying the artifact; a later multi-method study using more than 1,300 developers and four longitudinal cases strongly supported that process account.[32]

MAGE therefore uses **implementation capacity** in an economic rather than time-sheet sense. Every design hypothesis that has to be realized before the team can learn from it imposes a cost, even if the eventual keystrokes occupy only a small fraction of the calendar. Reduce that cost sufficiently and more alternatives become affordable, feedback arrives sooner, and intelligence can participate in more of the surrounding loop. Modern coding agents already reach beyond source generation into repository inspection, debugging, test construction, refactoring, tool execution, and multi-step changes. Whether they can safely take on still more of that loop is exactly the engineering question MAGE addresses.

This makes the disagreement empirically testable. **If agents remain trapped in the IDE, the 15-percent argument largely wins. If governed agents can assume meaningful portions of debugging, testing, refactoring, integration, migration, and operation without proportionate growth in human oversight, a fixed coding-time slice is no longer the right model of attainable leverage.** MAGE is about crossing that boundary without losing engineering control.

DevOps offers a particularly close precedent. Infrastructure as Code and Policy as Code moved knowledge that once lived in operating practice into executable artifacts. MAGE generalizes the move without insisting on code as the form: an engineering decision might become a type, model, schema, validator, permission, workflow, or documented review obligation. **Useful engineering knowledge increasingly becomes structure later work can inherit; residual semantic judgment remains with people.**

MAGE inherits the older disciplines rather than replacing them: explicit intent, short feedback, automated delivery, and governed autonomy solve different parts of the engineering problem.

7.1.2 Agents Entered an Implementation-Centered Discipline

The first generation of agentic software-engineering research inherited the engineering object already in front of it: a repository and a software-development task. The central question was how to make an increasingly capable reasoner perform that task more effectively.

The progression is instructive. RepoCoder retrieves repository-level information that would otherwise lie outside the immediate completion context[9]. SWE-agent shows that the inter-

[32] Paul Ralph, the Sensemaking–Coevolution–Implementation theory[94], and its multi-method follow-through[95], which combined a survey of more than 1,300 developers with four longitudinal case studies and reported strong support for the SCI account. For an older and deliberately stronger version of the "implementation is design" argument, see Jack W. Reeves[96].

face between an agent and its computer materially affects its ability to navigate repositories, edit files, and execute tests[10]. AutoCodeRover goes further toward representation: rather than treating a project as merely a collection of files, it exploits program structure and an abstract syntax tree to localize and reason about changes[11]. Later work extends the trajectory toward inferred intent, extracting specifications from the same artifacts[97]. These systems differ substantially, but share an orientation: improving the machinery through which an agent acts on the software artifact.

MAGE accepts that machinery and changes the engineering object. The question is no longer only how an agent should navigate a repository, retrieve context, use tools, or construct a patch. It is what durable environment should be engineered around autonomous implementation: which system properties deserve explicit representation; which obligations deserve authority; what evidence work must produce; how representations remain aligned with the implementation; and how recurring judgment becomes machinery inherited by later work.

The difference is one of level, not opposition. Repository retrieval, structured code search, agent-computer interfaces, tests, and tool use all belong inside a governed engineering environment. MAGE's claim is that they should be understood as pieces of a larger engineered substrate rather than as accessories to a coding agent.

7.1.3 The Old Dream, with a New Substrate

The dream is older than the present substrate. In the 1980s, Charles Rich and Richard Waters's Programmer's Apprentice envisioned an AI system not merely as a programming tool but as an agent in the software process. The engineer would share programming knowledge with the Apprentice and delegate work across implementation, design, requirements, verification, explanation, and documentation[12]. Its internal representations made programming concepts explicit so that routine questions could be answered from represented knowledge rather than reconstructed from first principles.

The project realized important pieces of that vision, but under different constraints. Its intelligence depended heavily on programming knowledge deliberately encoded into specialized representations; Rich and Waters themselves observed that the economics failed when too much knowledge had to be supplied for each problem. Commodity intelligence changes that premise. A general-purpose reasoner can now arrive with substantial prior competence, enter an unfamiliar repository, use tools, interpret natural-language intent, and construct substantial implementations without the engineering organization first encoding everything it might need to know.

Old ideas therefore return throughout MAGE. The ambition of an intelligent programming partner is old. Explicit machine-usable engineering knowledge is old. Models, constraints, verification, feedback, and delegated realization are old. What changes is the economics. Earlier systems could demonstrate pieces of the vision; today's systems realize enough of it to expose the next-order engineering problems: keeping representations trustworthy,

deciding which obligations deserve authority, producing evidence at agentic velocity, and converting recurring judgment into machinery that later work inherits.

MAGE does not depend on a historically novel ambition. It is an engineering account of what happens when much of that ambition becomes operational at production scale.

This history places MAGE in a broader progression. AI has long engineered external machinery around finite reasoners through memory, structured representations, planners, tools, and world models. Agentic software engineering applies related machinery to work over repositories through retrieval, program-aware search, purpose-built interfaces, tests, and inferred specifications. MAGE moves the engineering boundary outward again: the repository and agent remain inside the picture, while purposeful system models, authoritative obligations, evidence, controls, and the machinery that maintains them become first-class objects of software engineering.

7.1.4 Representation Engineering

As implementation becomes cheaper, engineers can spend more of their effort on the representations through which humans and machines reason about the system. **Representation engineering** is the work of choosing, creating, maintaining, and connecting those representations. Requirements, architectural views, state machines, dependency graphs, policies, measurement models, and other representations answer different engineering questions. The skill is to choose a representation that makes the consequential property easier to reason about than reconstructing it from implementation, then maintain whatever correspondence that representation claims as the system changes.

This is different from context engineering. Retrieval can place more existing information before a reasoner; representation engineering can change the form in which the engineering question is expressed. An agent might retrieve enough source to reconstruct an architectural dependency, or work over a dependency model in which that relation is already explicit. The first improves access to the implementation; the second changes the representation over which reasoning proceeds.

None of this is a new representational or governance tradition. Requirements engineering, architecture, programming languages, formal methods, model-driven and model-based engineering, knowledge representation, safety and assurance engineering, and research on distributed cognition have long moved knowledge, reasoning, evidence, or control into explicit structures outside an individual engineer.[33] A type can exclude an invalid local state[47]; an intermediate representation can expose the structure a transformation needs[99]; a systems model can connect requirements, behavior, architecture, and evidence[100,101]. What changes with agentic software engineering is the economic importance of these representations: they can become active surfaces through which autonomous work is informed, scoped, checked, and reviewed.

[33] External-representation moves in recent AI: LLM+P externalizes planning into a classical planner[6]; MemGPT externalizes state into managed memory tiers[7]; Graph of Thoughts externalizes intermediate reasoning as a graph[8]; and WorldCoder externalizes a persistent world model as code[98].

Part II developed the modeling problem in detail: a representation is useful only for the questions its abstractions preserve, and it remains useful only while its claimed relationship to the system stays trustworthy. The professional consequence is simpler. As implementation becomes cheaper, choosing and maintaining the representations through which later work can reason becomes a larger part of software engineering itself.

7.1.5 Where Engineering Effort Moves

Implementation shifts; responsibility does not. The fleet can assume more realization work; engineers remain responsible for intent, abstraction, evidence, tradeoffs, and acceptance. The lifecycle keeps its phases—requirements, design, implementation, validation, operation—but the allocation of work within them changes. Figure 7.1-1 draws the shift.

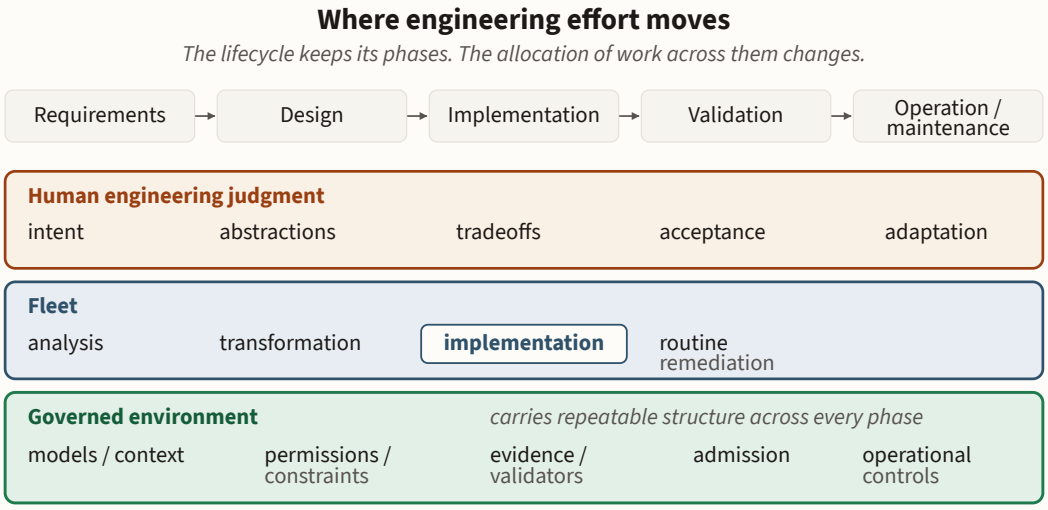


Figure 7.1-1: Where Engineering Effort Moves. *The lifecycle does not acquire a new set of phases. Implementation shifts disproportionately toward the fleet, while human effort concentrates in intent, abstraction, evidence, tradeoffs, and adaptation. The governed environment carries repeatable structure across all phases. Delegation changes the allocation of work across those phases, not the engineer's accountability for the resulting system or the judgment that acceptance still demands.*

Operational experience also changes value under this price structure. A conventional review can solve an instance. A durable representation or mechanism can change what every later instance has to rediscover or decide. That is governance conversion when the lesson comes from failure, and ordinary engineering capital formation when it does not. Either way, the engineering question is the same: ask whether the judgment belongs in the next reviewer's head or whether future work should inherit it from the environment.

The answer is not always "mechanize." Guidance remains useful where interpretation is genuinely semantic; human review remains appropriate where consequence or ambiguity

warrants it; and every new mechanism adds carrying cost. The goal is to **mechanize where doing so pays**, not to automate everything.

7.1.6 Review When Implementation Is Abundant

Code review does not disappear when agents produce the code. Its center of gravity moves.

At low change volume, an experienced engineer can treat the diff itself as the principal reasoning surface. Read the implementation, reconstruct the surrounding architecture, infer the affected behavior, inspect the tests, decide whether the change belongs. That strategy stops scaling as implementation becomes abundant. The scarce resource is no longer the ability to produce another diff. It is the engineer's ability to determine what the change means.

This pressure predates agents. Long before agentic programming, empirical work at Microsoft found that larger changes were harder to review well: as more files were touched, the proportion of useful reviewer feedback declined, and large changes imposed greater reviewer effort[102]. Agentic generation changes the magnitude of that problem, not its basic cognitive shape. The older result establishes that large changes tax human review and reduce useful feedback. It does not establish that explicit models would relieve the strain; models as the remedy are MAGE's proposal.

Greiler's more recent SCOPE proposal responds by changing the review operating model: detailed implementation review moves increasingly into the developer–agent loop, and team review becomes risk-proportional assurance over code together with higher-level artifacts and evidence[103]. That proposal already includes agent review, developer attestation, risk-based team assurance, and accountability. SCOPE changes the operating model for review; MAGE additionally changes the surfaces available to the reviewer. When the environment holds explicit models of structure, behavior, ownership, execution, measurement, and provenance, those models become review surfaces in their own right. The engineer need not reconstruct every relevant property from the diff.

Explicit models provide intermediate review surfaces. A change can be projected onto component ownership, permitted dependencies, service edges, lifecycle states, deployment placement, user journeys, measurement obligations, and provenance. Deterministic mechanisms discharge the obligations that are mechanically decidable before scarce human attention is spent. The reviewer then spends attention on the remainder:

- **Is the intended change desirable** — the question a diff could never answer on its own.
- **Do the models capture the relevant concern** — or has the change moved a property no current representation exposes.
- **Is the evidence sufficient** — do the checks that ran actually bound the risk this change carries.
- **Should the tradeoff be accepted** — the residual judgment that stays with a person.

None of this removes code from review. It changes when the reviewer must descend to it. Implementation becomes one evidentiary surface among several, rather than the origin

from which every system property must be reconstructed. The reviewer still reads the lines wherever implementation detail is where the risk lives. The models supply an index into the engineering significance of a change; they do not revoke the reviewer's authority to go look.

Review moves upstream, from inspecting realization toward evaluating intent, consequence, evidence, and the judgment that remains.

Appendix C collects representative forms of these models and shows how their authored and derived facts can be reconciled with the implementation; it develops the review surfaces as reusable model patterns rather than a prescribed catalog. Appendix I carries the within-case evidence on model coverage and correspondence. Those measurements speak to how much of the exercised system could be related to explicit models; they do not measure review time or review quality.

The same shift in review surfaces has a broader implication: effective engineering capability may increasingly be a property of the environment as well as the foundation model. A weaker reasoner operating over accurate representations, well-routed context, strong tools, and independent evidence may outperform a stronger reasoner over an opaque repository. MAGE does not establish that substitution quantitatively; Section 6.4 makes its parameterization and eventual use in engineering routing an empirical research question.

7.1.7 The Software Factory Risk

The alternative trajectory deserves attention. An organization can increase agentic capacity without substantially changing the representations through which it understands its software. Agents can receive more repository context, use more tools, execute broader changes, run tests, respond to failures, and operate increasingly large portions of the development lifecycle. Each improvement can make the software factory more productive.

But production capacity and engineering understanding are different quantities. An agent may successfully modify a system because it can reconstruct enough relevant context from source code, documentation, history, dependencies, tests, and telemetry. Another agent may review the result, automated checks may pass, and operational evidence may remain healthy. None of those mechanisms necessarily leaves behind an explicit account of the consequential properties of the resulting system. As implementation grows faster and less familiar, an organization can therefore become better at changing its software without becoming correspondingly better at explaining what that software is supposed to do and why it should be trusted.

The issue is engineering oversight. Engineers need not personally inspect every artifact or make every implementation decision. But they must retain an independent basis for understanding the consequential properties of the system, deciding which obligations govern it, evaluating the evidence for those claims, and changing the engineering environment when those claims no longer hold. Ultimately that authority belongs not to the software factory but to the people and institutions responsible for what it produces.

This becomes an assurance problem when the organization must answer questions that production alone cannot settle. What obligations apply? Where are they represented? Which parts of the system realize them? What evidence supports that correspondence? What assumptions remain? What changed since the previous assessment? If answering those questions requires another agent to reconstruct the engineering case from code, tickets, logs, policies, and history, then assurance itself depends on another inference over the artifacts produced by the factory. Better reasoners and better context can make that reconstruction more reliable. They do not make the underlying engineering knowledge explicit or give its obligations independent authority.

This distinction survives improvements in agent capability. Future agents may reconstruct the engineering case so cheaply and reliably that explicit representations are unnecessary for many acts of production. That would solve an important cost and reliability problem. It would not answer the oversight question if the organization's understanding of the system still depends on asking the producing machinery to reconstruct its own engineering case.

MAGE carries this engineering of the environment further. Agents can help maintain not only implementation and the context needed to change it, but the representations through which the implementation remains understandable: architectural relationships, behavioral models, invariants, decision structures, provenance, measurements, and their claimed correspondences to the realized system. Analyses can then operate directly over those representations, and selected obligations can be checked independently of the agent that produced the change. This does not eliminate uncertainty or require complete formalization. It changes which questions must be reconstructed from implementation each time.

The distinction matters because machine-scale implementation creates an opportunity for machine-scale assurance. The engineering environment need not remain limited to representations and checks that humans could feasibly maintain and apply by hand. A machine can maintain thousands of explicit relationships, continuously reconcile derived models against implementation, run analyses after every change, and surface the residual judgments that still require human attention. The objective is not a factory that humans understand by inspecting everything it produces. That goal becomes less plausible as production scales. The objective is a factory that preserves the engineering structures through which consequential properties remain inspectable, challengeable, and governable.

A software factory should therefore be evaluated not only by how much software it can produce, but by whether the organization retains engineering oversight: whether it can continue to understand, assure, govern, and take responsibility for the systems it produces.

7.2 MAGE in the Engineering Tradition

The reorganization just described resembles older engineering practice. Civil, mechanical, electrical, aerospace, and systems engineers routinely answer consequential questions through reduced representations of what they build: load models, circuit schematics, state-space models, simulations, drawings, requirements, and analyses. A useful representation makes an important property cheaper or safer to reason about than the realized artifact itself.

Software has always modeled too, but source code occupied an unusually powerful position. It was symbolic, executable, cheap to copy, and precise enough to become both the artifact and the representation engineers most often worked through. Secondary models therefore had to repay a standing synchronization cost. That economic difference helps explain why model-heavy software methods remained concentrated in domains where the assurance payoff justified the burden.

7.2.1 The Disciplines That Got There First

Older engineering disciplines had stronger reasons to separate representation from realization. Fabrication could be expensive or irreversible; full-scale trials could be destructive; physical iteration could take weeks or months. A structural model could answer a loading question before concrete was poured. Circuit simulation could expose a design error before a mask set was fabricated. Finite-element analysis could explore stresses before a sequence of physical prototypes was machined and broken.

The common engineering move matters: **ask the question over the cheapest representation that preserves the property you care about, then use evidence to justify realization.** Mature engineering still uses prototypes, experiments, and direct inspection; models do not replace the artifact. They organize the reasoning around it.

7.2.2 Model-First Engineering

The Siemens reconstruction illustrates what this looks like in a model-first engineering culture. The public account describes persistent models carrying geometry, requirements, behavior, physical relationships, allowable change, and traceability before an agent enters the picture. Software can appear downstream as one realization among others. The agentic question is therefore not how to recover engineering structure from source code, but how to let autonomous tools operate safely through structure the organization already made explicit.

That inversion clarifies MAGE's position. Software-first agent systems largely ask, *How do we make a codebase legible to an agent?* A model-first discipline can ask, *How do we let the agent work through the same engineering representations we already trust?*

Degrees of freedom clarify MAGE's relationship to conventional model-based engineering. A sufficiently complete model can determine realization closely enough that transformation,

generation, or synthesis is the natural next step. MAGE includes that case, but does not require it. Its models can instead specify the obligations that matter and the tolerances those obligations permit while deliberately leaving other realization choices open. Those degrees of freedom are not defects in the model; they are choices engineering has no reason to settle in advance. Where tolerances become narrow and little consequential freedom remains, deterministic generation may be preferable. Where substantial freedom remains, autonomous realization can choose among acceptable implementations. Commodity intelligence may make both ends of this spectrum cheaper: strong formalization where assurance warrants it, and selective Modeling and Alignment where complete specification does not. Section 6.3 develops this economic consequence.

7.2.3 Why Software Came Late to Model-First Engineering

Software's economics differed from those of older engineering disciplines. **Software realization was extraordinarily cheap; detailed design and implementation labor were not.** Once source code existed, another compilation, execution, or copy of the program could usually be produced at negligible marginal cost. But producing, understanding, debugging, and changing that source consumed skilled engineering labor—and the source itself was already a precise, executable representation through which engineers reasoned about the system.

That blurred a distinction older engineering disciplines could make more cleanly. A structural engineer reasons through a design before concrete is poured; a semiconductor designer simulates a circuit before fabrication. In software, the artifact presented to the cheap realizing mechanism—the compiler—is itself the product of an iterative design process. Reeves made the strongest version of this argument by calling source code the final software design. MAGE does not require that full claim. It requires only the narrower one: **software design and implementation have historically been tightly coupled, and code has served simultaneously as realization, executable specification, and a primary surface for further reasoning.**[34]

Secondary software models therefore faced an unusual economic test. The implementation was already precise, executable, and immediately available, while an architectural, behavioral, or process model imposed an additional correspondence burden. A model could be useful when authored and misleading after the implementation moved beyond the relation it claimed. Many projects therefore kept such representations thin unless their communication, reasoning, or assurance value repaid that standing synchronization cost.

Commodity intelligence changes both sides of that ledger. It lowers the skilled labor required to realize and revise software, and it can lower parts of the labor required to derive, update, reconcile, and check additional representations. Neither cost becomes zero. The important change is that **implementation and live modeling can become cheaper**

[34] Jack W. Reeves's "code is design" argument[96] is a useful historical extreme; MAGE relies only on the narrower claim that software design and implementation have traditionally been tightly coupled.

together, making model-mediated software engineering economical over surfaces where the synchronization tax once exceeded its return.

Figure 7.2-1 sets those three economic variables — and the change that moves them — in a single view.

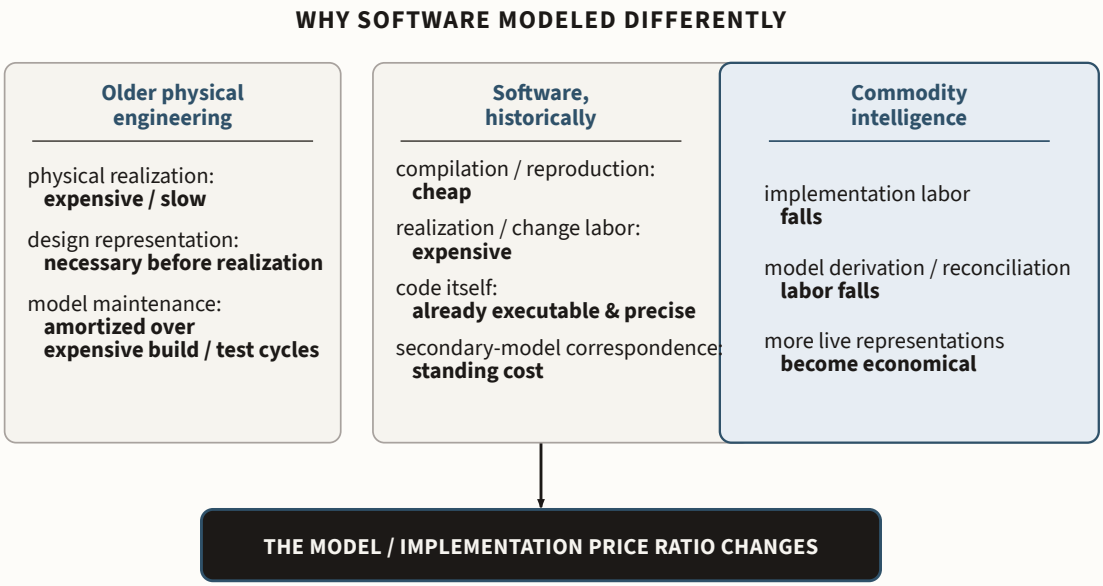


Figure 7.2-1: Why software modeled differently. *Software was never model-free. Compilation and reproduction were cheap, but producing and revising the detailed implementation required skilled labor, and code itself served as an executable engineering representation. Additional models therefore carried a standing correspondence cost. Commodity intelligence lowers both realization labor and parts of the cost of maintaining secondary representations, expanding the surfaces on which live models can repay their upkeep.*

7.2.4 The Language of Engineering

Models are the language of engineering. A circuit schematic is not a circuit; a finite-element model is not a bridge; a process-flow diagram is not a refinery. Each is a purposeful reduction that preserves selected relationships so engineers can reason about them without reasoning directly over the realized artifact. The representation is useful because it leaves most of reality out.

Software has had a more complicated relationship with this separation. Code is unusually expressive, precise, executable, and cheap to change. As the preceding section argued, those properties made it possible for implementation itself to carry much of the system's design. But the same malleability creates a problem as software evolves: architecture, intent, constraints, and operational knowledge can remain implicit in an artifact whose structure is continually changing. The system remains executable even when the engineering knowledge needed to understand and govern it has become difficult to recover.

This is one reason software has repeatedly developed disciplines that recover selected structure from the implementation or maintain it alongside the implementation. Requirements engineering externalizes obligations. Architecture exposes consequential structural relationships. Type systems and intermediate representations make selected program properties explicit. Formal methods construct representations over which stronger claims can be established. Model-driven and model-based approaches go further by making selected models primary engineering artifacts. These traditions differ substantially, but they share an engineering move: represent the property at a level where it can be reasoned about.

MAGE makes greater use of that move because commodity intelligence changes its economics. When implementation is expensive, maintaining another representation of the system must justify the additional synchronization cost. When implementation becomes cheap and agents can help construct, reconcile, query, and maintain representations, it becomes economical to move more engineering knowledge out of the implementation and into forms designed for the questions engineers need to answer.

The separation is not absolute. Implementation remains an important source of truth, and some properties are most naturally reasoned about there. Nor does every useful representation need to become authoritative. The point is that code need no longer bear so much of the burden of simultaneously being the realization, the repository of system knowledge, and the surface over which engineering reasoning occurs. Figure 7.2-2 sets the two inheritances and MAGE's composition in a single view.

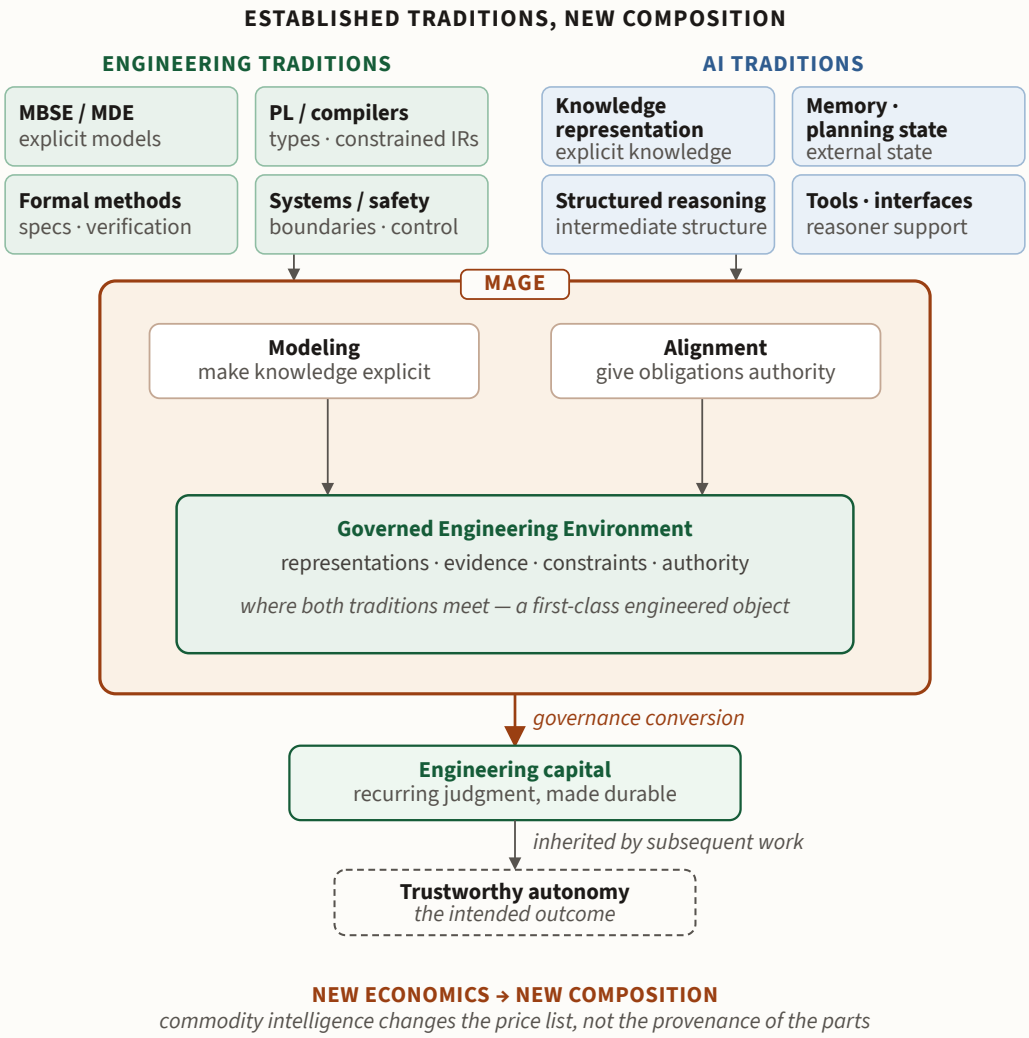


Figure 7.2-2: Established traditions, new composition. *MAGE draws on two long-running traditions. Engineering disciplines developed explicit models, constrained representations, verification, and external control; AI repeatedly extended reasoners through knowledge representation, planning state, memory, structured reasoning, tools, and surrounding machinery. MAGE brings these inheritances together around autonomous software work: Modeling makes engineering knowledge explicit, Alignment gives selected obligations authority, and both meet in the Governed Engineering Environment. Governance conversion carries recurring judgment into durable engineering capital. Commodity intelligence changes the economics of this composition, not the provenance of its parts.*

7.2.5 What Happened to the Cost?

Software models have always faced an economic test: the reasoning or assurance they provide must repay the cost of constructing and maintaining them. Section 6.3 treated that boundary directly. Commodity intelligence may move it from both directions, making stronger formalization cheaper in some settings while also making selective representation, reconciliation, and evaluation worthwhile in systems that could never justify complete formalization.

The professional consequence is simpler. Cheaper implementation does not make engineering effort disappear; it changes where scarce effort pays. More of that effort can move toward specifying intent, choosing abstractions, maintaining useful representations, constructing evidence, governing autonomous work, and deciding what should acquire authority. The governed engineering environment itself still carries construction, maintenance, and friction costs, so the shift pays only where the structure future work inherits reduces enough reconstruction, judgment, rework, or risk to justify what it costs to carry.

7.2.6 What MAGE Claims Is New

MAGE does not claim to have discovered models, constraints, verification, external memory, structured reasoning, or autonomous software agents. Its claim is architectural and disciplinary. Much recent agentic software engineering has asked how increasingly capable agents can perform work within the software-engineering environment they inherit. MAGE asks how that environment itself should be redesigned when autonomous implementation becomes a primary productive substrate. Its answer is to make purposeful models, Alignment machinery, and the governed engineering environment first-class objects of software engineering; to treat recurring judgment converted into durable structure as engineering capital; and to leave consequential semantic judgment with engineers where representation and mechanism cannot responsibly carry it. The tools are new; the underlying engineering arrangement is not.

7.2.7 Beyond Software Engineering

MAGE was developed and studied in software engineering, so that is where this book's evidence lies. But the pattern it describes may be broader. Its central questions concern representation, evidence, authority, and durable judgment—the problems that can arise when commodity intelligence performs consequential work.

The strongest neighboring case is other engineering. Beyond that, some forms of professional knowledge work have similarly explicit representations, rules, evidence, and approval structures.

- **Other engineering disciplines.** Mechanical, civil, electrical, aerospace, and systems engineering already reason through purposeful models: geometry, schematics, state-space models, load cases, requirements, tolerances, simulations, and assurance evidence. MAGE would not ask these disciplines to discover modeling. The question is whether

autonomous systems can work through the representations engineers already trust, use the analyses native to them, respect declared tolerances, and remain bounded by independent evidence and authority.

- **Accounting.** Accounting transforms economic activity into structured representations: ledgers, charts of accounts, classifications, reconciliations, controls, and audit trails. An autonomous system need not reconstruct those rules from prose for every transaction. It can reason over the accounting representations, apply analyses such as reconciliation or variance checking, and operate under approval limits, segregation-of-duties rules, and closing controls. Ambiguous classification, materiality, and other consequential judgments can remain with an accountant.
- **Legal case analysis.** Legal work also relies on reduced representations: parties, claims, elements, facts, authorities, procedural posture, deadlines, and relationships among precedents. An autonomous system might organize those structures, trace propositions to authority, test whether a claim's elements have supporting evidence, or identify conflicts among authorities. Interpretation, strategy, credibility, and the weight of competing authority remain more judgment-laden.
- **Technical writing and synthesis.** A long technical work has structure that no single drafting episode should have to reconstruct: claims, terminology, dependencies among arguments, citations, examples, cross-references, and stylistic obligations. Explicit representations can make those relationships inspectable and support analyses for consistency, missing support, broken dependencies, or stale examples. Some obligations —citation presence, terminology consistency, numbering, cross-references—can be given mechanical authority, while substantive argument remains with the author.

These examples differ intentionally. The unit of application is not the profession as a whole but the particular surface. A filing deadline may be highly governable while a novel legal interpretation is not; an accounting reconciliation may be mechanized while a materiality judgment remains human; an engineering tolerance may be checked automatically while a design tradeoff stays open.

The questions travel even when the mechanisms do not: what must the reasoner understand? What representation makes that question tractable? Which properties can be analyzed? Which obligations can be given independent authority? Which judgments should remain with a person? Which repeated decisions are worth making durable?

That broader generalization remains a hypothesis. The evidence in this book is software-engineering evidence.

7.3 The Engineer

The preceding chapters moved outward from MAGE to software engineering and then to engineering more broadly. The final question returns to the person responsible for the system. If agents can increasingly implement changes, maintain models, run analyses, and propose engineering moves, what remains the engineer's work?

The answer is not whatever tasks machines happen to perform poorly today. It is responsibility for what the system is supposed to mean: which distinctions matter, which properties should hold, which representations deserve trust, what evidence is sufficient, which obligations should be made authoritative, and which tradeoffs are acceptable.

7.3.1 Authority Follows Competence

MAGE increases both the leverage of engineering knowledge and the consequences of getting it wrong. An agent can increasingly carry much of the method itself: recognize a modeling opportunity, choose among known model families, construct a state machine, query a dependency graph, propose an invariant, or suggest that recurring inspection become machinery. Appendix E packages that repertoire in the Self-Governance mastery skill. As the repertoire becomes machine-readable, the engineer is distinguished less by recalling the available moves than by judging where they apply. An agent's command of the repertoire does not establish that a chosen abstraction captures what matters, that a proposed invariant expresses the intended property, or that the resulting tradeoff is acceptable. Those judgments remain engineering work.

Professional authority

Engineering authority is not merely technical. Societies entrust engineers with consequential decisions because engineers are expected to understand the systems they certify and to stand behind them. The institutional details differ across domains, but the principle holds: authority follows competence and responsibility. Engineers can carry legal, professional, and sometimes personal liability for the decisions they sign. An AI system can generate an artifact or assemble evidence. It cannot assume the professional responsibility attached to an engineer's judgment.

Other engineering professions make answerability institutionally explicit through professional licensure. In U.S. professional-engineering practice, the concept of responsible charge attaches authority to the engineer exercising direct control and supervision over consequential engineering work; licensed engineers are also subject to enforceable ethical and legal duties directed toward public health, safety, and welfare.[35]

[35] On responsible charge — direct control and supervision by the engineer exercising professional judgment — see NSPE Position Statement No. 10-1778 (rev. May 2024)[104]; on the statutory duties and practice exemptions that bound where licensed authority is required, NSPE Position Statement No. 09-0173 (rev. November 2023)[105].

Demanding that engineers be able to oversee agentic decisions is therefore not merely an organizational preference. It is part of the basis on which engineering authority can legitimately be exercised. The responsible engineer must have a means to understand and challenge the system rather than merely defer to the machinery that produced it.

The converse matters too. If meaningful engineering oversight cannot be established for a system making consequential decisions, withholding that authority from the system is itself a legitimate—and sometimes necessary—engineering outcome.

The judgment frontier

One social purpose of the engineer is to be entrusted with, and answerable for, technical work too complex or consequential for ordinary individuals to undertake for themselves.[106] A small bridge across the ditch in my parents' yard allowed the neighborhood kids to visit more easily, but a few boards crossing a span of three feet did not call for a structural engineer. The Golden Gate Bridge did. Large engineering organizations extend the same principle internally: a chief engineer may be answerable for a system assembled through layers of engineers and specialists, below which lie tasks simple enough that engineering expertise is no longer needed either to perform the work or to establish that it was done correctly. GenAI can push that boundary downward. If machine intelligence can reliably perform more of the work, fewer human engineers may be needed to build the same system. Nothing about engineering requires preserving jobs that no longer require engineering judgment.

But there is another boundary: the judgment frontier. Part VI showed that some obligations cannot yet be reduced to mechanical checks. A capable reasoner must still decide whether the evidence is enough, whether a tradeoff is acceptable, or whether something is simply right for its purpose. Better GenAI will move this frontier too. We may entrust machine intelligence with judgments that in 2026 we reserve for experienced engineers, including increasingly contextual and qualitative ones. Yet technical capability and social authority are different things. Society may accept an engineering organization with far fewer human engineers before it accepts one with none: no human professional entrusted with the work, no human being who can be called to account for its judgments, and only machine intelligence standing behind the result.[36] The frontier is therefore not only what machines can judge. It is what society is willing to let them judge on our behalf.

There is a second reason not to erase the human side of that frontier: resilience. A society whose engineering capacity exists only through machine intelligence has made that intelligence critical infrastructure. If the models become unavailable, untrustworthy, compro-

[36]Contemporary calls for greater social control over advanced AI have argued that decisions about powerful systems should not be left solely to the organizations developing them, but should reflect broader social judgment.[107] Frank Herbert imagined an extreme version of such a choice in Dune: after the Butlerian Jihad, its fictional society prohibits "thinking machines" and deliberately preserves human capacities to perform work that machines might otherwise perform.[108] Herbert's answer is extreme, but the underlying question is real: societies must decide not only what machine intelligence can do, but what roles they are willing to surrender to it and what human capacity they wish to preserve.

mised, or controlled by someone else, the society has no independent engineering capacity to fall back on. Some human engineering competence may therefore remain valuable even when it is no longer the cheapest way to produce engineering work. Other industries have learned the same lesson about efficiency and reserve capacity: systems optimized around lean inventories and just-in-time supply can be highly efficient in ordinary conditions while leaving too little cushion when a common disruption arrives. The COVID-19 pandemic made that tradeoff unusually visible.[109] The judgment frontier is therefore partly about trust and accountability, but also about preserving the ability to act without the machine.

Resilience need not mean preserving only human reserve capacity. It may also favor diversity in where machine intelligence resides. If consequential engineering increasingly depends on models available only through a small number of remote providers, access to those providers becomes part of the engineering infrastructure on which individuals, firms, and societies depend. Smaller local or on-device models could provide another layer of reserve capacity: perhaps less capable than frontier systems, but inexpensive, independently operable, and sufficient when paired with strong representations, tools, and evidence. Whether that capability becomes technically or economically competitive is uncertain. The engineering principle is not. A resilient system should be wary of concentrating an essential productive capability behind a dependency it cannot itself control.

MAGE predicts that implementation will become increasingly fungible and that fewer people will be needed to carry it. It does not follow that a resilient society should allow independent human engineering capacity to disappear with it.

Inset — Engineering at the frontier of authority

ENGINEERING PRACTICE

Engineering does not always decide whether to proceed. Its responsibility is to make the engineering case legible enough that those who do decide know what is understood, what can be assured, what remains uncertain, and what risk they are accepting.

Engineering history contains painful demonstrations of this distinction. Before Challenger, Morton Thiokol engineers recommended against launching at the temperatures expected on January 28, 1986. Management reversed that recommendation, but the engineering objection survived in the record, allowing the Rogers Commission to reconstruct both the technical disagreement and the decision process that followed it.[110] Columbia illustrates a different failure: engineers raised concerns about the foam strike and sought additional imagery, but important assumptions and uncertainties did not reach the Mission Management Team with sufficient force or clarity. The Columbia Accident Investigation Board subsequently called for an independent technical authority and safety-assurance organization with authority over safety decisions.[111]

Volkswagen's diesel-emissions scandal brings the problem into software. Engineers implemented software that recognized emissions tests and changed vehicle behavior; the subsequent legal record made it possible to reconstruct what engineers and managers knew, what they concealed, and who participated in the decisions. Volkswagen engineer James Liang ultimately pleaded guilty to conspiracy and was sentenced to 40 months in federal prison.[112] The disclosures of Frances Haugen and, later, former Facebook engineering director Arturo Béjar bring the pattern entirely into software systems. Facebook's own researchers studied harms experienced by young users, while Béjar later testified about internal measurements of such harms and his efforts to communicate them to senior executives.[113,114] Whatever judgment one makes about the appropriate product or policy response, these records made a more basic question answerable: what did the organization know about the behavior of its system, who knew it, and what did those with authority do with that knowledge?

These cases suggest a simple professional test. When consequential harm occurs, the engineering record should show what engineers knew, what they did not know, what they warned about, and who chose to proceed. This is not merely defensive documentation. Making knowledge, uncertainty, disagreement, and evidence explicit is part of how engineering informs consequential decisions and makes those decisions answerable.

The distinction matters increasingly as software agents become capable of consequential action. Engineering oversight does not require that engineers personally make every decision an agent makes. It requires that engineers be able to establish the relevant bounds: what authority has been delegated, what properties are assured within it, what evidence supports those claims, what departures can be detected, and where the limits of that assurance lie. The engineering challenge is to extend the frontier over which meaningful authority can be exercised as the capabilities and autonomy of the system expand.

Sometimes the available evidence will not establish the desired bound. Engineers are not always the final decision-makers. Executives, military commanders, regulators, political leaders, and societies may have legitimate reasons to proceed despite engineering uncertainty: the consequences of inaction may themselves be severe, and competitive or geopolitical pressures may make accepting substantial risk rational. Engineering's responsibility is to preserve the distinction. A decision to accept an inadequately bounded risk does not make the risk bounded. The record should make clear what engineering could establish, what it could not, and who possessed the authority to proceed anyway.

That answerability matters after the decision as well. Society needs identifiable people and institutions from whom an account can be demanded and to whom consequences can attach. Professional licenses can be revoked, certifications withdrawn, organizations sanctioned, and individuals held legally responsible. A machine cannot become the terminus of that chain of responsibility. Delegating a decision to a machine does not delegate the responsibility that justified the human or institutional authority to make it.

This is the MAGE position on increasingly capable autonomous systems. The objective is not to hold autonomy behind an arbitrary boundary because autonomy is dangerous. It is to push outward the boundary within which autonomy can be responsibly granted. Better models can expand what machines can do. Better engineering environments must expand what humans and institutions can understand, constrain, assure, and govern. Modeling makes consequential knowledge explicit; Alignment gives selected obligations authority; evidence makes the resulting claims challengeable. Together, these mechanisms enlarge the domain over which engineers can responsibly delegate action while retaining the ability to understand what has been delegated and to answer for the consequences.

Engineering has always required moving between levels of abstraction: choosing the level that exposes the property at issue and descending when the current abstraction no longer answers the question. Working with agents extends that practice. The shallow reading is natural-language programming: say what you want and receive code. The engineering task is broader. As agents acquire more of the technical repertoire, the engineer's leverage moves

upstream—from supervising individual actions toward authoring and judging the environment in which those actions occur.

That upstream work can persist after the individual act of judgment. A well-chosen abstraction, authoritative constraint, or reliable validator can continue carrying part of an engineer's reasoning into work performed by other people and agents. This is one reason the environment can accumulate engineering capital. It is also why authoring the environment carries unusual leverage: a consequential judgment made well once may shape thousands of later changes.

But inheritance is not abdication. Someone must still recognize when the environment no longer fits the system, when an old distinction has stopped mattering, or when a new obligation lies outside what the existing machinery can see. The expert contribution moves toward creating, evaluating, and revising durable structure; it does not disappear once that structure exists.

The division is useful enough to name; Table 7.3-1 sets the two columns side by side.

Table 7.3-1: The authorship division. *Self-Governance can recognize and propose engineering moves; the engineer remains answerable for the significance, intended meaning, authority, and tradeoffs those moves serve.*

Self-Governance carries	The engineer remains answerable for
recognizing a modeling opportunity	determining its significance
proposing a representation	establishing what it means
extracting candidate concepts	choosing the consequential abstraction
finding an inconsistency	adjudicating the intent behind it
formalizing a known property	deciding what property should hold
proposing a governance mechanism	deciding what it should enforce
identifying a recurring failure	determining acceptable risk and tradeoff

Organizations may distribute these responsibilities across engineers, platform teams, review boards, or other accountable roles. The allocation should be explicit and governed.

Natural language may be one interface to the work. It is not the engineering method.

7.3.2 Sign-Off Moves Toward Models and Evidence

Imagine an agentic system in which the fleet constructs most of the implementation, maintains selected engineering representations, runs validators, and assembles evidence for the claims the design is supposed to satisfy. Human review then looks less like authorship of the implementation and more like technical acceptance: inspect the representations, challenge their assumptions, examine the evidence attached to important claims, request deeper analysis where the case is weak, and decide whether the result is fit to accept.

This is a professional prediction rather than a finding of this book. At scale, implementation-level inspection cannot remain the primary human assurance mechanism. That was true before coding agents; no engineer establishes confidence in Linux by holding its source code in working memory. Agentic implementation makes the mismatch more immediate by producing large, unfamiliar changes faster than a person can conventionally inspect them.

Models provide intermediate review surfaces. A structural representation can expose decomposition and permitted dependencies; a behavioral representation can expose legal states and transitions; a decision model can expose authority; a measurement model can connect an engineering claim to the observations meant to support it. Tests, static analyses, benchmarks, simulations, and operational measurements then provide evidence about those claims. Neither representation nor evidence is sufficient alone.

The relationship can be stated in four lines.

> **Models** state selected claims about the system and the properties that matter. **Evidence** says why we should believe those claims. **Governance** says what evidence is required and what decisions that evidence can authorize. **The engineer** remains answerable for whether those are the right claims, representations, tradeoffs, and standards of evidence.

The object of review moves; accountability does not. An engineer has never needed to hand-implement every consequential choice to remain answerable for the resulting system. Agentic systems extend this arrangement across more of the system. The engineer delegates the keystrokes and stays answerable down to the code and past it — to the properties a model names and the evidence a validator produces.

Early evidence already describes part of that shift. A six-month longitudinal study of engineers adopting AI assistants found most of them writing less code and spending more time directing, checking, and correcting what the model produced — a mode its authors name *supervisory engineering work*. It did not come free: flow and cognitive load both eroded even as output held[115]. Their observations are consistent with this shift; MAGE's proposal is to give that supervision better reasoning surfaces — explicit models and evidence rather than an ever-growing stream of unfamiliar output.

Real answerability requires competence and authority. The reviewer has to distinguish local defects from structural problems, and a structural diagnosis needs a path to the architecture, models, mechanisms, or infrastructure that can address it. Otherwise responsibility becomes nominal: the engineer can see the problem but cannot change the environment that produces it.

The models themselves remain objects of engineering. Their fidelity is not assumed; the environment must maintain whatever correspondence each model claims to the realized system. Traceability, re-derivation, comparison, and drift detection are ways to produce that evidence. MAGE does not replace testing and inspection with models; it relates that evidence to explicit engineering claims.

7.3.3 From Full-Stack to Full-System Engineering

Software has used full-stack engineer to describe someone able to work across multiple implementation layers: interface, application logic, data, infrastructure, and deployment. Commodity intelligence may make that breadth easier to obtain. An engineer assisted by capable agents can increasingly construct and modify artifacts across layers without personally possessing the implementation fluency that each layer once demanded.

But MAGE suggests a broader requirement. The engineer responsible for the resulting system must reason not merely across its implementation stack, but across the full system and its lifecycle. A representation chosen during design determines what later implementation can express; an architectural boundary determines what can be validated independently; operational evidence can reveal that an earlier abstraction was wrong; a recurring failure may justify changing the model, mechanism, or environment inherited by every subsequent change. These are not separate concerns merely because organizations have historically assigned them to separate roles.

Building DocAble made this breadth concrete for me. I needed to understand regulatory obligations such as the ADA and WCAG; product questions such as usability, user expectations, and the capabilities and limitations of existing accessibility tools; document formats and their underlying representations, including PDF, Office, LibreOffice, LaTeX, and Typst; accessibility semantics such as reading order, alternative text, tables, contrast, and document structure; software architecture and the models through which those formats could be transformed safely; agent behavior and the boundaries within which autonomous remediation could operate; validation and assurance sufficient to distinguish plausible output from conforming output; cloud-computing and worker models; memory, streaming, and performance behavior; deployment and operations; and the recurring failures that revealed when some part of the architecture needed to change. No single concern defined the system. Decisions among them interacted.

I have found that this process develops the engineer as well as the system. One fear about increasingly capable machines is that engineers will lose expertise as machines perform more of the work. That can happen if delegation removes the engineer from the feedback through which expertise develops. My experience with MAGE has been nearly the opposite. Cheap implementation enables a kind of hyper-exploration: I can form a hypothesis, model it, realize it, measure what happens, discover where my understanding was wrong, and repeat the cycle many times in the time a conventional implementation might once have taken. MAGE gives that exploration structure. The resulting models, evidence, constraints, and mechanisms become durable engineering capital in the Governed Engineering Environment, but the learning does not accrue only to the environment. The engineer goes through the loop too. Repeatedly choosing representations, confronting predictions with evidence, finding the proper level of a failure, and deciding what knowledge should become durable machinery forces the engineer to learn. Abundant implementation can deskill an engineer who merely consumes its output; used this way, it can instead become an unusually powerful medium for developing engineering judgment.

This is systems thinking applied to software engineering under agentic leverage. Local implementation can increasingly be delegated, but consequential judgment often requires following a property across requirements, models, architecture, implementation, validation, deployment, operation, and maintenance. The engineer need not personally perform every activity or master every technology. The requirement is stronger in a different direction: the engineer must understand enough of each relevant domain to recognize how decisions made in one part of the system constrain, enable, or invalidate reasoning elsewhere.

The professional trajectory may therefore be from full-stack toward full-system engineering. Full-stack breadth crosses implementation layers. Full-system breadth crosses domains, abstractions, and lifecycle boundaries. As agents make implementation expertise cheaper to apply, the latter may become the scarcer and more consequential form of engineering competence.

7.3.4 What Should We Teach?

If engineers increasingly review systems through representations and evidence rather than reconstructing every property from implementation, education should shift accordingly. This book is not an education study, so the argument is a professional hypothesis rather than a curricular finding.

The curriculum remains recognizable

Most of the foundations remain. Algorithms still matter. Operating systems still matter. Security arguably matters more. Programming does not disappear, nor does software engineering. The educational consequence of commodity intelligence is not a wholesale replacement of the curriculum. Engineers still need enough implementation fluency to understand the systems they build, diagnose failures, evaluate generated work, and descend through abstractions when necessary. They still need the disciplinary foundations that make those judgments possible.

What changes is the allocation of attention. If routine implementation becomes cheaper while specification, validation, coordination, and judgment remain scarce, education should reflect that shift. The question is therefore less what subjects disappear than which competencies deserve more or less practice.

Shift the emphasis

Less educational effort may need to go toward producing routine implementation by hand, and more toward selecting abstractions, specifying properties, measuring systems, validating claims, reasoning about quality attributes, and governing autonomous work. Modeling, specification, architecture, measurement, programming languages, and formal methods deserve more room—not because every engineer should become a language theorist, systems modeler, or verification specialist, but because these fields teach durable ways to choose abstractions, state properties precisely, make invalid states harder to express, and

separate a claim from the evidence that establishes it. Students should also learn to distinguish improving a reasoner's context from improving the representation of the engineering problem itself.

Coordination deserves greater emphasis as well. An engineer directing several autonomous workstreams must decompose a larger objective, identify dependencies and integration points, sequence work where necessary, monitor progress and risk, and reconcile independently produced results. These are familiar project-management competencies, but agentic capacity changes the scale at which they become technical concerns: an individual engineer may command enough parallel implementation capacity to face coordination problems that previously arose mainly at team scale. This does not make every software engineer a project manager. It makes decomposition, dependency management, sequencing, and integration part of the technical problem of converting abundant implementation capacity into coherent engineering progress.

Change what students demonstrate

A capstone should therefore demonstrate more than a working artifact. Ask students to bring the **engineering case**.[37] The artifact should be one component of that case, alongside requirements, models, tradeoffs, measurements, validation evidence, and an explicit argument for acceptance. Students should also demonstrate that they can organize substantial work: decompose an objective, manage dependencies and integration, coordinate parallel realization, and adapt the plan as engineering evidence arrives. This extends rather than replaces the integrative purpose of the capstone: the question is not merely whether students can produce a system, but whether they can organize and justify the engineering work that makes the resulting system worthy of acceptance.

The implication for program objectives is smaller than the change in daily work. In my view, most engineering programs need little change in **what they claim an engineer should become**. The larger change is in how students practice and demonstrate those capabilities.

The durable curriculum should not be organized around the current agent interface. Prompt conventions, harness architectures, model APIs, and context-management techniques matter in practice and students should use them, but they change too quickly to define the intellectual core. Teach the principles beneath them—finite reasoning and abstraction, representation and fidelity, action boundaries and authority, independent evidence, security, measurement, and the design of environments in which uncertain components can be used safely—and let students encounter current agents throughout the curriculum.

[37] Existing accreditation already asks for a culminating design experience that integrates standards and multiple constraints and builds on earlier coursework, with software-specific criteria adding requirements analysis, security, verification and validation, and processes for complex software systems[116]. Computing-accredited programs likewise require a major integrative project[117]. The open question is not whether students should integrate their knowledge but what should count as evidence that they have — which is why the capstone should carry the engineering case, not only the demo.

7.3.5 The Entry-Rung Problem

The traditional path into software engineering runs through implementation. Junior engineers become useful by writing code, and through that work they encounter larger systems, learn why designs fail, and gradually acquire the judgment needed to take responsibility for more of the system. GenAI changes this apprenticeship model. If routine implementation requires fewer people, the profession loses some of the work through which it has historically trained its future senior engineers.

This creates an entry-rung problem. Requirements discovery, modeling, security, systems reasoning, validation, and engineering judgment are harder to teach through a short framework-oriented path—and they are also the capabilities that become more valuable when implementation is abundant. Practitioners concentrated in implementation therefore need paths into architecture, assurance, modeling, and system responsibility. New entrants need ways to acquire those foundations even if the market demands less handwritten construction. The educational problem is not simply how to teach students to use GenAI, but how they acquire the engineering judgment that routine implementation previously helped them develop through experience.

MAGE shifts expertise toward choosing abstractions, defining properties, evaluating evidence, allocating authority, and remaining answerable for the system. Commodity intelligence will not define the future profession by leaving behind a stable residue of tasks for humans; those tasks will continue to move as capabilities improve. The more durable change is in how engineering work is organized. The profession is not shrinking toward whatever implementation tasks machines still perform poorly, but reorganizing around responsibility for the engineered whole.

What Cannot Be Delegated?

Making implementation cheaper does not make software engineering easy; it changes where the difficulty lies.

Across this Part, implementation has moved without making engineering responsibility disappear. Autonomous systems can assume more realization, analysis, transformation, and routine validation. The engineer's work moves toward deciding what must be true, choosing the representations through which consequential properties can be understood, establishing what evidence is sufficient, allocating authority, making tradeoffs, and determining whether the resulting system is fit to accept.

None of those categories defines a permanent list of tasks reserved for humans. Better machines may participate in all of them. The durable distinction is not between work a machine can perform and work it cannot. It is between delegating an act and delegating responsibility for the engineered whole.

The Preface asked what we can safely delegate. The Conclusion approaches the same boundary from the other direction:

What cannot be delegated?

Conclusion

The Part That Stays Yours

> Though much is taken, much abides.
> — Alfred, Lord Tennyson, *Ulysses*

The question we asked

In the Preface, we asked:

> How do we safely grant autonomy to commodity intelligence?

The question was not how to generate more code. Commodity intelligence had already made implementation abundant enough to expose the harder problem: if an agent can produce and modify software at extraordinary speed, what must the surrounding engineered environment provide before we can safely grant that capability useful autonomy?

Seven Parts later, the book has an answer.

Give humans and agents representations suited to the engineering questions they have to answer. Give important obligations authority where the environment has enough meaning and evidence to decide them. When experience reveals knowledge or judgment that future work should not have to rediscover, make the lesson durable, and maintain it only while it continues to earn its keep.

Modeling extends what humans, agents, and tools can reason about and creates surfaces for analysis and assurance. Through Alignment, selected engineering intent becomes authoritative: the environment determines what it will require, permit, observe, and admit. Governance conversion makes useful judgment durable beyond the task that produced it. When later work benefits from that structure, it becomes engineering capital.

That answer raises a converse question, and it is the one on which this book should end:

What cannot be delegated?

The answer in three moves

Strip MAGE to three engineering moves and this is what remains.

1. **Work at the right representation.** Choose the reduction that makes the engineering question answerable without carrying unnecessary detail. A purposeful model can expose properties that humans and agents would otherwise have to reconstruct from implementation, making them available for analysis, review, and evidence. Build the model when those benefits repay its cost, and maintain the correspondences on which its use depends.
2. **Put authority where the obligation is legible and enforceable.** Guidance aims the work; authority binds consequences. Some obligations can be held directly through types, permissions, tests, validators, or gates. Others become decidable only after a suitable representation exposes the relevant semantics. Leave consequential decisions to human judgment when the environment lacks the meaning or evidence needed to decide them adequately.
3. **Make useful judgment durable.** When a recurring or consequential engineering decision is worth carrying forward, encode it in a model, mechanism, architecture, procedure, or evidence that later work can inherit. When later work benefits from inheriting that structure, treat it as engineering capital: maintain it, reconcile it, and retire it when its return no longer justifies its carrying cost.

Together, these moves produce the **governed engineering environment**. Models carry what must be understood; mechanisms make repeatable decisions; evidence supports those decisions; human judgment remains responsible for the rest.

That is Model-Based Agentic Software Engineering.

In practice, start where engineering judgment is being purchased repeatedly. Find a system map that has to be reconstructed, a defect class that keeps returning, a review that asks the same question, or an obligation that lives only in someone's memory. Pick one recurring cost. Ask whether a better model, a stronger mechanism, or both would retire it. Make one durable improvement, observe what changes, and let the next bottleneck determine the next investment.

The part that stays yours

We began with a machine capable of building almost anything you describe—and only what you describe. It is unable to supply the engineering obligations you fail to make known to it. That capability is both the opportunity and the engineering problem.

A probabilistic reasoner does not become deterministic because an instruction asks it to. The engineered environment can still determine which models it reasons through, which actions it may take, what evidence its work must produce, which obligations bind its consequences, and which decisions remain with another authority.

The profession therefore cannot be defined by whatever tasks machines still perform poorly this year. Those tasks will move.

The durable boundary is responsibility.

You decide what should exist and what "correct" means for it. You remain answerable for how the system is represented, which properties those models preserve, what variation is acceptable and what choices should remain free, what evidence counts in support of those properties, which obligations receive authority, and which tradeoffs remain matters of judgment. You change the environment when its assumptions fail. You remain answerable for the resulting system even as more of the work required to build and analyze it is delegated.

Commodity intelligence can assist with all of those activities. It can propose models, generate evidence, run analyses, recognize familiar engineering moves, and help maintain the environment itself. What it does not remove is the need for someone to decide which claims matter, what variation they permit, what evidence is sufficient, which obligations should bind the work, and whether the resulting engineering case is acceptable.

The argument for MAGE therefore has two horizons. Today, Modeling and Alignment can make autonomous engineering cheaper and more reliable by reducing repeated inference and giving selected obligations independent authority. Future agents may reason cheaply and well enough to reduce those concerns substantially. That will be an exciting development, not a failure of the method. But the second commitment remains: engineers—and the societies that rely on them—must retain the authority to understand, challenge, govern, and accept the systems that machines help produce.

The future of software engineering is larger than faster implementation. As realization becomes cheaper, engineering work reorganizes around what remains scarce.

Classical definitions of software engineering remain substantially right. The discipline was never defined by typing code; it was defined by systematic, disciplined approaches to developing, operating, and maintaining software. Commodity intelligence changes the relative cost of that work. As realization becomes cheaper, engineers gain more leverage by representing the system, shaping its architecture, producing evidence, validating claims, placing authority, and exercising judgment.

Other engineering disciplines have long separated responsibility for engineering judgment from physically realizing the artifact. Software remains unusual: it is extraordinarily mutable, and many of its constraints are authored rather than imposed by physics. But the professional shape increasingly rhymes.

The vocabulary increasingly rhymes as well: models and obligations, tolerances and degrees of freedom, evidence and assurance, authority and responsibility. The correspondence is not exact; the underlying engineering concerns are.

The engineering environment need not remain limited to what human engineers could feasibly inspect, remember, or check by hand. If machines now perform much of the realization, they can also help maintain richer representations, run analyses over them continuously, and enforce obligations at scales that would previously have been uneconomic. The opportunity is not merely to let agents use the software-engineering machinery we already built. It is to build better machinery because agents can use it.

Commodity intelligence did not abolish software engineering. By making implementation cheaper, it may force us to practice more of it.

Appendix Part I — Practice

Using and Extending MAGE

This part turns from the argument to its use. The appendices that follow provide working references, methods, and extensions for applying MAGE in different engineering settings. They are meant to be consulted as needed, not read in sequence.

Table 9.0-1: The practice appendices — each beside the question that sends you to it.

If you are asking…	Go to
How do several mechanisms compose into a useful engineering capability?	Appendix A — Engineering Stacks
I've seen this problem before. What engineering move might travel?	Appendix B — Engineering Moves
What exactly does one of the book's teaching models contain or guarantee?	Appendix C — Model Reference
What should I inspect or do while operating a governed engineering environment?	Appendix D — Operator's Reference
How do I externalize a body of engineering judgment as a mastery-skill?	Appendix E — How to Write a Skill
How does MAGE's adoption change across a product's lifecycle surfaces?	Appendix F — Configuring MAGE Across the Product Lifecycle
How does an organization actually adopt GenAI — from assistance to bounded delegation?	Appendix G — Adopting GenAI in an Organization

The online MAGE Mechanism Catalog carries what print leaves out: the complete living repertoire of mechanisms, implementation variants, and deeper case realizations.

Appendix A: MAGE Engineering Stacks

Appendix A — MAGE Engineering Stacks

A single mechanism can eliminate one failure class. A useful engineering capability often requires several mechanisms.

A provenance record tells us who changed an artifact, but not whether every mutation was recorded. A model can make an engineering fact explicit, but nothing guarantees that consumers read the current version or that the implementation still matches it. A sensor detects a failure; detection alone does not restore service. So mechanisms compose.

What a Stack Is

An **engineering stack is a recurring set of mechanisms that together provide a useful engineering capability.** The stacks here are reference architectures, not prescriptions: a different substrate may provide the same capability with fewer, different, or stronger mechanisms.

Start with the capability:

> What must the environment be able to do, and which guarantees are required to make that claim true?

The seven stacks that follow answer that question for recurring engineering concerns.

The Seven, and How They Relate

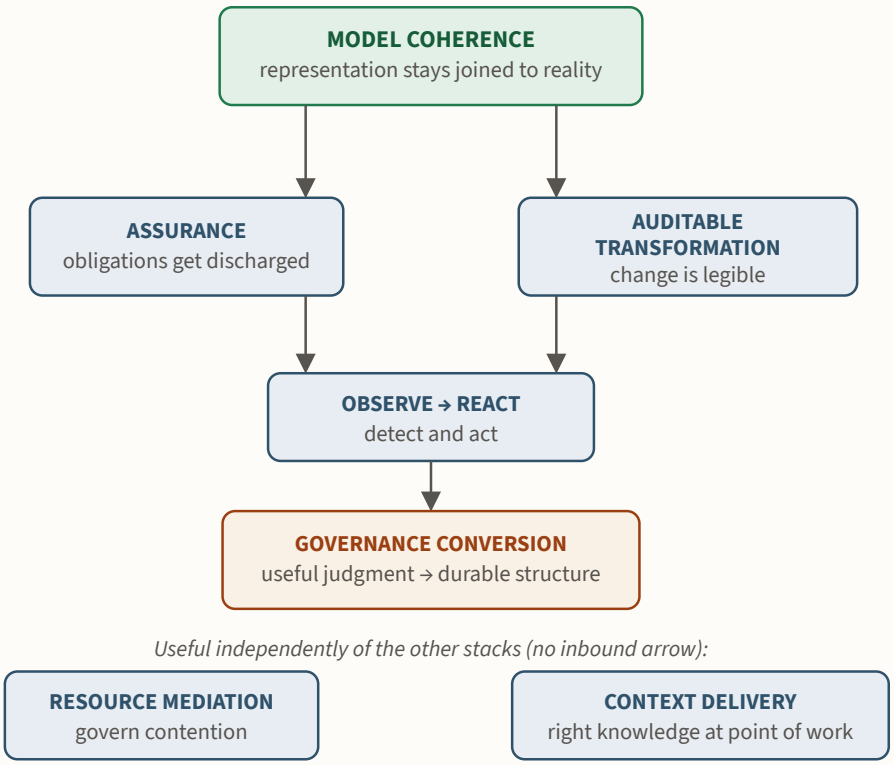


Figure A-1: Seven reference engineering stacks and their common relationships. Arrows show relationships among capabilities that recurred in one system, not a required architecture or adoption order. Resource Mediation and Context Delivery are shown separately because they provide useful capabilities independently.

Where to Look

A mechanism addresses a specific obligation or failure. A move names the transferable engineering judgment behind related mechanisms. A stack combines mechanisms into a reusable capability. Together, the stacks contribute to a governed engineering environment.

This appendix starts from a capability and asks which mechanisms must work together to provide it. Appendix B starts from recurring problems and shows how the same engineering move can produce different mechanisms in different settings. Each stack gives its capability, constituent moves, dependencies, and required composition. Concrete implementations of these moves appear in the companion MAGE Mechanism Catalog.

Why Stacks Exist

A stack is one form engineering capital can take when several mechanisms are needed to provide a capability. One failure may motivate a sensor. Another may expose a bypass that warrants a gate. A third may reveal missing provenance that warrants attribution. Mecha-

nisms introduced separately can eventually depend on one another strongly enough to form a coherent stack.

A Note on Implementations

The book describes mechanisms on a concrete coding-agent substrate because a real system makes them easier to see. The mechanism may be portable even when its implementation is not. For example, a harness lifecycle hook, middleware layer, git hook, CI stage, or wrapper process can each provide an enforcement point at which a deterministic step executes independently of agent cooperation.

A.1 Model Coherence

A.1.1 The Capability

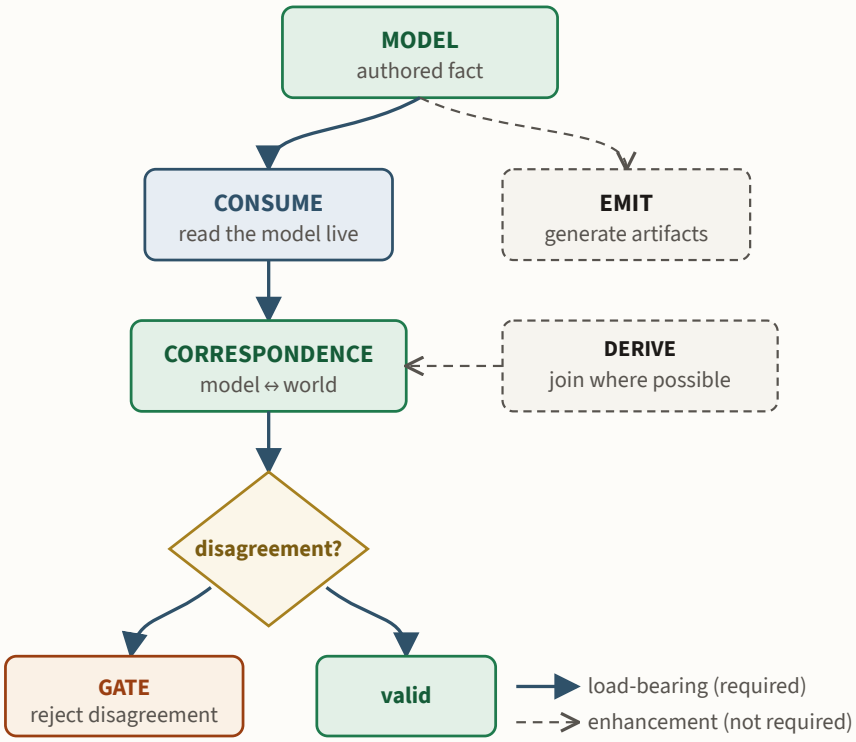
Keep explicit engineering models connected to both their consumers and the implementation they describe. A model remains useful only when people and agents read the current representation and the implementation continues to correspond to it.

A.1.2 When This Stack Earns Its Keep

Reach for it when:

- **Engineers or agents rely on explicit system models** to reason about the substrate.
- **Several consumers need the same engineering fact**, making copied facts sources of drift.
- **Implementation can diverge from authored intent** with nothing to catch the split.
- **Downstream artifacts can be generated** rather than hand-maintained in parallel.

A.1.3 The Composition



Solid path: the load-bearing composition. Dashed attachment: a useful enhancement, not required for the capability.

Figure A.1-1: The model-coherence composition. An authored MODEL feeds two paths: consumers CONSUME it live rather than copying its facts, and where the model owns the fact it can EMIT downstream artifacts. Consumption flows into a CORRESPONDENCE check — model against world — which DERIVEs its verdict from stable identities where an independent join exists; on disagreement a GATE gives the modeled property authority. Solid path: the required composition. Dashed attachment: a useful enhancement, not required for the capability.

A.1.4 Constituent Moves

Move	Role
MODEL	Put the engineering fact in an explicit representation.
CONSUME	Read the authoritative representation instead of copying its facts.
CORRESPOND	Compare model and implementation wherever an independent join exists.
DERIVE	Compute correspondence from stable identities where possible.
EMIT	Generate downstream artifacts when the model can own the fact.
GATE	Reject disagreement where the modeled property deserves authority.

A.1.5 Why These Travel Together

A model nobody consumes is documentation. A model whose consumers copy its facts becomes another source of drift. A model consumed but never checked against reality can go confidently wrong—trusted precisely while it lies.

Direct consumption avoids copied sources of truth; correspondence checks expose divergence between model and implementation; generation eliminates duplicate authority where the model can own a fact outright; and gates make consequential disagreements blocking. Not every model needs every mechanism: a descriptive model may remain advisory. Add stronger mechanisms only when the representation requires stronger authority.

Mechanisms: executable source-of-truth model · meta-model consumption · drift/parity gate · derived traceability · model-derived generation

A.2 Assurance

A.2.1 The Capability

State the engineering obligations, identify the population they apply to, and produce evidence strong enough for each property.

A.2.2 When This Stack Earns Its Keep

Reach for it when the important question shifts from

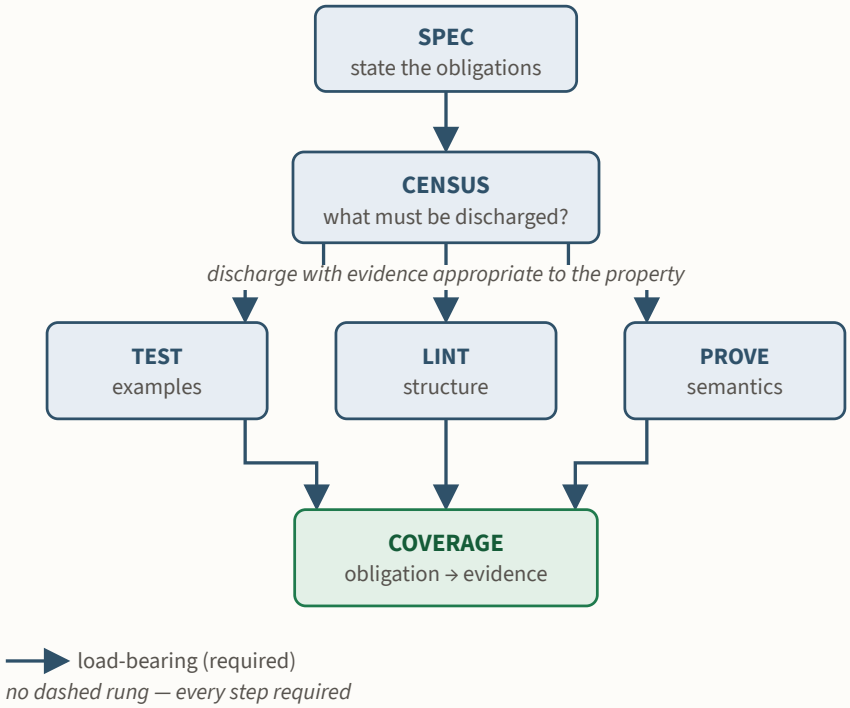
> Did the tests pass?

to

> Have we named the relevant obligations, discharged each one appropriately, and shown the population is covered?

That shift arrives whenever a property has to hold across a whole class of things — every mutator, every config field, every seam — and a green suite on a handful of examples no longer settles it.

A.2.3 The Composition



Solid path: the load-bearing composition. Discharge an obligation at the level where the property becomes legible.

Figure A.2-1: The assurance composition. A SPEC states the obligation; a CENSUS establishes the population it applies to; discharge fans out to the evidence each obligation deserves — TEST for examples, LINT for structure, PROVE for semantics — and all three lanes converge on COVERAGE, which joins each obligation back to its evidence so omissions show. Solid path: the required spine.

A.2.4 Constituent Moves

Move	Role
SPEC	State the obligation.
CENSUS	Establish the population to which it applies.
DISCHARGE	Apply tests, lints, proofs, or other evidence appropriate to the property.
COVER	Join obligations back to evidence so omissions become visible.

A.2.5 Why These Travel Together

A checker establishes a property only for what it checks. A proof says nothing about obligations left out of its population; a census with no evidence attached merely enumerates debt. Assurance therefore requires both evidence for each obligation and confidence that the relevant population is covered. Use evidence suited to the property—an example test where

an example suffices, a structural lint where structure is decisive, a bounded proof where semantics demand it—and map that evidence back to the full population so omissions remain visible.

One rule runs throughout: check an obligation at the semantic level where the property becomes legible. A property about a document's structure belongs in a check that reads structure, not one that scans bytes for it. This determines where the check belongs; it is not another mechanism in the stack.

Mechanisms: census-derived obligations · semantic validator · bounded proof · coverage gate

A.3 Auditable Transformation

A.3.1 The Capability

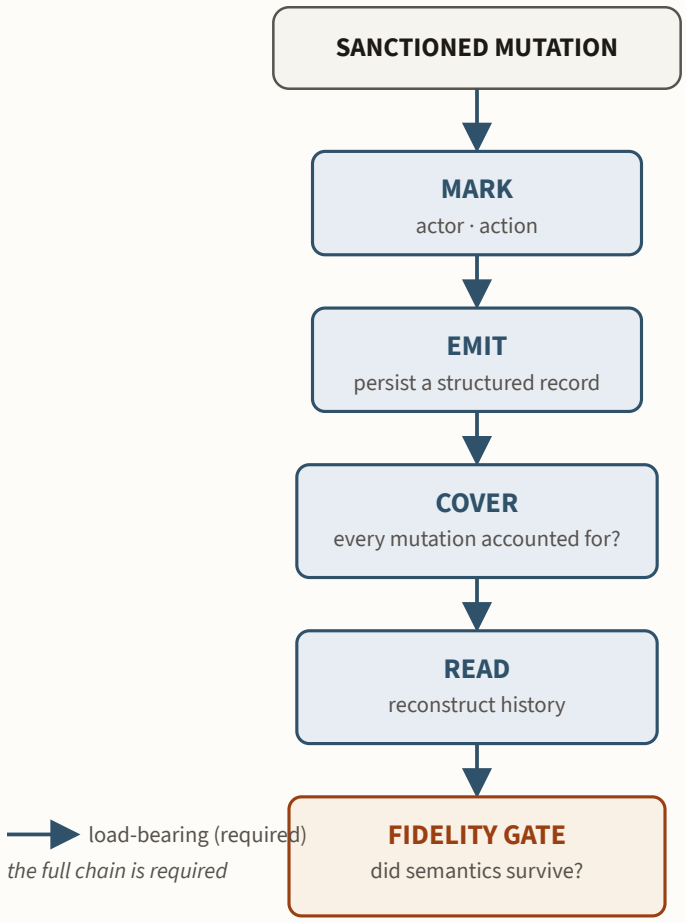
Make consequential transformations reconstructable and detect silent loss. Record who changed what, then verify that the transformed artifact retained the required semantics.

A.3.2 When This Stack Earns Its Keep

Reach for it where:

- **Multiple actors or agents mutate important artifacts**, making attribution and transformation history consequential.
- **Partial or bypassed instrumentation is plausible**, so "we log our writes" is not proof every write was logged.
- **Transformation can silently lose product semantics** — the output looks fine and is not.
- **Incident reconstruction requires both facts at once**: who changed what, and whether the result remained faithful.

A.3.3 The Composition



Solid path: the load-bearing composition. This joins causal legibility to product fidelity — broader than logging.

Figure A.3-1: The auditable-transformation composition. A sanctioned mutation flows through MARK (attach actor and action) to EMIT (persist a structured record); COVER detects any mutation that escaped attribution; READ reconstructs the transformation history from the records; a FIDELITY GATE checks that the transformed artifact kept its required semantics. Solid path: the core path, every step required.

A.3.4 Constituent Moves

Move	Role
MARK	Attach identity and context to the mutation.
EMIT	Persist a structured record.
COVER	Detect mutations that escaped attribution.

Move	Role
READ	Reconstruct the transformation history.
FIDELITY	Check that the transformed artifact retained its required semantics.

A.3.5 Why These Travel Together

Attribution without completeness produces a persuasive but partial history—convincing exactly where it is silent. Completeness without readable provenance proves only that records exist. And provenance alone cannot establish that a transformation preserved the product.

The stack must therefore establish both causal history and product fidelity: How did this artifact reach its current state, and did the transformation preserve its required semantics?

Mechanisms: provenance stamping · attribution coverage · fidelity gate

A.4 Observe → React

A.4.1 The Capability

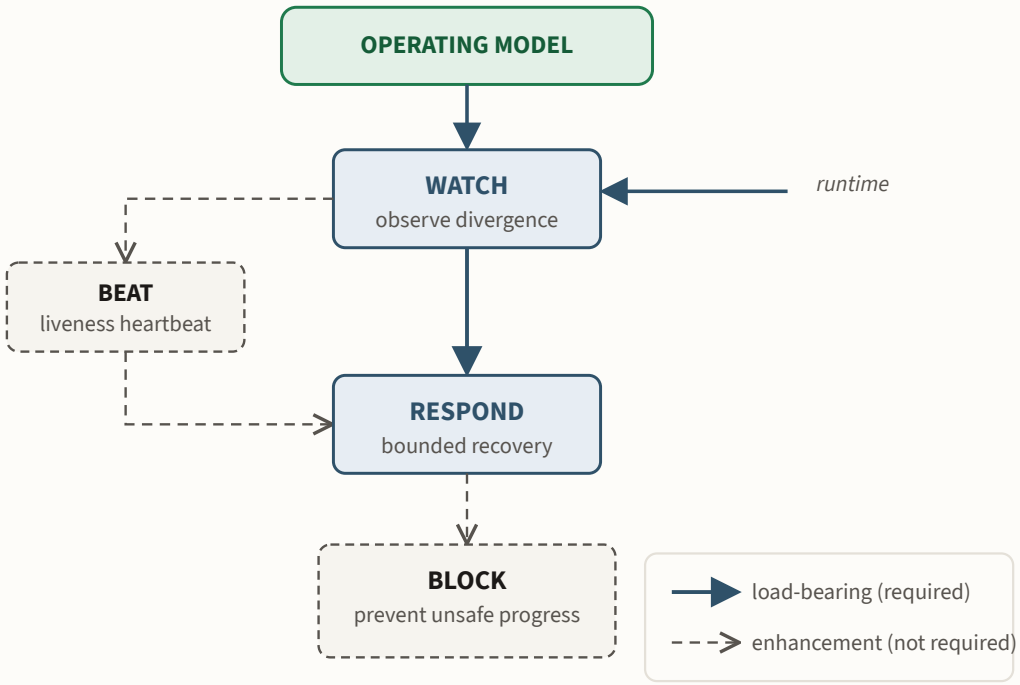
Detect when the running system diverges from its operating model and trigger a bounded response.

A.4.2 When This Stack Earns Its Keep

Reach for it when:

- **The system runs live**, and a divergence between expected and observed state has real cost.
- **Detection and response have drifted apart** — a dashboard nobody acts on, or a runbook nobody triggers.
- **Continuing can be more dangerous than stopping**, so an unsafe state needs the authority to halt progress, not just a note in a log.

A.4.3 The Composition



WATCH → RESPOND is the load-bearing path. Heartbeats and blocking authority strengthen it where the failure class requires them.

Figure A.4-1: The observe → react composition. An operating model drives WATCH, which reads runtime state; WATCH flows to RESPOND, which recovers. That solid path is required. Two dashed attachments strengthen it where the failure class demands: BEAT adds a liveness heartbeat so a hung process reads differently from a slow one, and BLOCK adds the authority to prevent unsafe progress. Solid path: the core path; dashed: enhancement. Dashed attachment: a useful enhancement, not required for the capability.

A.4.4 Constituent Moves

Move	Role
WATCH	Read runtime state against the operating model, so divergence surfaces as a signal.
RESPOND	Connect each signal to a bounded recovery action.
BEAT	*(strengthens)* Emit periodic liveness so a wedged process reads differently from a slow one.
BLOCK	*(strengthens)* Refuse unsafe progress where continuing is worse than stopping.

A.4.5 Why These Travel Together

Observation without response is a dashboard. Response without observation is a runbook waiting for someone to notice the fault.

WATCH and RESPOND form the core loop: observe divergence from the expected state, then trigger a bounded response. Add a heartbeat when silence must be distinguished from slow progress; add blocking authority when continuing is more dangerous than stopping.

Mechanisms: watchdog · heartbeat · bounded recovery · fail-closed gate

A.5 Resource Mediation

A.5.1 The Capability

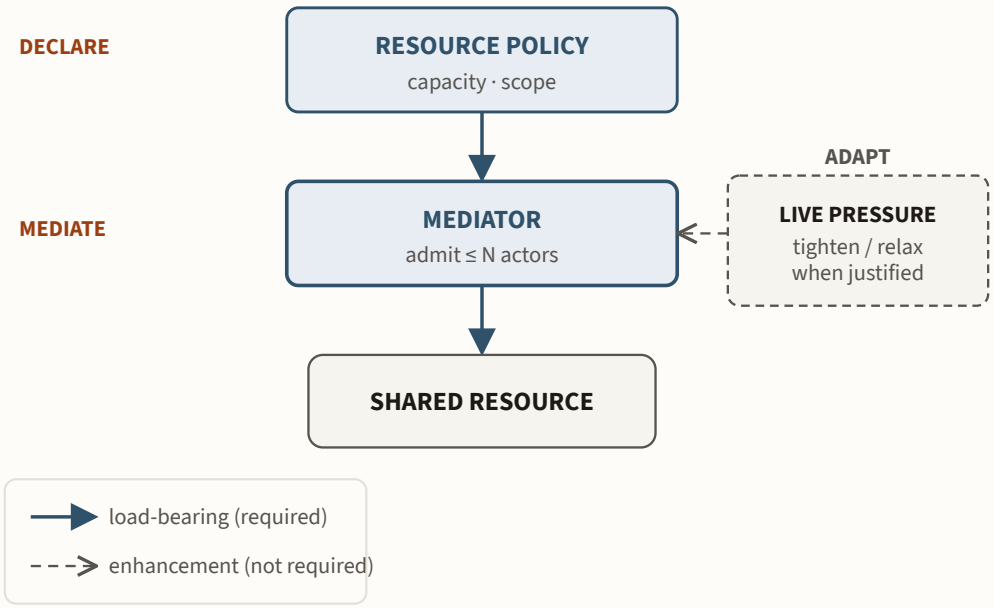
Prevent actor timing from becoming accidental concurrency policy for a shared resource. When actors contend for a scarce resource, move admission policy out of their timing and into an explicit control.

A.5.2 When This Stack Earns Its Keep

Reach for it when:

- **Actors contend for a scarce resource** — a test runner, a build host, a rate-limited API — and their arrival order decides who wins.
- **Availability has become an emergent property** of when jobs happen to start, rather than a stated policy.
- **Capacity should flex under live pressure**, but only once a fixed policy and a single admission point already exist.

A.5.3 The Composition



Solid path: the load-bearing composition. Dashed attachment: a useful enhancement, not required. N=1 and N>1 are settings of the same mediator, not separate stacks.

Figure A.5-1: The resource-mediation composition. A RESOURCE POLICY states which resource is scarce and what capacity is acceptable; a MEDIATOR admits at most N actors through one admission point to the SHARED RESOURCE. A dashed LIVE PRESSURE loop tightens or relaxes effective capacity when justified. N=1 and N>1 are settings of the same mediator, not separate stacks. Solid path: the required path. Dashed attachment: a useful enhancement, not required for the capability.

A.5.4 Constituent Moves

Move	Role
DECLARE	State which resource is scarce and what capacity is acceptable.
MEDIATE	Route access through one admission point.
ADAPT	*(strengthens)* Modify effective capacity from live pressure where justified.

A.5.5 Why These Travel Together

Unmediated contention makes resource availability an emergent property of actor timing—whoever starts first wins, and the policy is an accident. A mediator replaces that accident with explicit policy: one admission point and a declared capacity. The engineering move is not the semaphore; it is moving contention policy out of individual actors and into one

shared control. Whether that control admits one actor or many is a setting, not a different mechanism.

Adaptive control can improve utilization, but only after a fixed policy and common admission point exist. Capacity cannot be tuned coherently when no single control owns it.

Mechanisms: exclusive mediator · bounded mediator · pressure-responsive admission

A.6 Governance Conversion

A.6.1 The Capability

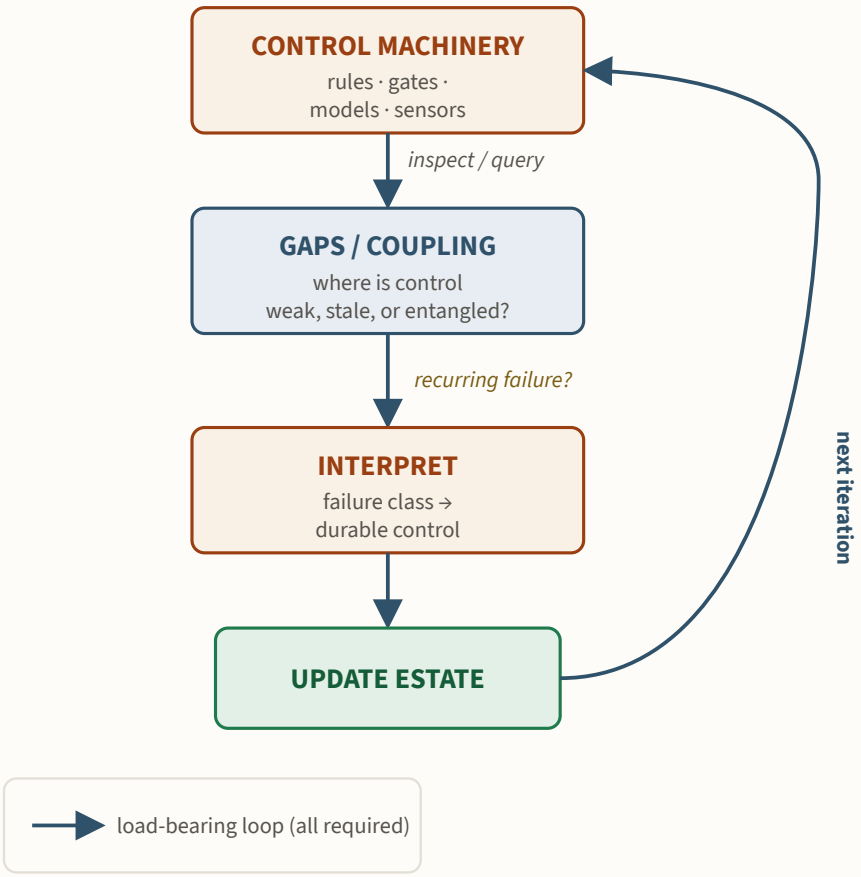
Turn recurring failures and judgments into durable engineering structure while keeping the control machinery legible. Model how controls relate, expose gaps and coupling, and convert recurring failures into new or stronger controls.

A.6.2 When This Stack Earns Its Keep

Reach for it when:

- **The controls have grown their own complexity** — they overlap, depend on one another, go stale, and carry blast radius nobody has mapped.
- **The same failure keeps recurring**, and each fix is a one-off patch rather than a durable control.
- **Lessons live in institutional memory** instead of in the environment, so they leave when the person who learned them does.

A.6.3 The Composition



Solid loop: the load-bearing composition. Four moves, not six mandatory stages.

Figure A.6-1: The governance-conversion composition, a loop. The CONTROL MACHINERY — rules, gates, models, sensors — is inspected and queried to expose GAPS / COUPLING: where control is weak, stale, or entangled. When a failure recurs there, INTERPRET converts the failure class into a durable control, and UPDATE MACHINERY folds it in; a solid feedback edge returns to the machinery for the next iteration. The four moves form the core loop.

A.6.4 Constituent Moves

Move	Role
MODEL	Make the control machinery explicit — rules, gates, models, sensors.
EXPOSE	Query the machinery for gaps, staleness, and coupling.
CONVERT	Interpret a recurring failure class into a new or strengthened control.
UPDATE	Fold the control back into the machinery; the model changes with it.

A.6.5 Why These Travel Together

As a governed environment grows, its controls overlap, depend on one another, go stale, and acquire blast radius. Eventually the control machinery itself becomes difficult to reason about. When that happens, model it explicitly.

Once the machinery is explicit, interpret recurring failures against it. A failure worth converting becomes a new or stronger mechanism, and the machinery model updates with it. This creates engineering capital: the environment changes so future work no longer depends on someone remembering the lesson. Rule registries, dependency graphs, indexes, and metadata are possible implementations.

Mechanisms: control census · control-dependency graph · governance conversion

A.7 Context Delivery

A.7.1 The Capability

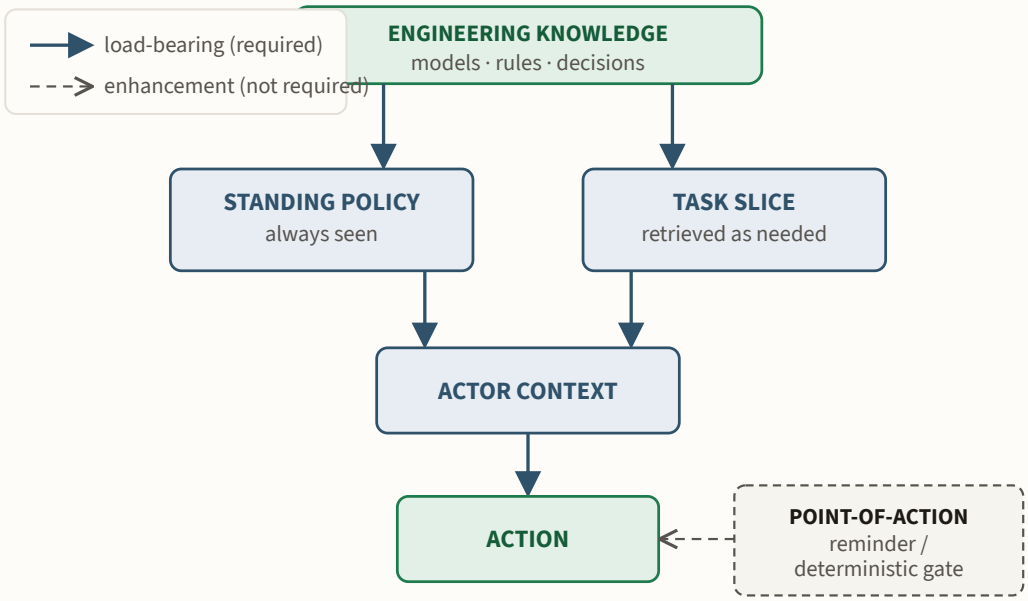
Put relevant engineering knowledge and obligations into the actor's context before the decision they must inform.

A.7.2 When This Stack Earns Its Keep

Reach for it when:

- **An actor must reason from engineering knowledge it does not carry** — models, rules, prior decisions.
- **The whole environment will not fit** in the actor's working state, and dumping all of it into context merely relocates the navigation problem.
- **Some obligations are critical and decidable**, and leaning on the actor to remember them is not good enough.

A.7.3 The Composition



Solid path: the load-bearing composition. Dashed attachment: a useful enhancement, not required. Context is a reasoning aid; critical decidable properties graduate into deterministic controls.

Figure A.7-1: The context-delivery composition. Engineering knowledge — models, rules, decisions — splits into a STANDING POLICY that is always seen and a TASK SLICE retrieved as needed; both feed the actor's context, which drives the action. A dashed POINT-OF-ACTION attachment reasserts important obligations at the moment of action, and for critical decidable obligations becomes a deterministic gate rather than a reminder. Solid path: the essential composition. Dashed attachment: a useful enhancement, not required for the capability.

A.7.4 Constituent Moves

Move	Role
STANDING	Supply compact policy that applies broadly.
SLICE	Retrieve the task-relevant model and engineering context.
DELIVER	Put that context into the actor's reasoning horizon.
REINFORCE	*(strengthens)* Reassert important obligations at the point of action where useful.
GATE	For critical decidable obligations, do not rely on delivery at all — enforce them.

A.7.5 Why These Travel Together

An actor cannot reason through knowledge it never sees. But loading the entire engineered environment into every context only moves the navigation problem downstream: now the actor must find the relevant fact in a wall of them. Context delivery therefore combines compact standing policy with task-specific retrieval.

Point-of-action reminders can reinforce obligations that still require judgment. Critical decidable obligations should instead become deterministic controls: context is a reasoning aid, not authority. Boot files, dynamic snippets, hooks, and nudges are current implementations; the portable pattern is standing policy, task-specific retrieval, and deterministic enforcement where warranted.

Mechanisms: standing context · dynamic context injection · point-of-action guidance

A.8 Composing a Stack

The seven preceding stacks are examples. The reusable method is how to compose one. Start from the capability, not the mechanisms.

A.8.1 1. Name the Capability

Prefer

> Every consequential mutation is reconstructable.

over

> We need provenance logging.

The first states the engineering result. The second has already chosen a mechanism, before you know whether it is the right one or the only one.

A.8.2 2. Enumerate the Failure Classes

Ask what would make the capability claim false. For reconstructable mutation:

- the actor is unknown;
- a record is missing;
- a bypass path exists;
- history cannot be joined;
- the transformation silently loses semantics.

Those failures determine what the composition must cover.

A.8.3 3. Assign One Guarantee to Each Failure

Map each way the claim can break to the guarantee that closes it.

Failure	Guarantee
actor unknown	attribution
record missing	emission
bypass exists	coverage
history fragmented	stable identity / join
semantic loss	fidelity check

Do not add a mechanism because the case you are copying from used it. Add it because a failure class needs its guarantee.

A.8.4 4. Find the Dependency Among Guarantees

Some guarantees only become meaningful once another exists.

MARK → EMIT → COVER → READ

A completeness check is meaningful only after the population to be accounted for has been defined. A list of mechanisms becomes a stack when their guarantees depend on one another.

A.8.5 5. Separate the Load-Bearing Path from the Enhancements

For each mechanism, ask:

> If I remove this, is the capability claim still valid?

If yes, the mechanism may still be valuable, but it is not load-bearing. The diagrams represent this distinction with solid paths and dashed attachments.

A.8.6 6. Match the Substrate

Do not reproduce a reference stack mechanically. A database, an embedded controller, a compiler, a SaaS application, and a research codebase each expose different observable facts and different deterministic seams. Preserve the guarantees; let the substrate decide the mechanisms.

A.8.7 7. Stop

The objective is not maximum governance. Stop when the smallest required stack makes the capability claim true for the failure classes that matter.

A.8.8 The Method in One Picture

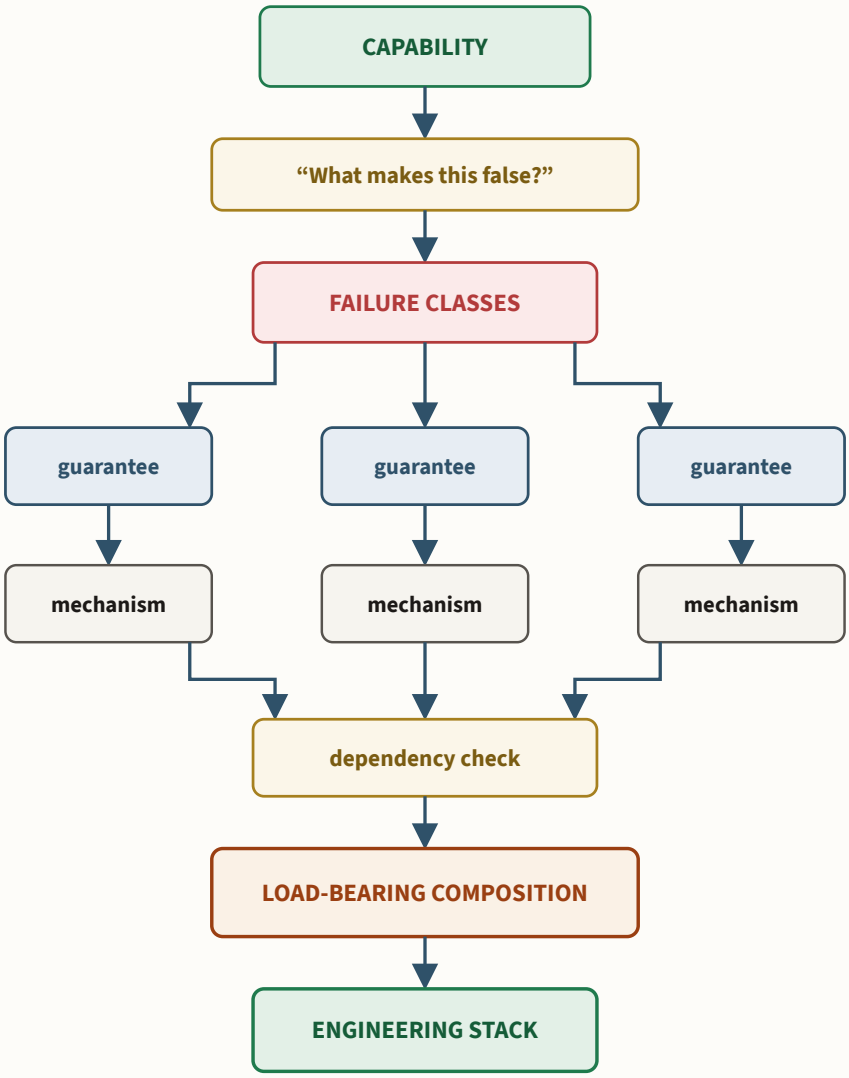


Figure A.8-1: Composing an engineering stack. Begin with a capability and its failure classes. Identify the guarantees required to close those failures, select mechanisms that provide them, determine their dependencies, and retain the smallest required composition that makes the capability claim valid.

Appendix B: Engineering Moves

Appendix B — Engineering Moves

This appendix shows how the same engineering judgment can produce different mechanisms in different systems.

Each section starts from a recurring engineering problem, names the move that addresses it, and shows two realizations.

Read each page in one direction:

Problem → Move → two realizations.

A move is portable engineering judgment: close an action surface, derive obligations from a model, deliver policy where a decision occurs. A mechanism is one concrete realization of that move: a typed mutation API, a coverage census, a context-injection hook. Mechanisms vary by system; the move travels.

Appendix A shows how mechanisms compose into engineering capabilities. This appendix shows how one move can take different forms. The online MAGE Mechanism Catalog contains the broader implementation repertoire.

Figure B-1 maps each recurring problem to its corresponding engineering move.

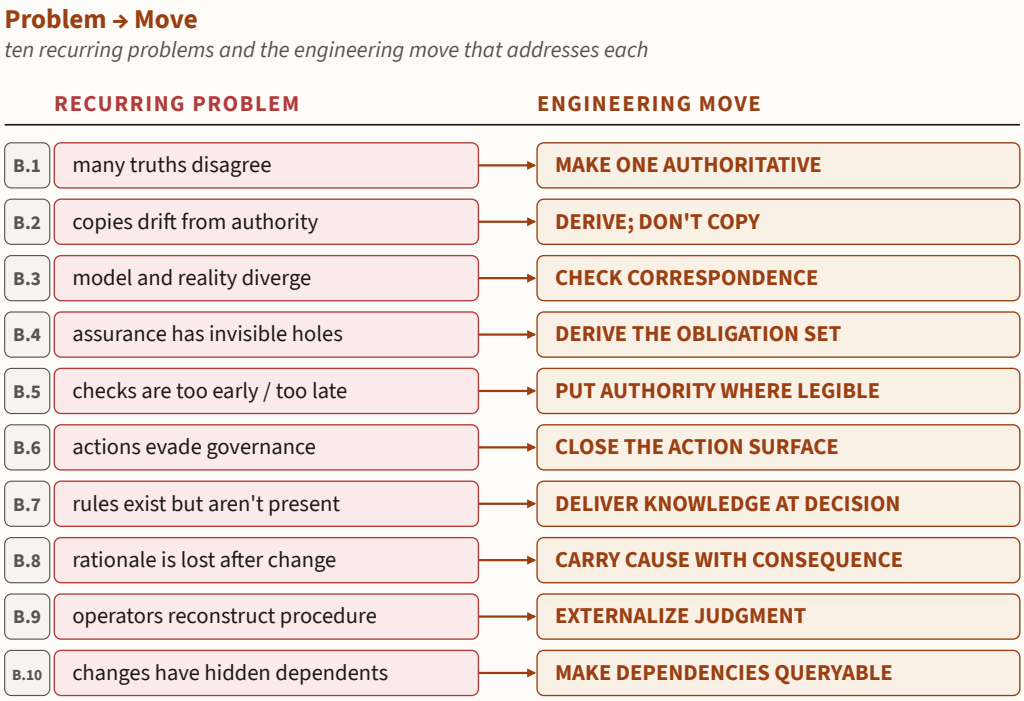


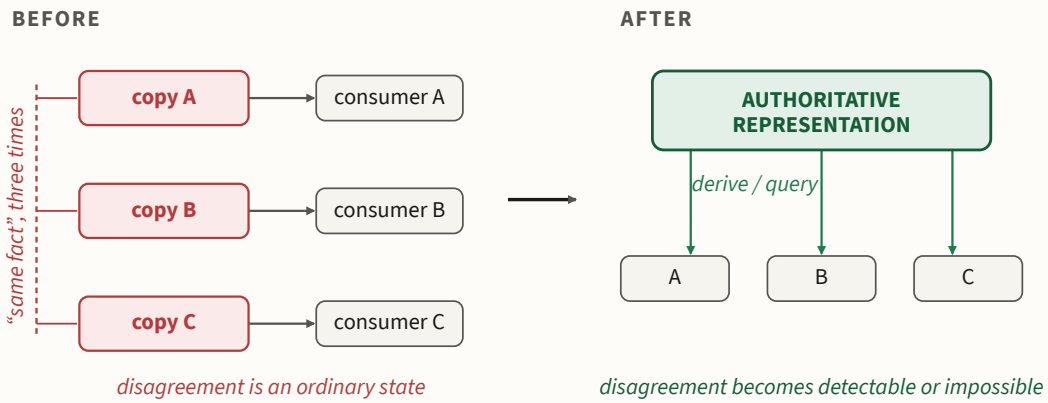
Figure B-1: Ten recurring engineering problems and the moves that address them. *The moves apply the method developed in Parts II–IV: model knowledge people repeatedly reconstruct, give decidable obligations authority, and make recurring judgment durable when doing so is worth the cost.*

B.1 Make One Source Authoritative

Problem. Consequential facts are duplicated across consumers. A service relationship may appear in deployment configuration, policy, documentation, and tests; a timeout may appear in callers, workers, and orchestration. Once those copies can change independently, disagreement becomes an ordinary state.

Move. Choose one machine-readable representation as the authoritative source of the fact. Downstream consumers should query it or derive from it rather than maintain independently editable copies.

Figure B.1-1 contrasts the two states.



Choose one representation to carry the fact, make it machine-readable, and derive the consumers from it.
Downstream artifacts derive from or query the authoritative representation rather than maintaining independent copies.

Figure B.1-1: One authoritative representation. *BEFORE — copies A, B, C each feed a consumer and can drift apart, so disagreement is possible. AFTER — one authoritative representation, with A, B, C derived or queried from it, so disagreement becomes detectable or impossible.*

Example — Service policy. DocAble represents permitted service flows in a structured service model. Network policy and wiring derive from that model rather than maintaining independent accounts of permitted communication. A service relationship is authored once and propagated to its consumers.

Example — Timeout ordering. Scattered wall-clock budgets are represented in one timeout-budget model whose nesting relation is checked mechanically. The model makes the required ordering explicit: inner budgets must fit inside outer ones. Without it, that rule remains scattered across implementations.

Related mechanisms: Service-flow / API model · Executable Source of Truth · Model-driven codegen · Timeout-budget ordering model · Required-configuration-per-role manifest · Synchronization model.

B.2 Derive; Don't Copy

Problem. An authoritative source loses authority when consumers copy its values into independently maintained snapshots. The model may remain nominally authoritative while engineering decisions operate on stale copies.

Move. Consume authoritative representations by query or derivation rather than creating a second editable copy.

Figure B.2-1 sets the snapshot against the query.

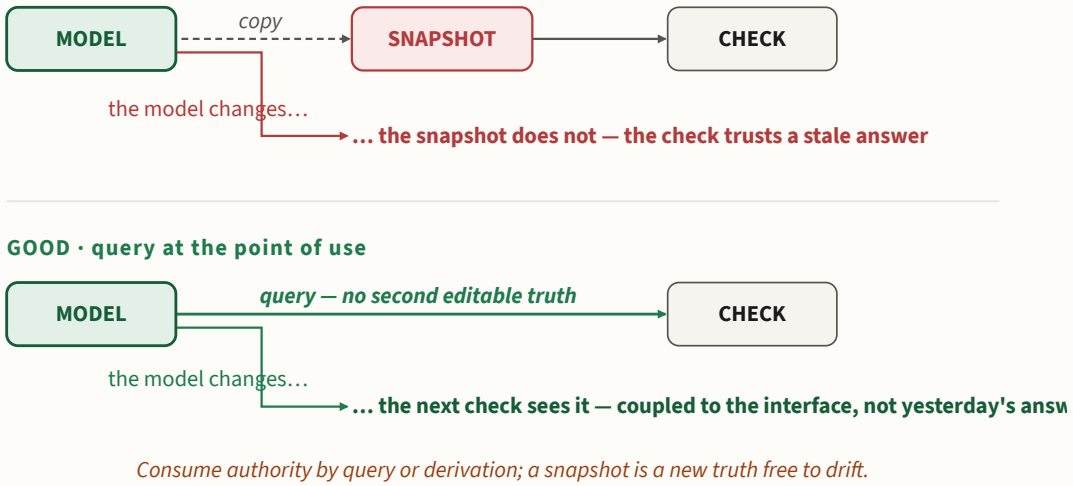


Figure B.2-1: Query, don't snapshot. *BAD — the model is copied into a snapshot that a check reads; the model changes, the snapshot does not. GOOD — the check queries the model directly, so the next check sees the change.*

Example — Live model consumption. Model-aware lints, tests, and agent briefs query the current representation instead of embedding copied values. When the modeled architecture changes, consumers see the new state through the model interface; there are no stale copies to find and update.

Example — Generated artifacts. Consumers that should not query at run time can instead be generated from the authoritative model at build time. The downstream artifact is still derived from the model rather than maintained as a second source of truth.

Related mechanisms: Meta-model consumption discipline · Model query surface · Model-driven codegen · Doc-hygiene provenance.

B.3 Keep Representation and Reality in Correspondence

Problem. A model becomes dangerous when engineers and agents continue to trust it after it no longer corresponds to the system it represents.

Move. Treat correspondence as an invariant. Check both that reality still satisfies modeled claims and that consequential implementation elements have not appeared outside the model's declared scope.

Figure B.3-1 shows the two directions.

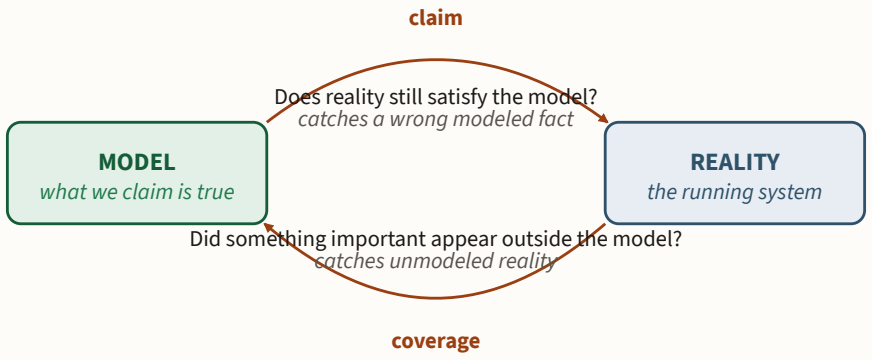


Figure B.3-1: Correspondence runs both ways. *A two-way loop joins MODEL and REALITY. One arrow asks whether reality still satisfies the model, catching a wrong modeled fact; the other asks whether something important appeared outside the model, catching unmodeled reality.*

Example — Architecture drift. A component-and-zone model declares where code belongs and which dependencies it may take. A gate compares the declared architecture against the imports the implementation actually makes. A change that violates the modeled structure fails the check at commit time.

Example — Governance coverage. The same correspondence idea runs the other way. An orphan-coverage walk starts from code and asks what model or control governs it. Uncovered code is not automatically wrong; it shows where implementation currently sits outside the modeled governance estate and requires an explicit decision.

Related mechanisms: Component & zone model · Drift & parity gates · Cross-source coherence lints · Orphan-coverage metric · Symbol-anchored traceability.

B.4 Derive What Must Be Assured

Problem. Measures of existing evidence—tests written, lines covered, checks passing—cannot reveal obligations that were never encoded. Before measuring coverage, establish what evidence should exist.

Move. Derive what must be assured from the engineering model, then compare that required set with the evidence that exists.

Figure B.4-1 traces the required set against the evidence.

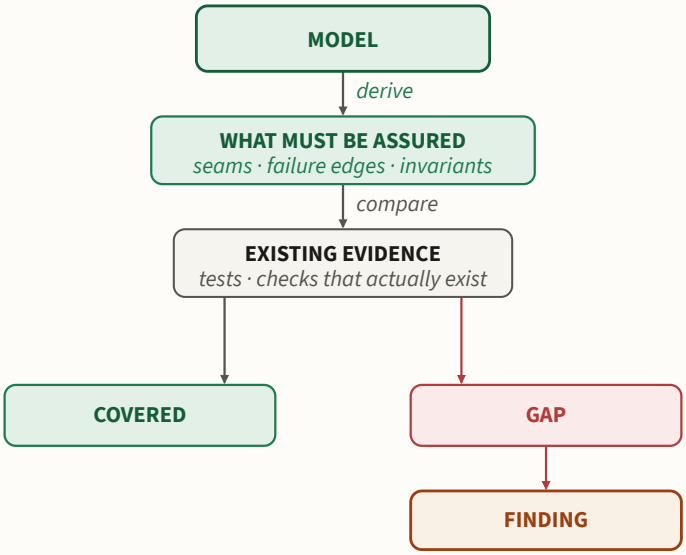


Figure B.4-1: Required set minus present evidence. *The model derives what must be assured; that required set is compared against existing evidence; the intersection is covered, and the remainder is a gap that raises a finding.*

Example — Test obligations. Structured models expose seams, failure edges, and invariants. Those elements generate a census of what should be tested. Compare that census with the existing suite, and every missing test becomes mechanically visible and can block the gate.

Example — Governance obligations. The same move applies to controls. When governance targets are explicit, the environment classifies controls by what they protect and surfaces every target with no control — or only advisory guidance. Here the evidence is governance mechanisms rather than tests, but the move is the same: derive the required set, compare it with what exists, and report what is missing.

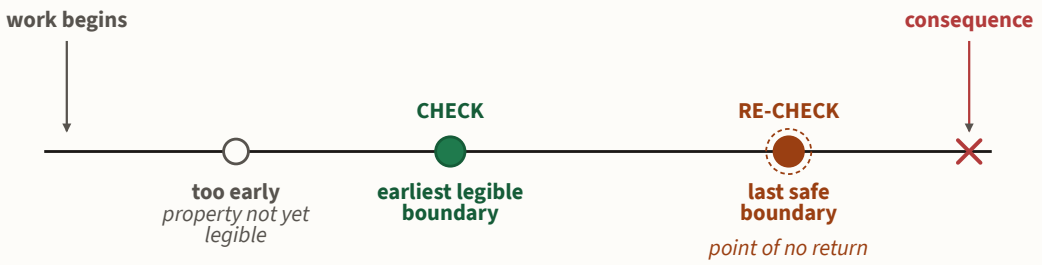
Related mechanisms: Model-derived test-obligation census · Coverage → model-node mapping · Control-coverage census · Governance Graph · Journey-criticality → test-tier placement.

B.5 Put Authority Where the Property Becomes Legible

Problem. An obligation can be checked too early, before the relevant property is observable, or too late, after avoidable cost or consequence has occurred. Later changes can also make earlier evidence stale.

Move. Enforce at the earliest boundary where the property is decidable. If later work can invalidate that evidence, re-establish it at the last safe boundary before consequence.[38]

Figure B.5-1 places both boundaries on one timeline.



orce at the earliest boundary that can decide the property; if later work can invalidate the evidence, re-establish it at the l

Figure B.5-1: Earliest legible, last safe. *A timeline runs from where work begins to consequence. Too-early sits where the property is still invisible; the earliest-legible boundary carries the first check; the last-safe boundary carries a re-check just before the point of no return.*

Example — Abort early. The sentinel first-commit mechanism inspects a newly dispatched agent at its first meaningful commit. That point is early enough to stop bad work before it travels far, but late enough that an artifact exists to check.

Example — Re-prove at completion. Epic completion sits at the other end. Earlier tests may have passed before later integration made their evidence stale. The final Definition-of-Done re-runs the required checks against HEAD before closure, re-establishing evidence at the consequential boundary.

Related mechanisms: Sentinel first-commit early-abort · Pre-commit hook · Staged deploy gates · Epic Definition-of-Done (Final-Opus rerun) · Enforce at the right semantic level.

[38] The analogy to time-of-check/time-of-use (TOCTOU) is useful: a valid check does not justify a later action if the relevant state can change between check and use. Here the intervening change may be another commit, an integration step, a generated artifact, or a deployment transition. See Part IV for the fuller treatment.

B.6 Close the Action Surface

Problem. When a consequential operation can occur through arbitrary routes, governance must account for an open-ended set of ways to change the system. Validators, provenance, and policy cannot cover paths they cannot enumerate.

Move. Route consequential actions through a bounded, named interface. A closed action surface exposes a finite set of operations, each of which can carry policy, provenance, and validation.

Figure B.6-1 sets the open surface against the closed one.

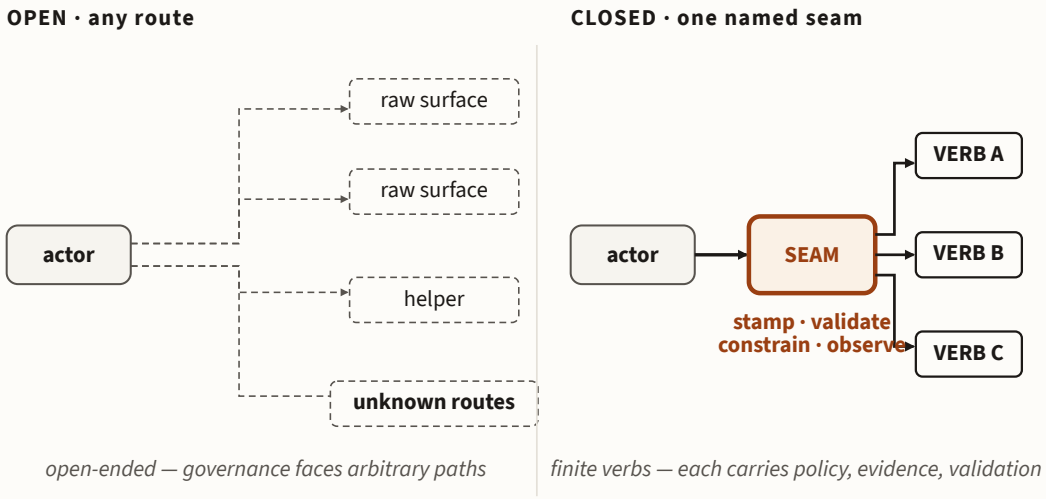


Figure B.6-1: Open surface versus closed seam. *OPEN — an actor reaches a raw surface by many routes, including unknown ones. CLOSED — the actor passes through one seam that exposes a small set of named verbs, each able to stamp, validate, constrain, and observe. The distinction is encoded by shape, line style, and weight rather than color.*

Example — Document mutation. DocAble routes remediation through a closed set of named mutator verbs. Because the verb set is finite, every verb can stamp provenance, register inserted content, and run the required checks. New capabilities must extend the verb set rather than introduce another ungoverned path.

Example — Infrastructure access. Raw queue operations are confined to one dispatch seam. Queue semantics and atomicity are encoded and reviewed at that seam rather than reconstructed at each call site.

Related mechanisms: Closed remediation-verb sets · PdfModel · Sole raw-Redis seam · ServiceClient · Canonical walkers · One Door Enforced.

B.7 Deliver Knowledge Where the Decision Occurs

Problem. A rule cannot influence a decision if the actor does not encounter it in time. As the knowledge base grows, relying on the actor to search the full corpus becomes increasingly unreliable.

Move. Deliver the knowledge needed for the current work where the decision occurs; do not rely on the actor to find it in the full corpus.

Figure B.7-1 shows the selection at the decision point.

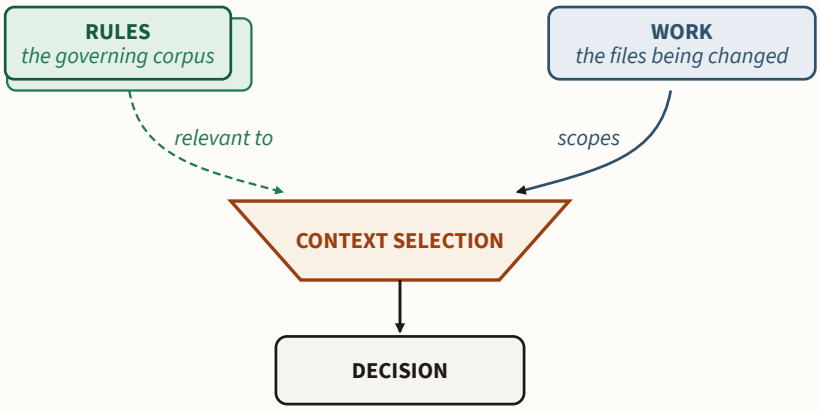


Figure B.7-1: Knowledge delivery at the decision point. *Context selection joins the current work with the rules relevant to its scope and supplies that subset to the decision. The distinction is encoded by shape and line style rather than color.*

Example — File-scoped context. Dynamic context injection selects the rules governing the files in a task and injects those rules into the agent's brief. The relevant rules arrive with the assignment rather than leaving the agent to find them in the full governance corpus.

Example — Dispatch admission. Brief linting applies the same principle one step earlier. A task cannot launch until its brief contains the required markers and context.

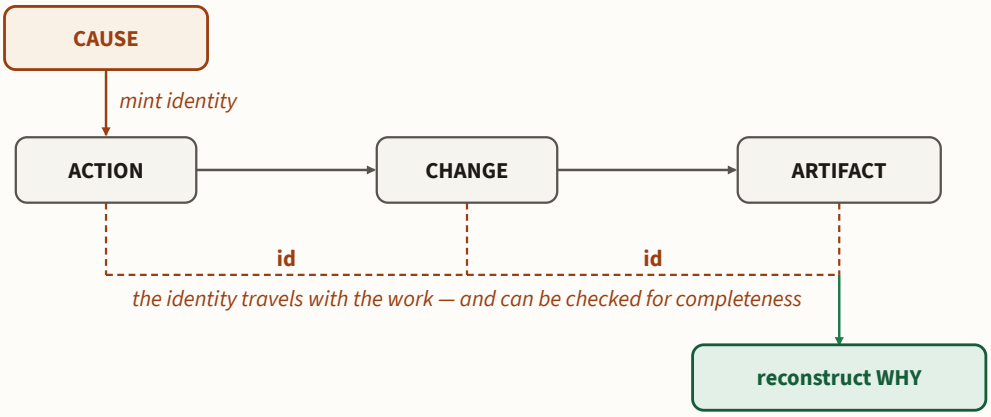
Related mechanisms: Dynamic context injection · Governed Knowledge Base · Brief linting · Mandatory snippet-table enforcement · Point-of-action policy delivery.

B.8 Make Cause Travel with Consequence

Problem. After work propagates through commits, generated artifacts, and transformations, the final state may preserve what changed while losing why it changed. Reconstructing causation afterward is expensive and often ambiguous.

Move. Capture provenance at the causal event and carry it forward with the resulting change. Where possible, check it for completeness so missing links are detected.

Figure B.8-1 follows the identity from cause to consequence.



Capture provenance at the causal event and carry it forward with the resulting change.

Figure B.8-1: Cause travels with consequence. *A cause mints an identity; the identity rides action to change to artifact; from the final artifact the identity lets a reader reconstruct why, not just what.*

Example — Agent changes. When an agent makes a change, the environment records a typed caused-by relation to the originating task or reason and requires that relation at commit. The reason is recorded at the causal event rather than inferred later from the diff.

Example — Artifact remediation. Document mutation uses a different carrier. Individual mutators stamp their changes, and tooling reconstructs a changelog from those stamps. A wiring lint fails when a sanctioned mutation verb lacks attribution. Provenance completeness becomes a build-enforced invariant rather than a convention.

Related mechanisms: Caused-by provenance · Per-mutator attribution stamps · Derive-changelog · The `a11y_` inserted-artifact convention · F10 mutator-stamp-wiring lint.

B.9 Externalize Recurring Operational Judgment

Problem. Operational work remains expensive when each incident requires an operator to reconstruct the same judgment: what healthy state looks like, what a symptom means, which procedure applies, and what evidence confirms recovery.

Move. Make recurring operational judgment explicit. Generate or execute what can be mechanized; provide guidance where judgment remains necessary; and keep both tied to the system they describe.

Figure B.9-1 traces the system model into the operational model.

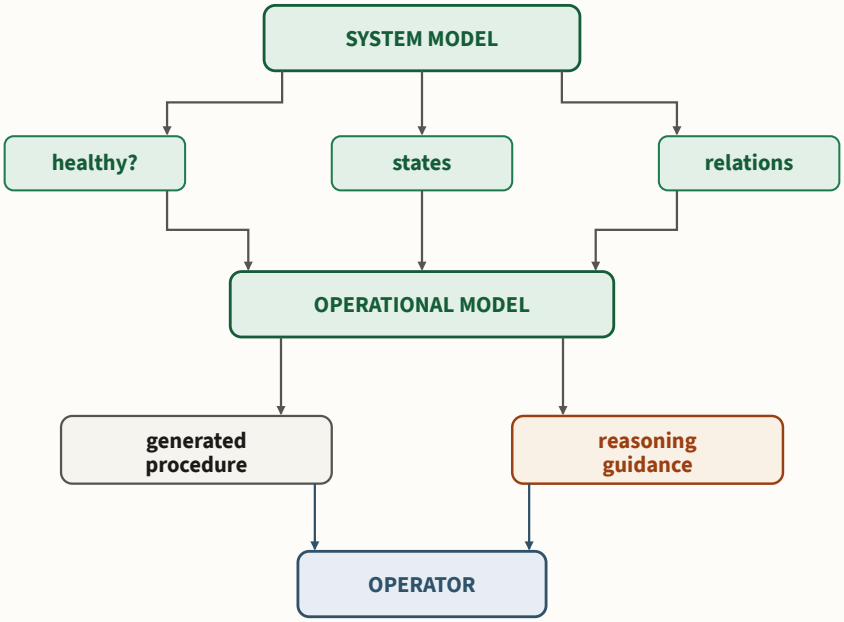


Figure B.9-1: From system model to operational guidance. *Healthy-state predicates, states, and relations feed an operational model that produces generated procedure where possible and reasoning guidance where judgment remains necessary.*

Example — Generated runbook. DocAble's lifecycle model names operational subsystems and their healthy-state predicates. The operator runbook is generated from that representation, keeping procedure aligned with the system model. When a subsystem's healthy-state predicate changes, the generated runbook changes with it.

Example — Event-bound playbook. The orchestrator reacts to typed fleet events through per-topic playbooks. The playbook records the trigger and response once, so the orchestrator can execute them when the event recurs rather than reconstructing the procedure each time.

Related mechanisms: Lifecycle model → generated runbook · Operational playbooks · Operator runbook skill · Encoded Operational Judgment · Orchestrator-as-reactor.

B.10 Make Hidden Dependencies Queryable

Problem. Cross-cutting mechanisms depend on assumptions about the substrate they inspect. When those dependencies remain implicit, a substrate change can either break many controls at once or silently invalidate part of their coverage.

Move. Represent consequential dependencies explicitly so the impact of a proposed change can be queried before it lands.

Figure B.10-1 fans a change out to its dependents.

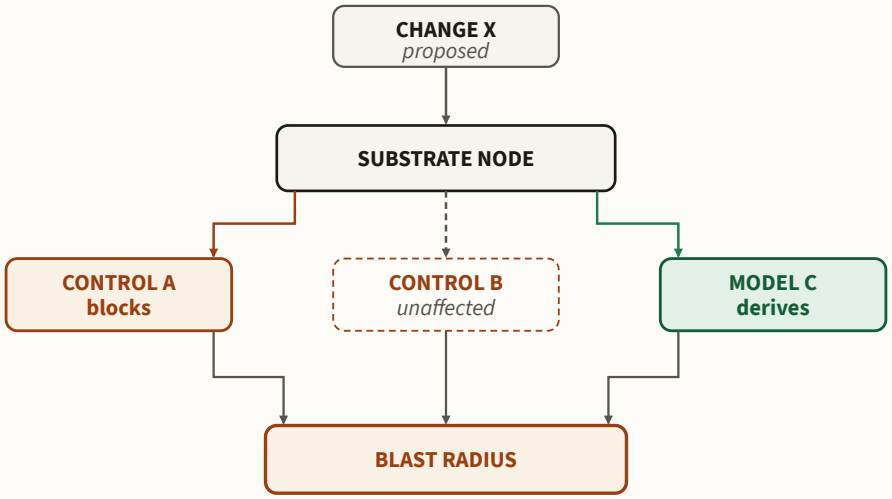


Figure B.10-1: Impact as a graph query. *A proposed change reaches a substrate node; typed dependency edges identify the controls and models that depend on it, and their union defines the affected set.*

Example — Control blast radius. Each control declares its substrate dependencies as structured metadata. A query identifies which controls depend on a proposed substrate change and how each relation should be interpreted.

Example — Traceability. Symbol-anchored traceability applies the same move across engineering artifacts. Model elements, lints, code, proofs, and registries join through derived, re-checked edges. The resulting graph lets engineers query what a change affects and inspect why.

Related mechanisms: Control ↔ substrate dependency · Computed Control Blast Radius · Governance Graph · Symbol-anchored traceability · Derived Traceability.

Appendix C: Model Reference

Appendix C — Model Reference

Parts II–IV introduce only the model detail needed for the engineering argument. This appendix collects the principal schemas, invariants, derivation directions, and correspondence machinery for those models.

Each model begins with an engineering question and uses the smallest representation that makes the relevant property explicit. The forms here are reference patterns, not a required inventory.

The appendix serves as both a reference and a pattern book: the representations are specific enough to expose their engineering properties but general enough to adapt to other systems. Each section also separates what the model represents from the machinery that may later give selected declarations authority.

Each section answers the same four questions:

- **Engineering question** — what must the engineer be able to know?
- **Representation** — what entities and relations make that question answerable?
- **Property** — what can now be stated precisely?
- **Authority and correspondence** — which facts are authored, which are observed or derived, and how is disagreement detected?

Correspondence alone does not establish correctness. A descriptive model can accurately describe an undesirable system; a normative model supplies authored intent against which the implementation can be checked. Part III supplies the mechanisms that can give selected declarations authority.

Figure C-1 establishes the pattern specialized by the later figures.

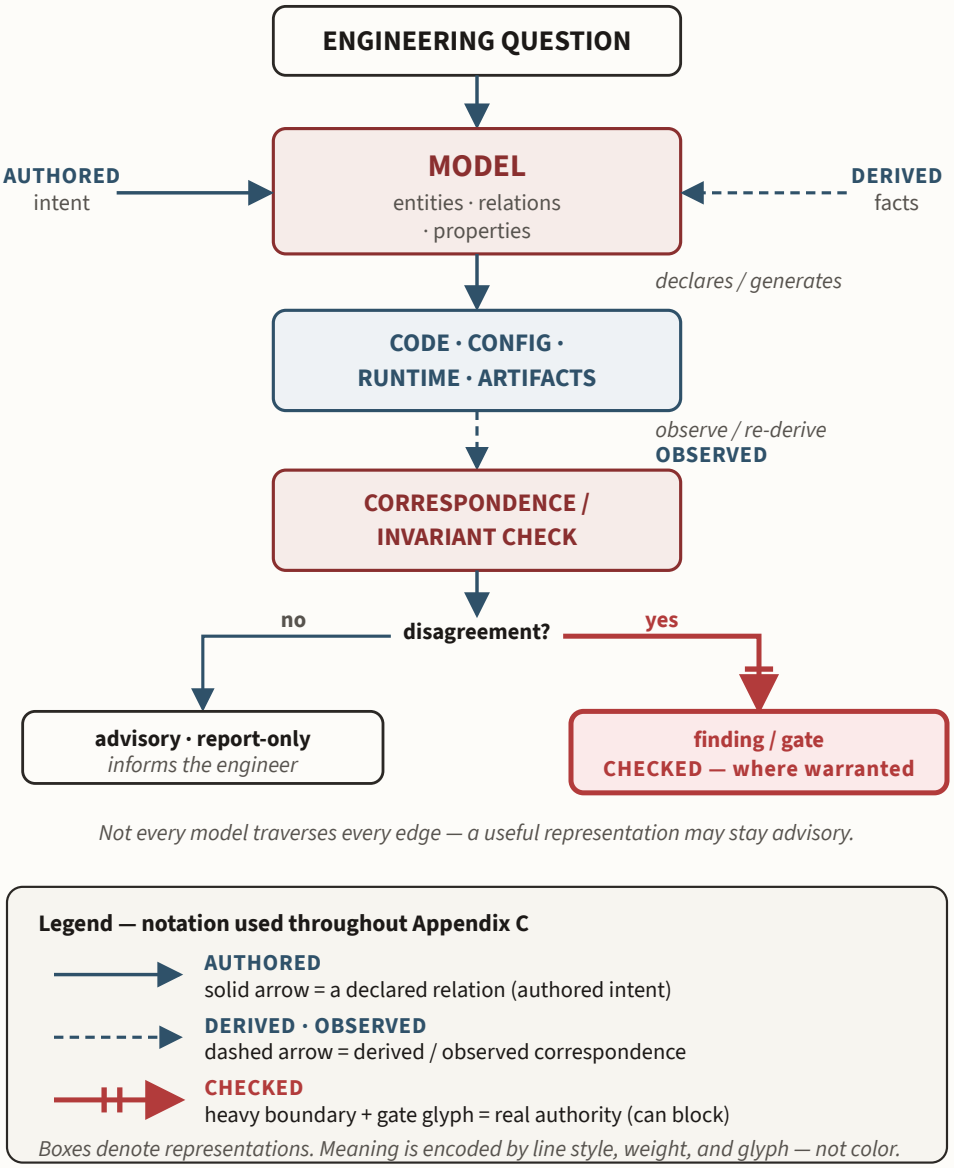


Figure C-1: The executable-model pattern. *An engineering question selects a representation containing authored, derived, or observed facts. Correspondence machinery compares represented facts with implementation or runtime evidence. Disagreement may remain advisory or, where the obligation has authority, feed a deterministic gate.*

C.1 Structure and Boundaries

Structural models represent what exists and how elements relate. In review, they expose which architectural boundaries a change affects. Ownership and boundary policy often share the same representation but answer distinct questions: who owns a surface, and which relationships are permitted.

C.1.1 Component and Zone

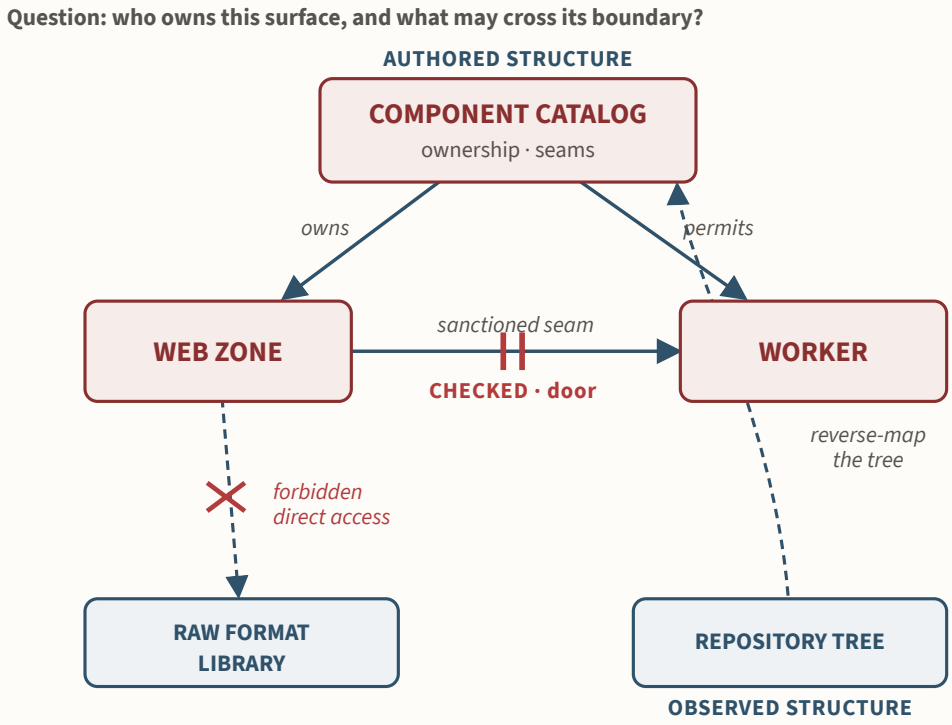
Engineering question. Where am I, who owns this surface, and what may cross its boundary?

The model maps implementation surfaces to components and declares the seams through which those components may interact.

Representation. A minimal form:

```
Component
    id
    name
    focus_directories[]
    tags[]
    boundary_kind
Zone
    component_id
    directory
Seam
    source_component
    target_surface
    relation_kind
    permitted
```

Figure C.1-1 joins the authored catalog to the observed tree.



Legend: see Figure C-1. Gate glyph = a guarded door (authority); X = a forbidden crossing with no door.

Figure C.1-1: Structure and boundaries. *Authored ownership and permitted seams are reconciled with the observed repository tree. Sanctioned crossings pass through declared doors; forbidden direct access raises a finding.*

Property. Representative properties, each with the source that decides it:

Table C.1-1: Representative properties of the component-and-zone model, and the fact source each draws on.

Property	Fact source
Every in-scope source directory belongs to exactly one component	Authored ownership joined to the observed tree
Every component's declared directory exists	Repository
Boundary crossings use sanctioned seams	Authored model
Powerful mutation surfaces are reachable only through declared doors	Authored model plus observed dependency

Directory existence can be derived from the repository; ownership and permitted boundaries are authored architectural decisions.

Authority and correspondence. The repository tree is mapped back to the component catalog. A directory with no owner, multiple owners, or an observed dependency crossing

an undeclared boundary becomes a correspondence finding. Where reconciliation depends on source locations, anchor references to resolvable symbols rather than line numbers so the correspondence can be re-derived after edits. The finding may remain advisory or feed a lint, build failure, or admission gate.

C.1.2 Service Flow

Engineering question. Which service may call or reach which service or resource, and under what policy?

Representation.

```
Service
    id
    role
Resource
    id
    kind
Flow
    source
    target
    relation
    auth_posture
```

The central object is the declared edge. Each permitted service-to-service or service-to-resource relation records both reachability and its authentication posture; an observed edge with no declaration is a mismatch.

Property. Only declared service-to-service and service-to-resource relationships are permitted, each with its declared authentication posture. The same graph carries both connectivity and access policy.

Authority and correspondence. The declared graph supplies the *ought*. Static call sites, generated configuration, deployment wiring, and runtime observation each supply part of the *is*. A correspondence check asks whether each observed edge is declared and, where it matters, whether each declared edge still exists.

The same representation pattern applies to firewall policy, service-mesh authorization, network policy, and cloud access control: the representation declares permitted relations while the enforcement mechanism varies by substrate.

C.1.3 Two Simpler Structural Forms

Two simpler forms use the same pattern: reconcile observed facts or edges against a declared set.

- **Domain registry.** A registry stores a slowly changing fact under one stable key. Consumers query or join through that key rather than maintaining independent copies.

- **Bill of materials.** A bill of materials records the third-party dependencies present in a build. Where an authored dependency set exists, completeness can be checked by reconciling the observed dependency edges against it.

C.2 Behavior and Ownership

Behavioral models represent state, transition, and time. Ownership models represent who may act on or hold a resource. The two are distinct but often joined: behavior constrains possible transitions; ownership constrains who may perform them.

C.2.1 Lifecycle

Engineering question. What states may this object occupy, and which transitions are legal?

Representation. A minimal job lifecycle moves an item from `FREE` to `LEASED` on claim, from `LEASED` to `DONE` on completion, and from `LEASED` back to `FREE` on expiry or release. `DONE` is terminal.

Property. The invariants sort into safety and liveness:

Table C.2-1: Representative lifecycle invariants, by kind.

Invariant	Kind
Only declared transitions occur	Safety
Terminal states do not re-enter processing	Safety
A claimed job eventually completes or returns to an available state, under stated assumptions	Liveness

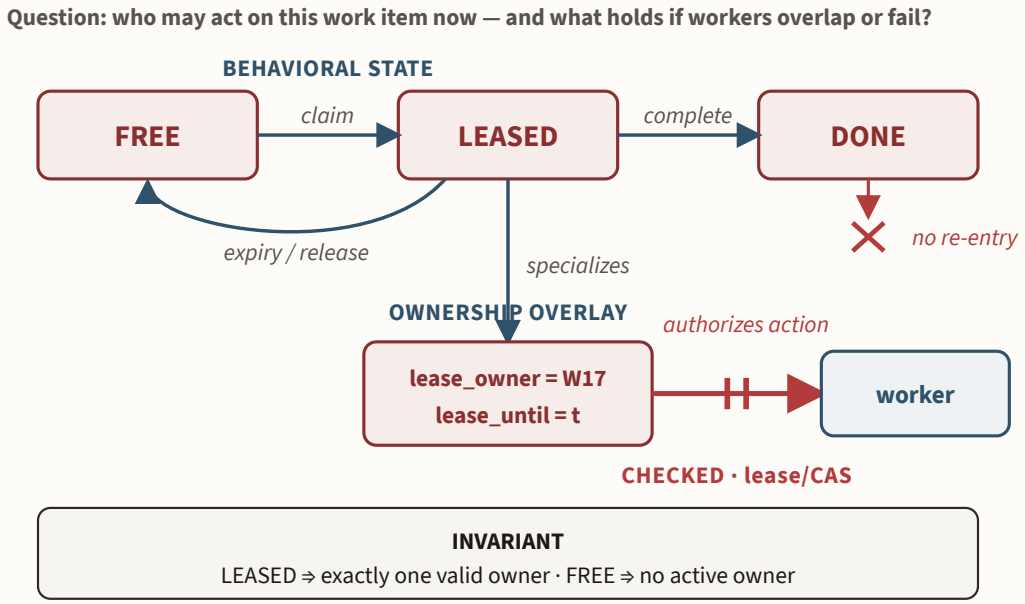
Authority and correspondence. Transition operations or checks can mediate actual state changes against the declared transition relation. Any implemented transition absent from that relation becomes a finding. Where two lifecycles interact, represent their cross-machine transitions explicitly rather than leaving those relationships implicit in code.

C.2.2 Ownership and Lease

Concurrency adds ownership and lease state to the lifecycle. The resulting model must represent not only which transitions are legal, but which actor may perform them now.

Engineering question. Who may act on this work item now, and what stays true if workers overlap, fail, or retry?

Figure C.2-1 lays the ownership overlay over the state machine.



Legend: see Figure C-1. Ownership is a working class in its own right, shown here joined to behavior.

Figure C.2-1: Behavior plus ownership. *The lifecycle FREE → LEASED → DONE is combined with a lease that records the current owner and expiry. The ownership invariant requires exactly one valid owner while LEASED and no active owner while FREE.*

Property. Representative invariants:

- **A leased item has a valid owner**, and a free item has none.
- **Ownership changes through the declared acquisition and release mechanism**, not by ad hoc mutation.
- **Terminal work cannot be reclaimed.**
- **Where ordering is required, observed acquisition order respects the declared order.**

Authority and correspondence. Some ownership facts are runtime facts and must be observed. The ownership protocol itself is authored intent. Atomic operations, leases, compare-and-swap, transition primitives, and admission checks can make selected invariants enforceable.

Concurrency limits and ownership answer different questions. A semaphore or mediator constrains how many actors may execute; single-writer ownership identifies which actor may mutate. They require distinct representations because they govern different properties.

C.3 Execution and Placement

Execution and placement require two related views: where components run and connect, and how a host schedules their work. Keeping those views separate lets review distinguish changes to topology from changes to execution policy.

Engineering question. Where does work run, and which scheduling decisions may change with the host it runs on?

Representation. Use separate views for topology and host policy.

- **Deployment topology** represents where components run and how they connect: web → queue → worker → storage.
- **Host execution policy** maps each host profile to a scheduling policy: elastic fan-out, bounded fan-out, or serialized execution.

Figure C.3-1 shows the stable graph with the policy as a separate layer beneath it.

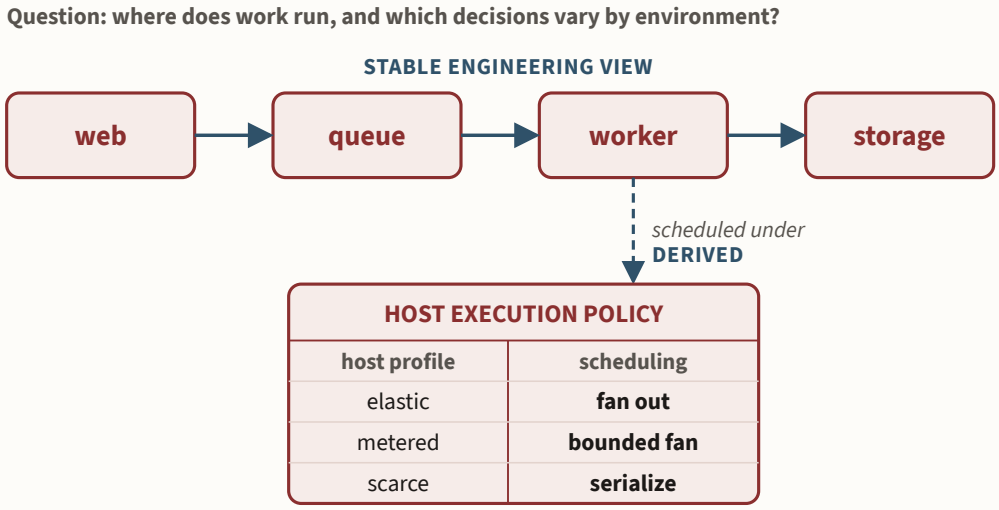


Figure C.3-1: Stable topology, variable execution policy. *Deployment topology remains stable while host profiles select different scheduling behavior.*

Property. A host's concurrency ceiling belongs to scheduling policy, not to the deployment graph.

Authority and correspondence. Deployment configuration reconciles against the topology; scheduler behavior reconciles against the host execution policy. Either finding may remain advisory or feed a deployment or scheduler gate.

C.4 Measurement

Measurement models represent quantitative properties of the running system and the bounds against which they are evaluated. In review, they expose what is measured, what bound applies, and what evidence supports that bound.

Engineering question. What quantity matters, how is it measured, what bound has been declared, and what should happen when the bound is exceeded?

Representation.

```
Measure
    id
    quantity
    unit
    scope
    observation_source
Bound
    measure_id
    threshold
    aggregation
Disposition
    bound_id
    mode        # report | warn | gate
```

The disposition stays separate from the measured bound; its mode records whether a bound is report-only, warning-producing, or admission-blocking. Measurability alone does not justify enforcement.

Figure C.4-1 separates the measurement from the authority that may or may not attach to it.

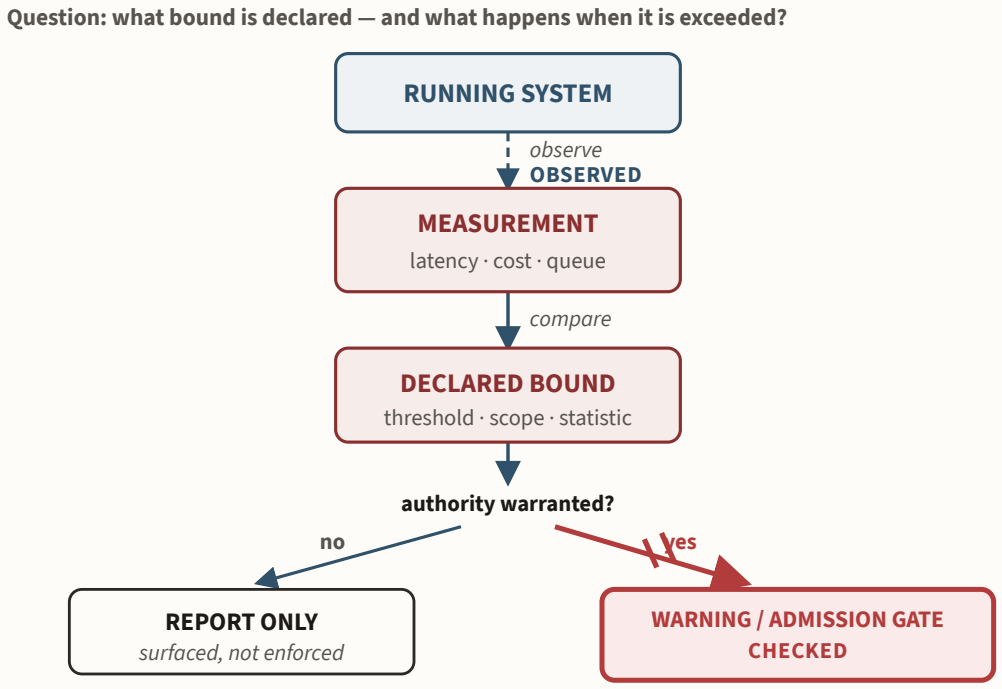


Figure C.4-1: Measurement does not imply authority. *A sensor produces an observed measurement, which is compared with a declared bound. The result remains report-only unless the evidence warrants warning or admission authority.*

Property. A measurement model can state that request latency remains below a declared bound for a defined request class, queue depth below a declared ceiling, or processing cost within a specified envelope. Scope, statistic, and aggregation window are part of the property; a threshold without them is underspecified.

Authority and correspondence. Sensors supply observations; the model supplies their scope, interpretation, and declared bounds. Measurements remain observations unless the evidence warrants warning or gate authority.

C.5 Documentation and Provenance

Provenance models record what happened. Documentation models represent the human-facing claims that should remain synchronized with those facts. In review, they expose whether required records exist and whether derived documentation still reflects its source.

Engineering question. What happened, and which human-facing account of it must remain true?

Representation.

```
Action
    id
    artifact
    operation
    actor
    mechanism
    timestamp
    result
DerivedDocumentation
    source_fact
    rendered_claim
```

Where documentation can be derived from structured facts, generate it from those facts rather than maintaining an independent copy.

Figure C.5-1 puts the fact before the prose.

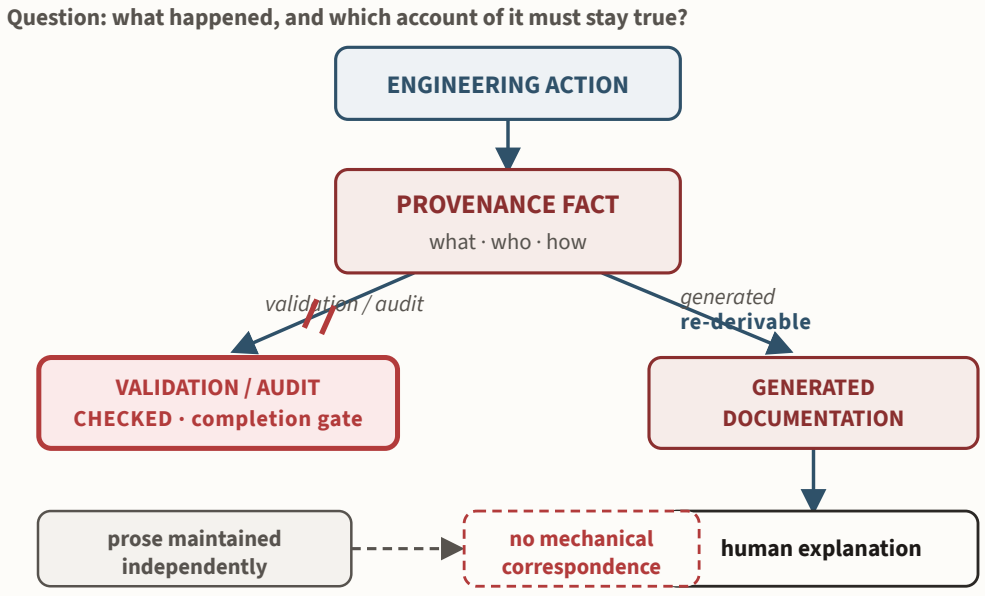


Figure C.5-1: Facts before prose. *A structured provenance record supports mechanical validation and re-derivable documentation; independently maintained prose lacks that correspondence.*

Property. Representative properties:

- **Every relevant mutation leaves a provenance record.**
- **Generated documentation reflects the current structured facts** from which it is derived.
- **References in generated documentation resolve to existing identities.**

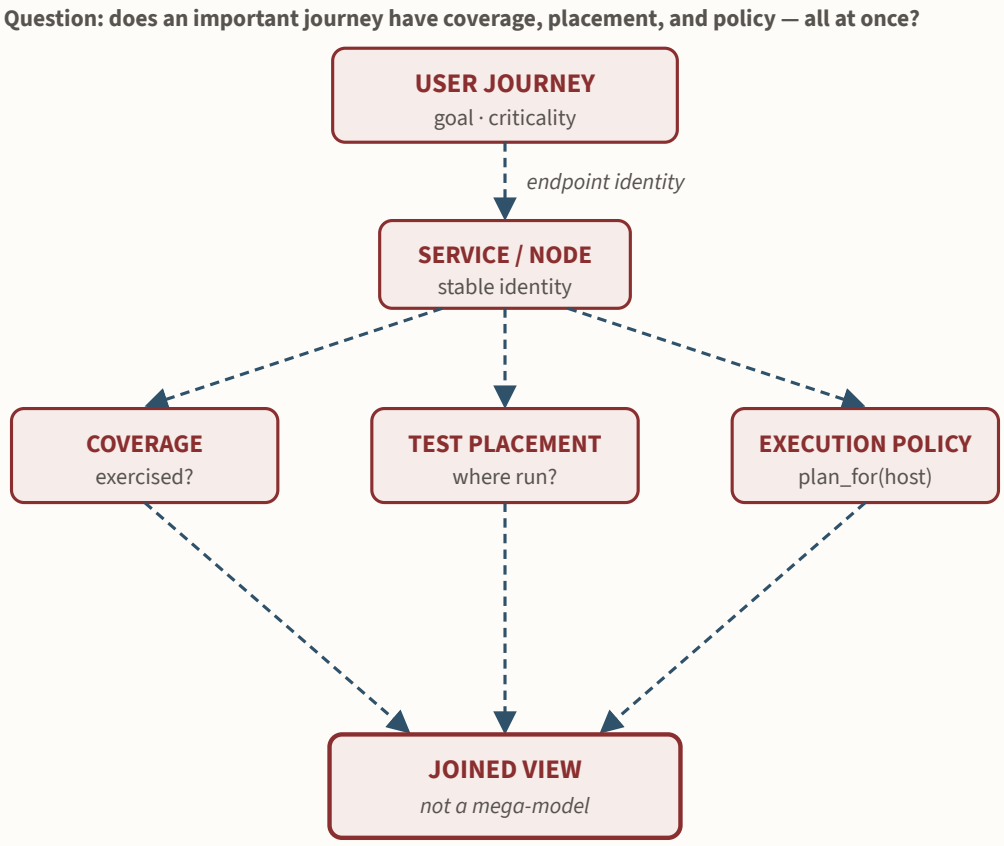
Authority and correspondence. Structured provenance can be validated mechanically, and generated documentation can be re-derived from its source facts. Free prose generally cannot be checked for semantic equivalence by the same machinery and therefore remains subject to human review.

C.6 Joining Views Around a Scenario

An engineering question may span several models without requiring one larger stored model. Keep question-specific representations separate and join them through stable identities when a cross-model property must be evaluated.

Engineering question. Does an important user journey exercise its declared endpoints, receive the required test coverage and placement, and execute under the correct host policy?

Figure C.6-1 traces the join.



Legend: see Figure C-1. Every join edge is dashed — derived correspondence on stable identities, no copied facts.

Figure C.6-1: Composition by reference. *Stable identities join the user-journey, service, coverage, test-placement, and execution-policy views. The resulting joined view supports cross-model queries without duplicating those facts into a single stored model.*

Property. The joined view states cross-model properties that no single view can:

- **Every declared journey step has the required test coverage.**
- **Declared journey dependencies agree with the actual call sites.**
- **Declared endpoints are exercised.**
- **Test placement reflects journey criticality.**
- **Execution respects the selected host policy.**

The reusable pattern. State the engineering question, select the relevant views, join them on stable identities, and evaluate the cross-view property. The scenario is a traversal across representations, not an additional stored model.

C.7 From Representation to Authority

Modeling and Alignment are distinct activities. A model may reduce reconstruction, expose dependencies, preserve intent, or support prediction while remaining advisory. Alignment begins when a represented obligation receives authority through a deterministic mechanism.

Table C.7-1 maps each representation to the obligation it exposes, a correspondence mechanism that can evaluate it, and the authority that may attach.

Table C.7-1: From representation to possible authority. Each row identifies a represented obligation, a correspondence mechanism that can evaluate it, and the authority that may attach.

Representation	Obligation exposed	Correspondence mechanism	Possible authoritative mechanism
Component / zone	Each surface has one owner; boundaries use sanctioned seams	Reverse-map the tree and dependency edges	Boundary lint or admission gate
Service flow	Only declared service and resource edges are legal	Call and configuration parity	Policy or build gate
Lifecycle	Only legal transitions occur	Transition-site reconciliation	Transition primitive or invariant check
Ownership	Active work has a valid owner	Runtime ownership checks	Lease, compare-and-swap, or transition mechanism
Execution	Placement and scheduling follow declared policy	Deployment and profile reconciliation	Deployment or scheduler gate
Measurement	Observed quantity stays within a declared bound	Sensor plus bound evaluation	Warning or admission gate
Provenance	Actions leave the required structured record	Record and artifact validation	Completion gate
Joined scenario	Cross-view obligations agree	Identity joins across views	Composed check or gate where decidable

The correspondence mechanisms in the third column compare represented intent with implementation or runtime evidence. Where the property warrants enforcement, the resulting finding can feed a deterministic gate. The handoff from Modeling to Alignment is:

```
      MODELING                          ALIGNMENT
make knowledge explicit   →   give selected obligations authority
```

```
represented obligation      →   authoritative mechanism
 "what is / ought?"             "must this hold?"
```

Models create surfaces on which Alignment can act. Alignment determines which represented obligations deserve authority, where the relevant evidence can be observed, and which deterministic mechanism should enforce them.

Appendix D: Operator's Reference

Appendix D — Operator's Reference

Operational reference for running a Governed Engineering Environment

This appendix is an operational reference for a Governed Engineering Environment. Use each card to answer a specific operating question, then return to the work.

The deck supports four jobs: orient, diagnose, act, and certify. The cards within each group answer a specific operating question. Operating Doctrine summarizes the rules that govern the rest.

These cards are operating aids, not universal benchmarks. Use a metric only when it helps evaluate or steer a specific engineering claim. DocAble values are reference observations, not targets. The governing question is whether the environment makes future work more capable, trustworthy, or economical than it costs to maintain.

D.1 The Operator's Dashboard

The dashboard asks whether autonomous work is producing durable, trustworthy progress. Read the outcomes first; use diagnostics to explain why they changed.

Track five primary readings: durable throughput, defect escape, human-attention burden, representation health, and engineering-capital return. The first three measure outcomes; the last two ask whether the environment can sustain them. Churn, model coverage, support ratio, control count, grammar coverage, and navigation cost are diagnostics, not objectives.

Table D.1-1: The Operator's Dashboard. Five primary readings, the question each answers, the desired direction of change, and useful diagnostics.

Reading	What to ask	Healthy direction	Useful diagnostics
Durable throughput	How much accepted work survives without reopening, rollback, or repeated repair?	More accepted capability without proportional growth in intervention	closure rate; reopen rate; change cadence; churn
Defect escape	What incorrect or policy-violating work crosses the boundary that was supposed to catch it?	Falls for governed obligations	validator findings; escaped defects; rollback/incident rate
Human-attention burden	How much repeated reconstruction, review, or adjudication does each unit of durable work require?	Falls where knowledge or judgment can be made durable; remains where human judgment is deliberate	interventions per landed change; review time; recurring decisions
Representation health	Can the representations being relied upon still answer the questions they claim to answer?	Claimed correspondence holds; stale or uncovered surfaces remain explicit	drift checks; traceability; freshness; coverage/relevance
Engineering-capital return	Does accumulated structure make later work more capable or cheaper than the structure costs to carry?	Reuse and inherited capability rise while carrying cost remains justified	mechanism reuse; recurring-class disappearance; maintenance burden; retirement candidates

DocAble's support ratio, control count, Missing-Model drain, navigation pilot, and churn curves are observations from one build, not operating targets. Appendix H records the underlying measurements and their limitations.

Treat uncalibrated visual indicators as status, not quantitative measurements.

D.2 Daily Operator Review

Use these five questions at a regular cadence and after consequential changes. The card introduces no new measures; each question points to the card that helps answer it.

1. **Did durable work move forward?** → Dashboard and System Health. Check accepted capability, reopenings, rollbacks, and unexpected churn.
2. **Did any trusted representation become doubtful?** → Representation Health. Check the correspondence claims that current work actually depends on.
3. **Did an obligation escape the mechanism intended to enforce it?** → System Health and Release Readiness. Identify the boundary, evidence, validator, or gate that failed, or confirm that the obligation was intentionally left to judgment.
4. **What did a human have to reconstruct or decide again?** → Human Judgment. Repeated semantic work may be worth making explicit or durable, but recurrence alone does not justify automation.
5. **What should the environment learn from the recurrence?** → Governance Conversion and Engineering Capital. Decide whether the lesson justifies a representation, control, procedure, architectural change, retirement of stale machinery, or no durable change.

```
DAILY OPERATOR REVIEW                              "The morning five questions"
──────────────────────────────────────────────────────────────────────────────
1  Did durable work move forward?                → Dashboard / System Health
2  Did a trusted representation become doubtful? → Representation Health
3  Did an obligation escape enforcement?          → System Health / Release Readiness
4  What did a human have to decide again?         → Human Judgment
5  What should the environment learn?             → Governance Conversion / Capital
──────────────────────────────────────────────────────────────────────────────
```

These reviews may run automatically at lifecycle boundaries, but classifying and interpreting the results may still require human judgment.

D.3 System Health

Is the environment operating correctly right now?

Begin with outcomes. Check whether accepted work is landing, governed obligations are holding, validators and gates are checking what they are supposed to check, and human intervention or rollback has risen unexpectedly. Use churn, coverage, drift, and other local measures only to diagnose changes in those outcomes.

Three readings

- **Durable throughput.** Accepted work remains accepted without reopening, rollback, or repeated repair.
- **Defect and policy escape.** Incorrect or policy-violating work passes the boundary intended to prevent it.
- **Intervention burden.** Humans get pulled back into repeated reconstruction, review, or emergency repair.

Use churn, validator coverage, correspondence failures, and queue pressure only when they explain movement in the primary readings.

```
SYSTEM HEALTH               "Is the environment healthy right now?"
──────────────────────────────────────────────────────────────────
Durable throughput       accepted work survives
Defect / policy escape   governed obligations hold
Human intervention       repeated attention is bounded
──────────────────────────────────────────────────────────────────
Diagnose with:
churn · correspondence · validator reach · resource pressure
```

Read churn as motion, not health

Churn measures code motion, not system health. A migration or architectural rewrite can produce high churn in a healthy environment, while a quiet repository can still contain stale representations or repeated human adjudication. Appendix H records the churn observed in DocAble.

D.4 Representation Health

Can I trust this representation for the question I am asking?

Trust begins by stating the correspondence a representation claims. A derived service map may describe the implementation at HEAD; an ownership model may encode intent the implementation must satisfy; a generated policy may be the authoritative source from which implementation artifacts derive. These are distinct correspondence relations.

Read four things together:

- **Correspondence.** Does the claimed relation still hold? Descriptive models should still match observed reality; intent-bearing models should still be satisfied where authority applies; generated artifacts should still derive from their declared source.
- **Coverage and relevance.** Does the representation cover the engineering surface for which it is being relied upon? Missing coverage may be legitimate when implementation lies below the model's intended grain.
- **Traceability.** Can the reader move from the model claim to the territory it concerns, and back where that reverse relation is meaningful?
- **Freshness.** Has the claimed correspondence been re-established since the territory or the authored intent last changed?

```
REPRESENTATION HEALTH     "Can I trust this representation for this question?"
──────────────────────────────────────────────────────────────────────────────
Correspondence   claimed relation still holds
Coverage         intended surface is represented
Traceability     claims resolve to their subjects
Freshness        relation re-established after change
──────────────────────────────────────────────────────────────────────────────
Ask first: what correspondence does this model claim?
```

If the claimed correspondence fails, do not rely on the representation for that reasoning. The finding does not determine whether the representation or the represented system should change.

In DocAble, the Missing-Model surface fell from 56% to 7.89% over nine passes. Appendix H records the measurement and its limitations.

D.5 Human Judgment

Where is human attention going, and what do we keep deciding again?

Human attention is scarce, but some work should remain human. Novel architecture, ambiguous evidence, and consequential review may require human judgment. Look instead for knowledge people repeatedly reconstruct or decisions they repeatedly make that could economically become durable.

Ask two questions:

- **Where is attention being spent?** Distinguish creation of new intent, evidence interpretation, reconstruction of known context, routine review, and exception handling.
- **Which judgments recur in substantially the same form?** Ask whether a model, alignment mechanism, procedure, or other durable structure could keep people from making the same decision again.

```
HUMAN JUDGMENT                          "Where is scarce attention going?"
──────────────────────────────────────────────────────────────────────────
Novel intent / architecture     expected human work
Semantic validation             often intentional
Repeated reconstruction         investigate externalization
Routine adjudication            investigate authority
Recurring emergency             diagnose the class
──────────────────────────────────────────────────────────────────────────
Repeated ≠ automate automatically.
Ask whether making it durable will repay its carrying cost.
```

The goal is less avoidable repeated reasoning per unit of durable output, while people remain responsible for work whose novelty, ambiguity, consequence, or poor governability still requires judgment.

D.6 Engineering Capital

Is accumulated engineering structure producing value for later work?

Engineering structure becomes capital when later work inherits productive capacity from it. Models, validators, procedures, generators, and gates become capital when the value they provide to later work exceeds the cost to build, run, understand, reconcile, and maintain them.

Read capital through five questions:

- **Return.** What future work became cheaper, safer, faster, or possible because this structure exists?
- **Inheritance.** Do later engineers and agents reuse the capability, or reconstruct an equivalent solution?
- **Carrying cost.** What does the structure cost to run, understand, reconcile, and maintain?
- **Depreciation.** Has the system, obligation, model substrate, or surrounding workflow moved far enough that the structure's return is falling?
- **Retirement.** Which structure now costs more to carry than the future value it preserves?

```
ENGINEERING CAPITAL          "Is accumulated structure still earning its keep?"
──────────────────────────────────────────────────────────────────────────────
RETURN        later work gains capability
INHERITANCE   value is reused, not reconstructed
CARRYING      upkeep remains proportionate
DEPRECIATION  fit and return are watched
RETIREMENT    stale capital is removed
──────────────────────────────────────────────────────────────────────────────
Capital = productive structure, not apparatus quantity.
```

Support ratio and control count measure accumulated apparatus, not capital return. Evaluate that apparatus by the future capability it enables net of carrying cost. Appendix H records DocAble's observed support and control growth.

One way to test for engineering capital is to hold the task and reasoner approximately fixed while changing the environment. If the same intelligence can answer larger questions, reconstruct less context, require less intervention, or produce more durable work in one environment than another, the difference is evidence that the environment stores productive capacity.

D.7 Governance Conversion

What should the environment learn from this recurrence?

When the same kind of failure recurs, ask whether the environment should change. First classify what keeps going wrong: missing knowledge, a weak representation, an implicit obligation, late evidence, misplaced authority, a failed mechanism, an architectural weakness, or an obsolete control. Then decide whether the lesson is worth making durable.

Engineering Capital asks whether existing structure still earns its keep; Governance Conversion asks whether a recurring lesson should change that structure.

The conversion decision has three stages:

- **Classify the gap.** Identify the recurring problem rather than its latest symptom: missing knowledge, representation, obligation, evidence, authority, mechanism, architecture, or an obsolete control.
- **Choose the durable response.** Possible responses include a representation, constraint, sensor, validator, gate, procedure, architectural change, retirement of stale machinery, or deliberate retention of human judgment.
- **Price the investment.** Build the response only when expected recurrence cost and consequence justify both construction and carrying cost.

```
GOVERNANCE CONVERSION
"What should the environment learn?"
Recurring event
      ↓
CLASSIFY THE GAP
knowledge · representation · obligation · evidence
authority · mechanism · architecture · obsolete control
      ↓
CHOOSE THE RESPONSE
model · constraint · sensor · validator · gate
procedure · redesign · retire · keep human
      ↓
PRICE THE INVESTMENT
future value > construction + carrying cost ?
```

The goal is fewer costly recurrences and durable responses that repay their carrying cost. Cheap, rare failures may remain unconverted.

See Chapter 3.4 for the full treatment.

D.8 Brownfield Migration Drill

Most organizations already possess fragments of a Governed Engineering Environment: documentation, tests, review rules, CI checks, ownership files, runbooks, deployment policy, and expert knowledge. Brownfield migration asks three questions: Which engineering knowledge needs stronger representation? Which obligations already have effective authority? Where do gaps still force people to reconstruct what the environment should know?

When a wiki carries important system knowledge, link its claims to the implementation and evidence they concern. Check mechanically decidable claims automatically. Give stronger structure only to knowledge whose future use justifies the cost. Chapter 4.2 gives the full treatment.

Joining the wiki to the code

Four additions can make a wiki a more trustworthy engineering entry point. Add them only where future use justifies the added maintenance burden.

Table D.8-1: Joining wiki knowledge to implementation. Four additions and the behavior each must provide.

Element	Required behavior
Bidirectional links	Wiki pages name the code and tests that back their claims; code and tests name the pages that explain their purpose. This bidirectional relation provides traceability between claims and the implementation or tests that support them.
Agent briefs	Tell agents how to locate, traverse, cite, and maintain the relevant wiki material.
Design templates	Name the wiki pages the work touches, and record when a missing page has to be created, before implementation begins.
Definition-of-Done checks	Before work is accepted, compare affected pages with the current repository rather than relying on the agent's report that they were updated.

Add metadata only when a defined consumer uses it. Tags, backlinks, ownership fields, and machine-readable mirrors should feed an audit, brief, retrieval step, analysis, or gate; otherwise they add another surface that can drift.

The migration path

The migration has four stages—Audit, Synchronize, Govern, and Extend—and each has an exit criterion. Advance when the criterion is met, not when enough time has passed.

Table D.8-2: The brownfield migration path. Each stage names its work and exit criterion.

Stage	Work	Exit criterion
Audit	Inventory important engineering knowledge, implementation surfaces, and ex-	Important gaps are classified rather than merely counted.

Stage	Work	Exit criterion
	isting controls. Classify which claims have evidence or authority, which rely on convention, which disagree with the system, and which implementation regions legitimately fall below the grain of any useful explicit model. Keep this stage advisory.	
Synchronize	Repair high-value correspondence during ordinary work. When an agent or engineer touches a linked surface, re-check nearby claims, repair stale anchors, and surface semantic disagreements explicitly.	The representations you intend to trust have explicit, credible correspondence to their subjects.
Govern	Give selected obligations authority at the earliest boundary where the required property is decidable. Some obligations can be checked directly over artifacts or actions; others depend on the representations strengthened in earlier stages.	The high-value obligations selected for governance no longer depend primarily on someone remembering to inspect them.
Extend	Add richer representations, broader evidence, stronger mechanization, or additional authority only where expected return exceeds carrying cost.	Every addition has a named consumer or engineering question and a stated reason to exist.

The promotion rule

Audit → reconcile → trust → govern → extend selectively.

Do not give an inaccurate representation authority. Do not require an explicit model where a local control already settles the obligation economically. Do not add structured metadata without a defined consumer. Add structure only when it is worth the cost: it should reduce repeated reconstruction or bring an important obligation under control.

Orphan triage

Classify each unlinked region into one of three states so coverage reflects the intended model grain rather than raw implementation count.

1. **Missing representation** — a real subsystem with no explaining page or model; write or propose one.
2. **Missing anchor** — the representation exists, but the link back from the code is absent; add the join.
3. **Below the grain** — implementation too fine-grained or generic to warrant its own representation, such as typed-ID aliases, configuration loaders, or enums.

See Chapter 4.2 for top-down and bottom-up model construction, lint-and-cover preparation, the Missing-Model Metric, and cost-based governance sizing.

D.9 Brownfield Progress

Where is the next high-value gap?

Brownfield progress has two independent dimensions. **Representation reach** asks how much of the important system can be understood through trustworthy explicit models. **Authority reach** asks how much of the important obligation surface is constrained, sensed, validated, or gated at an appropriate boundary. Neither is a maturity score that must reach 100 percent.

Strong local authority can exist with thin explicit modeling when tests, permissions, sandboxes, or admission controls directly observe the relevant property. Rich models can also coexist with weak authority when violations still depend on human detection. Address whichever uncovered gap matters most.

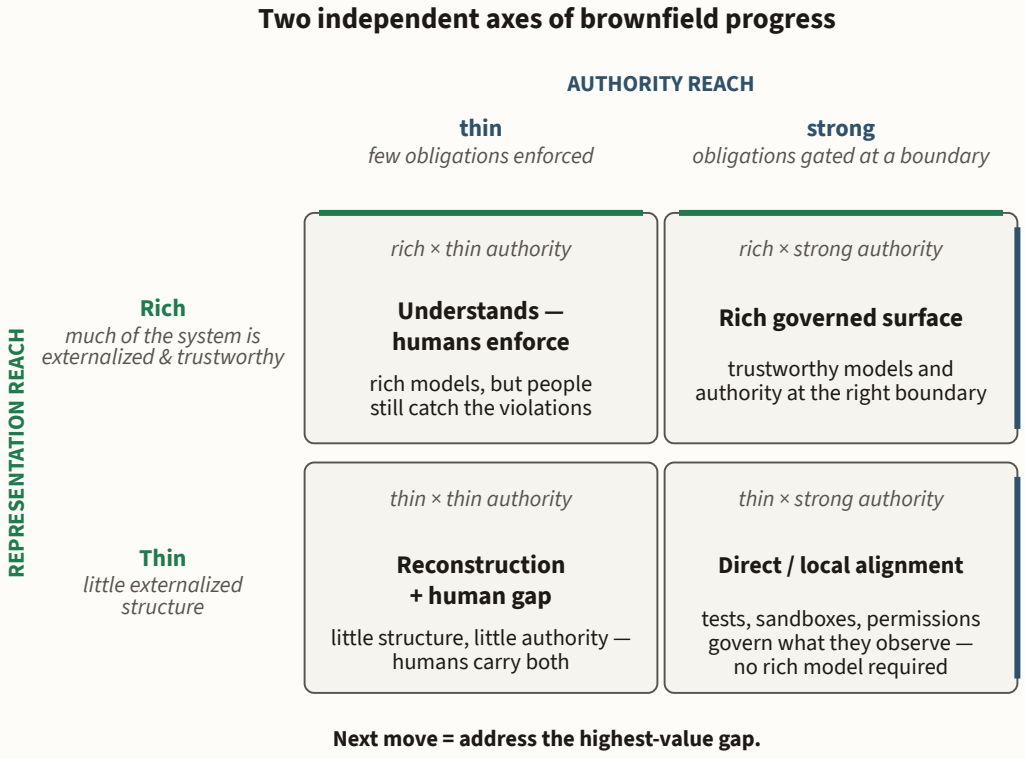


Figure D.9-1: Brownfield progress. *Representation reach and authority reach are independent axes. A system may be strong on either dimension without being strong on the other; the next investment should address the highest-value gap rather than maximize both indiscriminately.*

Apply Audit → Synchronize → Govern → Extend to a selected surface rather than to the organization as a whole. Appendix H records the observed DocAble measurements.

See Chapter 4.2 for the full treatment.

D.10 Release Readiness

Can I trust this release?

A release is ready when every obligation assigned blocking authority has current supporting evidence and every blocking gate passes. The checklist gathers those obligations at release time; it does not create new ones.

The specific checks depend on the system. A model-based obligation requires current correspondence only when the release relies on that model; a locally enforced property may require no system model. The general rule is obligation → evidence → authority.

The preflight

Instantiate the checklist for each release. List the obligations carrying blocking authority, confirm that each has current evidence, and confirm that every blocking gate passes.

```
RELEASE READINESS  ·  PREFLIGHT              "Can I trust this release?"
═══════════════════════════════════════════════════════════════════════
[ ]  Blocking obligations identified and current
[ ]  Required evidence present and fresh
[ ]  Trusted representations satisfy their correspondence contracts
     — where this release relies on them
[ ]  Validators / assurance checks exercised and green
[ ]  Admission / deployment gates green
[ ]  Overrides and accepted residual risks reviewed
═══════════════════════════════════════════════════════════════════════
All blocking obligations discharged  →  ship.
Otherwise  →  wait, record an explicit override, or revise the governing policy.
```

The override note

Treat an override as an engineering event. Record who accepted it, the available evidence, the bypassed obligation, and whether the exception reveals a recurring class that warrants later review. Do not carry the exception into future releases implicitly.

D.11 Evidence Quality

What justifies this claim?

Start with the claim and identify the evidence capable of supporting it. Different properties require different evidence: a structural invariant may be checked exhaustively, a performance claim may require measurement, a semantic-fidelity claim may require generative evaluation and human judgment, and a formal proof establishes only the property encoded under its stated model and assumptions.

For each claim, ask four questions:

- **Claim.** What exactly is being asserted, and over what scope?
- **Evidence.** What observation, measurement, proof, test, or other artifact bears on that claim?
- **Evaluation.** What procedure interprets the evidence, and what can that procedure miss?
- **Authority.** What consequence should the result have? Evidence may inform a human, produce a warning, or justify a blocking gate; evidentiary strength and enforcement authority remain separate decisions.

```
EVIDENCE QUALITY                              "What justifies this claim?"
──────────────────────────────────────────────────────────────────────────
CLAIM
  ↓  what would settle it?
EVIDENCE
  ↓  how is it interpreted?
EVALUATION
  ↓  what does the result deserve?
AUTHORITY
  advisory · warning · validator result · gate · human decision
──────────────────────────────────────────────────────────────────────────
No backing      →  unsupported claim
Weak backing    →  state the limit
Strong backing  →  still choose authority deliberately
```

The critical failure is claiming more confidence than the evidence supports when the claim matters. Preserve evidence provenance, state limitations explicitly, and distinguish what the evidence establishes from the authority the organization chooses to attach.

D.12 Operating Doctrine

Four rules govern operation of a MAGE environment.

- **Model to answer a question.** Externalize the knowledge and intent needed for humans and agents to reason at the required scale. Stop when additional representation costs more to maintain than the reasoning it saves.
- **Put authority where the obligation becomes decidable.** Use the earliest boundary with sufficient semantics and evidence to decide the obligation. Local authority may act directly on actions or artifacts; richer representations can make broader properties decidable.
- **Make useful judgment durable.** When a costly lesson recurs, decide whether future work should inherit a representation, control, procedure, architectural change, or other durable structure. Convert selectively.
- **Maintain the portfolio.** Keep required truths aligned with reality, watch carrying cost, repair depreciated structure, and retire machinery that no longer earns its keep.

The objective is a Governed Engineering Environment in which commodity intelligence can perform more useful work without requiring engineering judgment to be reconstructed repeatedly.

```
OPERATING DOCTRINE                                    "How do I run MAGE?"
──────────────────────────────────────────────────────────────────────────
MODEL
Answer the engineering question at the right abstraction.
ALIGN
Give selected obligations authority where they become decidable.
CONVERT
Make recurring, valuable judgment durable.
MAINTAIN
Measure return, reconcile drift, retire depreciation.
──────────────────────────────────────────────────────────────────────────
      Governed Engineering Environment
              ↓
      trustworthy autonomy
              ↓
      durable engineering throughput
```

Modeling and Alignment are complementary, not sequential. Modeling makes more properties available to reasoning; Alignment gives selected obligations authority, either through those models or directly over actions and artifacts. Governance Conversion makes recurring lessons durable. When later work benefits from that structure, it has become Engineering Capital. Operating the environment means keeping both its engineering claims and its economics sound.

Appendix E: How to Write a Skill

Appendix E — How to Write a Skill

Modeling a domain for an agent

Skills come in two broad kinds. A *tool-skill* gives an agent access to a capability: a CLI, API, MCP server, or other tool. A *mastery-skill* gives an agent a way to reason about a domain: the abstractions, distinctions, and engineering judgment an experienced engineer would bring to the work.

This appendix focuses on mastery-skills.

Tool-skill guidance is necessarily more provisional. The mechanisms by which agents discover and invoke tools continue to evolve — from command-line wrappers and vendor-specific tool interfaces to MCP and whatever follows them. Current platform documentation is therefore the right source for the mechanics of packaging a tool for an agent. The durable problem is not how today's agent calls a tool, but how to represent what an engineer knows so an agent can reason through it.

Much engineering knowledge begins in an inconvenient form: an experienced engineer knows how to do something. The knowledge may be scattered across conventions, examples, repeated instructions, documentation, and judgment acquired through practice. Giving an agent more instructions does not necessarily give it a model of the domain.

Writing a mastery-skill makes that knowledge explicit. Identify the domain's fundamental model, separate it into orthogonal facets, and connect them with a governing principle. The result is a model an agent can load and reason through. This is Modeling in miniature: make engineering knowledge explicit so the agent does not have to reconstruct it.

A mastery-skill remains a soft mechanism. It can teach an agent how to reason and guide the choices it makes; it cannot guarantee that the agent follows that guidance. Where a property must hold, encode it in Alignment: constraints, types, validators, gates, or other mechanisms that do not depend on agent cooperation.

The three mastery-skills introduced earlier in the book provide the worked examples: self-communicate models how the fleet communicates, self-governance how it applies the engineering method to its own work, and self-operate how it runs its operational lifecycles.

Despite their different domains, all three were constructed in the same way: find the domain's fundamental model, layer independent facets onto it, and tie them together with a governing principle.

Chapter 1 gives the construction method and quality bar; Chapter 2 applies them to the three skills. By the end, you should be able to recognize when a mastery-skill is appropriate, extract a model from recurring engineering judgment, and package that model so an agent can reason through it.

Appendix E - 1. Theory

A good mastery-skill has a recognizable structure. This chapter gives a three-step method for building one and the failure modes to test before shipping.

E.1.1 Anatomy

A skill is a directory with one required file and optional bundled resources. `SKILL.md` carries YAML frontmatter containing a name and a description, followed by the instructions the agent loads when the skill triggers. **Bundled resources** hold references, examples, scripts, and other material that the agent reads only when needed.

This structure supports progressive disclosure: the name and description make the skill discoverable, `SKILL.md` carries its governing structure, and bundled resources supply detail on demand. Put triggering information in the description, the governing model in `SKILL.md`, and bulk reference material in resources.

> **Implementation note (August 2026).** *Anthropic's current skill-authoring guidance recommends keeping loaded context concise, putting triggering information in the description, keeping* `SKILL.md` *under roughly 500 lines, moving invocation-specific detail into linked reference files, matching instruction specificity to task fragility, and evaluating skills on representative tasks. These mechanics will change; consult current platform documentation when implementing a skill. This appendix focuses on what knowledge the skill should contain and how to structure it.* (Source: Anthropic, "Skill authoring best practices.")

A **tool-skill** packages a capability the agent *invokes*; a **mastery-skill** packages judgment the agent reasons *through*. Tool-skills need little beyond reliable invocation — scope each to one capability, make its triggering conditions concrete, specify fragile operations precisely, and prefer deterministic scripts where generated procedures would drift. When a mastery-skill needs a tool interface, factor that interface into its own tool-skill and reference it.

E.1.2 From Domain Knowledge to Model

Build a mastery-skill in three steps.

- **Step 1 — Find the domain's fundamental model.** Name the model that makes the rest of the domain intelligible: the frame you would teach first to someone learning it. Do this before writing the resources. If you cannot state the fundamental model clearly, you are likely to produce a collection of tips rather than a coherent way of reasoning.
- **Step 2 — Layer orthogonal models onto it.** Decompose the domain into independent facets and represent each separately. Two facets that substantially overlap probably belong together; an important concern that fits nowhere indicates a gap in the decomposition. This separation also supports progressive disclosure — the agent can load a facet only when the task requires it.

- **Step 3 — Write the governing principle in `SKILL.md`.** The top-level file should not merely enumerate the resources. It should explain how the facets fit together and when to use each one. `SKILL.md` should provide enough structure to reason with the skill; the resources supply the detail required for particular tasks.

The structure supports composition and incremental adoption. Another skill can refer to the underlying model rather than duplicate it, and the fundamental model can be useful before every facet exists.

Orthogonal skills can also compose. Give each skill one reason to change, then use explicit interfaces to let one consume models, mechanisms, or observations another produces.

E.1.3 Failure Modes

A mastery-skill can fail in several predictable ways.

- **The description is vague.** The skill exists but does not trigger when needed. Put concrete task cues, formats, tools, or other recognizable triggers in the description.
- `SKILL.md` **becomes a manual.** Exhaustive reference material consumes context and obscures the governing model. Keep the top-level file focused and move detail into resources.
- **There is no fundamental model.** The skill becomes a collection of locally useful tips with no coherent way to reason across them. Return to Step 1.
- **The facets overlap.** Multiple resources partially encode the same concern, forcing the agent to reconcile them. Merge them or redraw the boundary along an independent axis.
- **Soft guidance is presented as hard enforcement.** A skill can guide an agent toward a behavior; it cannot guarantee that behavior. If a rule must hold independently of agent cooperation, implement a hard mechanism such as a lint, gate, type, or architectural constraint. The skill can explain and invoke that mechanism, but it should not claim to replace it.
- **The skill costs more to maintain than it saves.** Skills and hooks themselves require maintenance. Add skills and hooks only when recurrence or consequence justifies their upkeep.

A skill that never loads has no effect. Where a recurring event should reliably invoke it, pair the skill with an appropriate trigger or hook. The hook makes invocation more reliable; the skill still supplies guidance rather than enforcement.

Before shipping:

- [ ] The skill has been classified as a tool-skill or a mastery-skill.
- [ ] Its description names concrete triggering conditions and follows the platform's current discovery requirements.
- [ ] The top-level file contains the governing structure rather than bulk reference material.
- [ ] Detailed resources are loaded progressively.
- [ ] Instruction specificity matches task fragility.
- [ ] Deterministic, repeated operations are implemented deterministically where practical.

- [ ] For a mastery-skill, the fundamental model, orthogonal facets, and governing principle are explicit.
- [ ] Recurring invocation is supported by a trigger or hook where appropriate.
- [ ] Nothing is described as enforced when the skill can only guide.
- [ ] The skill has been evaluated on representative past tasks and on the models on which it will run.

Appendix E - 2. Applying the Recipe

This chapter applies the recipe to self-communicate, self-governance, and self-operate. Self-governance is the **recursive case**: it models the MAGE method itself. Self-operate tests the method against a less obviously modelable domain: operations.

E.2.1 self-communicate

Problem

The fleet and its operator produce prose and diagrams constantly — control descriptions, design docs, mechanism entries, runbooks, handoffs, and the orchestrator's own status reports and tradeoff explanations to the human. Ungoverned, it drifts: inconsistent terminology, inappropriate document structure, LLM tells, and poor figures. The problem is not generating prose but producing it consistently against an engineering standard. The skill packages the craft needed to do that.

Fundamental Model

Rhetoric as craft. Treat technical writing as a set of named techniques rather than taste alone. Document form, lexicon, voice, audit, and visualization then become independent facets of that model.

Orthogonal Models

One file per facet.

- **Rhetoric** (`writing/rhetoric.md`) — the device toolkit; sentence shape.
- **Document form** (`writing/engineering.md`) — the document's Diátaxis mode: tutorial, how-to, reference, or explanation.
- **Lexicon** (`writing/lexicon.md`) — one concept, one word.
- **Voice** (`writing/voice.md`) — the register and sound.
- **Audit** (`writing/audit.md`) — the grading procedure.
- **Visualization** (`drawing/diagrams.md`, Mermaid-first, with `charts.md` and `tables.md` for data figures) — the shapes prose carries poorly.

Each answers a different question about the document: form chooses its structure, lexicon its terms, voice its register, rhetoric its sentence-level technique, visualization what should be drawn, and audit how the result is checked.

Governing Principle

The governing principle is that representation should serve rather than distract from the idea. Apply the facets in order: choose the document form, draft in the house voice using varied rhetorical devices, use canonical terms from the lexicon, draw relationships that prose carries poorly, and audit before shipping. One additional rule applies throughout: name the concept once, then use the name.

Layout

```
self-communicate/
  SKILL.md
  writing/   rhetoric.md  engineering.md  lexicon.md  voice.md  audit.md
  drawing/   diagrams.md  charts.md  tables.md  svg-audit.py
```

One directory per concern, one file per facet — the tree *is* the orthogonal-model set.

Lesson

Rhetoric-as-craft turns a collection of style rules into a model. Separate facets can then load independently and be reused by other skills. The skill remains soft; where an audit result must block delivery, the audit belongs in a gate.

E.2.2 self-governance

Problem

An agent can have excellent tools and still engineer badly. It runs the tests, calls the formatter, queries the repository — and still models the wrong thing, gives authority to a fact that should stay advisory, or freezes a one-off mistake into a permanent rule. The missing capability is not another tool. It is the engineering method itself: knowing *what to model, what must hold, and which judgment is worth making durable.*

Self-governance packages that method as a skill. It teaches the agent to recognize the situation, choose a Modeling or Alignment move, act through the governed environment, and decide what should persist. The skill carries the method, not the codebase's facts. Concrete system knowledge—what the codebase intends, contains, and guarantees—must come from model providers rather than the skill's memory.

Fundamental Model

The MAGE engineering loop. Everything in the skill supports one cycle, run whenever the agent engineers:

- **Recognize.** What situation am I facing?
- **Model or align.** What must be understood? What must hold?
- **Choose the move and capability.** What engineering move fits, and which skill performs it?
- **Act through the GEE.** The skill proposes; the environment supplies consequence.
- **Learn.** What did the evidence show, and should any lesson become durable?

Self-governance is not a third MAGE activity; it applies Modeling and Alignment to the agent's own engineering work.

Orthogonal Models

The loop's questions are independent, so each becomes its own facet. Each of six directories answers one question; the router identifies the question and directs the agent to the corresponding facet.

- `modeling/` — Which representation would help, and how should it be improved? Carries the six model families, modeling moves, and a repertoire of established modeling techniques and their tradeoffs.
- `practice/` — What situation is this, and how much intervention is warranted?
- `system/` — What is true of this system? Retrieves concrete models through a provider contract and records their epistemic status.
- `skills/` — Which available capability performs the chosen move?
- `alignment/` — What should remain true without repeated human checking? Carries the mechanism repertoire and census.
- `learning/` — What should persist from this episode? Carries governance conversion and recurring failure analysis.

The facets answer independent questions: recognizing the situation does not select the model, choosing a move does not determine its authority, and knowing the method does not supply system facts. Load only the facet the current question requires.

Governing Principle

Apply MAGE to the work itself: model what must be understood, give authority to what must hold, and convert recurring judgment into durable engineering structure. Five operating rules follow:

- **Model before guessing.** Reach concrete system truth through a provider; do not reconstruct it from memory or raise confidence by rhetoric.
- **Ask what should become machinery.** At every failure, ask whether the lesson deserves durable structure—and default to no. Most failures should not become new governance.
- **Method before tool.** Choose the move the situation calls for, then the capability that performs it, not a capability because it is at hand.
- **Right-size the fix.** Prefer the smallest sound change that closes the class; propose a larger intervention rather than implementing it reflexively. Prevention by construction beats detection where the action space can honestly close.
- **Escalate judgment, not difficulty.** Escalate when authority or consequence requires a human decision, not merely because the problem is hard.

Self-governance is not self-certification. The agent may select, model, propose, and check, but its belief that the work is correct is not evidence that it is. A skill is soft: it guides a probabilistic agent but cannot block. Hard mechanisms may be proposed or scaffolded by the skill, but enforcement must come from the harness or another external mechanism. The deciding evidence comes from a mechanism that sits outside the reasoning which produced the change.

Layout

```
self-governance/
  SKILL.md            the router — the loop, the routing table, the ambient stance
  principles.md       the portable engineering method (the ambient reflexes)
  modeling/    repertoire.md  moves.md
  practice/    situations.md  judgment.md
  system/      model-access.md
  skills/      repertoire.md
  alignment/   repertoire.md  mechanisms/INDEX.md  mechanisms/<target>/<family>/<mechanism>.md
  learning/    governance-conversion.md
  templates/   system-models-starter-kit.md  state-machine-model-starter.py  …
               — scaffolds for creating system models, never system-specific truth
```

One directory per question, one file per facet. The mechanism census lives under `alignment/`, where prevent-versus-detect now sits one level below the loop.

Lesson

Self-governance models the MAGE method itself. Its fundamental model is therefore the engineering loop, not a taxonomy of mechanisms.

The Modeling–Alignment boundary matters especially here: self-governance can propose and scaffold a constraint, validator, or gate, but the resulting mechanism — not the skill's judgment — must supply the enforcement.

It also composes with the other skills: self-governance designs models and mechanisms, self-operate runs them, and self-communicate governs how they are represented.

E.2.3 self-operate

Problem

Operating an agent-fleet repository involves recurring but heterogeneous tasks — dispatch and recover agents, keep the mainline deployable, reclaim disk, weather colima and host-tool trouble, watch cron health, RCA an ambiguous signal. Without a model, each failure must be diagnosed almost from scratch. The skill provides a lifecycle map that routes symptoms to classes and typed runbooks that make responses repeatable.

Fundamental Model

The engineering lifecycles. A fleet repo runs the same few: manage-agents, manage-context, manage-git-repo, manage-deploy, manage-dev-env, plus cron and govern-your-own-loop. Every symptom belongs to one lifecycle, so every break routes to a *class* instead of being met cold. The lifecycle map is the fundamental model: it turns operational sprawl into a small set of recurring classes.

Orthogonal Models

- **Lifecycle map / symptom catalog** — identifies which operational lifecycle owns the problem.

- **Typed runbooks** (`examples/runbook-*.md`) — specify what to do, distinguishing runnable, judgment-automatable, and judgment-irreducible steps.
- **Hooks** (`hooks/`) — decide when a known reaction should fire automatically.

These answer three independent questions: where the problem belongs, what response it requires, and when that response should trigger.

Supporting resources: build and handoff templates (`templates/`) — used when operating work crosses into implementation.

Governing Principle

Establish the healthy state first, then classify deviations from it. Route each symptom to a lifecycle and run the corresponding procedure. When diagnosis reveals that a model, mechanism, or authority should change, hand that engineering decision to self-governance. Automate deterministic steps; prepare irreducible judgments for escalation.

Layout

```
self-operations/
  SKILL.md
  principles.md
  examples/   lifecycle-L1..L6-*.md   runbook-*.md
  hooks/      reflection_facet*.py  hook_*.py  _hook_*.py  README.md  ...
  templates/  pointers-starter.yaml  runbooks-starter.yaml
              gen-and-lint-partb-starter.py  EPIC-TEMPLATE-starter.md  design-doc-template-starter.md
```

Lesson

The lifecycle model turns incident response into routing: identify the affected lifecycle, select the corresponding runbook, and determine which steps can be automated. Typed runbooks distinguish runnable steps from automatable and irreducible judgment. Self-governance changes models and mechanisms; self-operate runs them and returns evidence; self-communicate governs their representations.

Figure E.2-1 shows the three skills acting on one governed engineering environment: self-governance changes it, self-operate runs it and returns evidence, and self-communicate governs its representations.

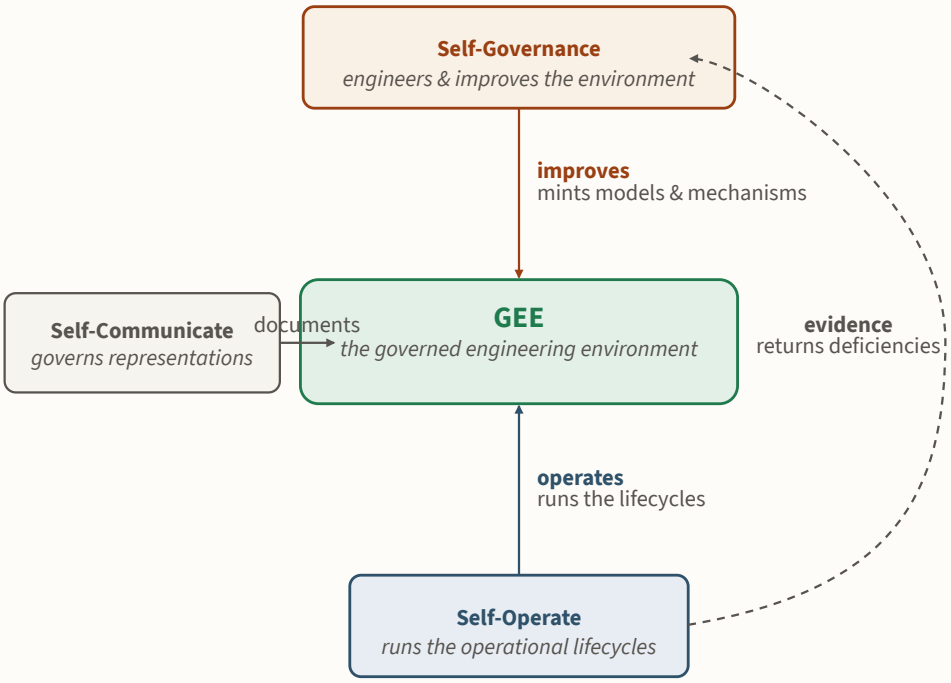


Figure E.2-1: Three orthogonal skills act on one governed engineering environment. Self-governance improves it; self-operate runs it and returns evidence; self-communicate governs the representations both produce.

Orthogonality does not require isolation. Each skill has one reason to change—communication, engineering judgment, or operation—and explicit interfaces connect them. Self-governance produces models and mechanisms that self-operate uses; self-operate returns evidence; self-communicate governs their representations.

Appendix F: Configuring MAGE Across the Product Lifecycle

Appendix F — Configuring MAGE Across the Product Lifecycle

Much of this book explains MAGE through software realization and evolution. A software product, however, is shaped across a broader lifecycle. Product staff discover what should be built; engineers realize and evolve it; operators run the resulting system and investigate its failures; assurance and compliance work establishes claims about its quality and obligations. These activities ask different engineering questions and therefore call for different configurations of MAGE.

The same modeling principle applies across that lifecycle. A product hypothesis, structured ticket, state machine, operational topology, runbook, incident timeline, or assurance case can each preserve information needed to answer an engineering question. Their purposes, lifetimes, correspondence claims, authority, and uses differ. The surfaces differ in what they need from MAGE. Discovery may preserve uncertainty; Engineering may need explicit system models and strong Alignment; Operations may turn recurring judgment into procedure; Assurance may require evidence connecting models to realization. There is no canonical portfolio of MAGE models or fixed boundary between reasoning and deterministic machinery.

Nor does MAGE require product-wide adoption. Models and mechanisms have construction and carrying costs, and their returns vary by engineering surface and obligation. An organization might begin with ticket-driven change, architecture, operations, or one consequential compliance obligation while leaving the rest of its practice largely unchanged. This appendix examines five recurring surfaces—Product Discovery, Engineering & Realization, Product Management & Maintenance, Operations & Incidents, and Assurance & Compliance—and then considers how useful local investments can connect across the lifecycle. Organizations divide this work differently, and information flows repeatedly among the surfaces. The relevant question is not whether an organization has "adopted MAGE," but where durable representation and authority repay their cost.

F.1 One Product, Several Engineering Surfaces

Figure F.1-1 places the product amid the engineering activities that shape it.

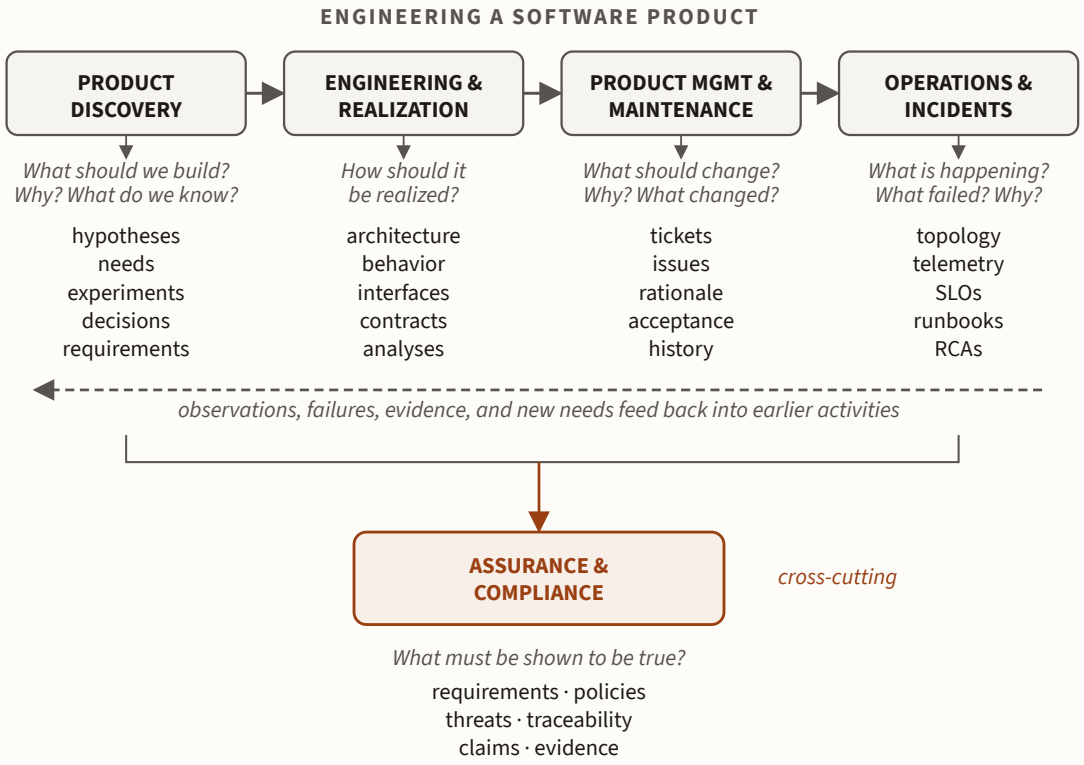


Figure F.1-1: MAGE across the product lifecycle. *Different lifecycle surfaces ask different engineering questions and therefore benefit from different representations. The horizontal flow is not a waterfall: experience and evidence feed earlier activities, while Assurance & Compliance spans the lifecycle. Each surface can adopt MAGE independently.*

The representations differ along five dimensions.

1. **Purpose.** Each representation preserves what a particular engineering question needs. A product hypothesis helps decide what to build; an architectural model exposes system structure; an incident timeline reconstructs what happened. They need not share a form merely because they concern the same product.
2. **Lifetime.** A model may be episode-scoped, change-scoped, system-lived, or organizational. An experiment may last for one investigation; a ticket may govern one change; an architectural boundary may survive hundreds of changes; a regulatory policy may govern several products.
3. **Claimed correspondence.** Age alone does not establish model drift. A 2022 ticket can remain a faithful representation of the intent governing a 2022 change without describing the product in 2026. MAGE rejects “keep the model equal to the code” as a universal synchronization rule because different models claim different relations to the realized system.

4. **Authority.** Representation does not imply authority. A discovery hypothesis may deserve preservation without enforcement; an accepted security invariant may deserve both.
5. **Reasoning posture.** Some representations support semantic or situational judgment; others expose stable properties that can become deterministic procedures or predicates. The appropriate mix depends on cost and consequence.

The five surfaces configure these dimensions differently. The sections that follow examine each in turn.

F.2 Product Discovery

Engineering question. What should we build, and why?

Product discovery operates where uncertainty is often legitimate. Customer needs clash, features begin as hypotheses, requirements are negotiated, and experiments exist because important questions remain unsettled.

Product discovery therefore needs to preserve what the team knows and why it made particular choices. Useful models can include stakeholder needs, product hypotheses, experiment definitions and results, prototypes, decision records, provisional requirements, and relationships among them. These representations preserve what the organization knows about the product problem rather than primarily describing its implementation.

Product discovery therefore usually needs light Alignment. Hypotheses, evidence, and provisional decisions can be worth preserving without becoming obligations that realization must satisfy. Provenance matters: where did a claim come from? Relationships matter: which evidence bears on which hypothesis? Promotion matters: which provisional decision has become accepted product intent? Much of the judgment remains human because discovery is partly the work of deciding what the obligations should be. Rationales should be recorded, but extensive formalization is often unnecessary.

Agents can still provide substantial leverage. A reasoner can retrieve experiments relevant to a proposed capability, identify earlier product decisions it appears to contradict, trace stakeholder needs, compare competing evidence, or reconstruct why a previous direction was abandoned. Externalizing that knowledge reduces reconstruction without pretending the resulting judgment is mechanical.

Discovery also exposes the distinction among unknown, tacit, and free. An apparently unconstrained choice may be genuinely open, may conceal an obligation not yet discovered, or may depend on one stakeholders have not articulated. Discovery helps determine which.

MAGE profile.

- *Characteristic models.* Needs, hypotheses, experiment results, product decisions, provisional and accepted requirements.
- *Lifetime.* Mostly episode- and product-lived; decisions and requirements may outlive the investigations that produced them.
- *Alignment posture.* Light. Preserve provenance and decision status; avoid prematurely giving hypotheses mechanical authority.
- *Role of autonomous reasoning.* Synthesis, retrieval, comparison, ambiguity, and reasoning across competing evidence.
- *Determinization opportunity.* Limited while the underlying product question remains genuinely unsettled. Stable process obligations may still be mechanized.
- *Degrees of freedom.* Product choices that stakeholders have genuinely left open.
- *Smallest useful adoption.* Persist consequential product decisions, their rationale, and their relationships to needs and evidence; make that structure available to later reasoning.

- *Lifecycle connections.* Requirements entering Engineering and Assurance; change intent entering Product Management; operational evidence that tests product assumptions.

F.3 Engineering & Realization

Engineering question. How should the product or requested change be realized while satisfying its obligations?

Engineering & Realization is the surface treated throughout Parts II–IV. Its characteristic models include structural, behavioral, ownership, decision, measurement, and provenance views of the realized system (the six model classes, Figure 2.1-4). The lifecycle question here is not what those models are, but where their knowledge and obligations come from and where they go.

Engineering does not manufacture all of its own obligations. Product discovery supplies needs, accepted requirements, and consequential product decisions. Product management supplies the intent and rationale for particular changes. Assurance contributes obligations that realization must satisfy. Operations supplies evidence from the behavior of the realized product.

Engineering connects those inputs to the system being changed and exposes structure that neighboring surfaces can reuse: component identity, architecture, behavior, interfaces, dependencies, resource relationships, and evidence about the realized change. Shared identity lets the same entity connect lifecycle surfaces. A requirement from discovery can trace to a component and behavioral obligation in engineering; operational evidence can resolve to the same component; an assurance claim can refer to the requirement, the governing model, and evidence from the realized system.

Parts II–IV supply the corresponding Modeling and Alignment practices.

Engineering need not eliminate realization freedom. Governing models determine some choices while deliberately leaving others open; new functionality may create choices, a newly discovered obligation may remove apparent freedom, and stronger Alignment may constrain a choice previously left open. The relevant question for any realization is which choices the governing obligations determine and which remain free.

MAGE profile.

- *Characteristic models.* Structural, behavioral, ownership, decision, measurement, and provenance models of the realized system; commonly including architecture, state machines, contracts, dependencies, knowledge graphs, resource relations, and invariants.
- *Lifetime.* Change-scoped through system-lived, depending on the model and the correspondence it claims.
- *Alignment posture.* Strong where consequential properties are decidable: tests, analyses, validators, model checking, constraints, permissions, and gates.
- *Role of autonomous reasoning.* Reasoning across interacting semantic constraints and realizing changes where governing obligations are explicit but implementation remains underdetermined.
- *Determinization opportunity.* High where modeling exposes stable properties that deterministic machinery can evaluate or realize.

- *Degrees of freedom.* Implementation choices left open after the governing obligations for the realization are accounted for.
- *Smallest useful adoption.* Externalize one repeatedly reconstructed or consequential system property, connect it to the surrounding lifecycle knowledge it depends on, and give stable obligations over it proportionate evidence and authority.
- *Lifecycle connections.* Product decisions and requirements from Discovery; change intent from Product Management; operational topology and evidence from Operations; assurance obligations and evidence requirements across the surface.

F.4 Product Management & Maintenance

Engineering question. What change are we asking for, and why?

Many software organizations already possess a lightweight modeling substrate without calling it one: the ticketing system. A structured ticket can preserve desired behavior, rationale, acceptance criteria, affected concepts, constraints, related changes, and prior decisions. When it carries enough information to reason about one requested change, it functions as a purposeful change-scoped model. It need not duplicate longer-lived models; it should connect the episode to the product and system knowledge that governs it.

The change-scoped nature of the ticket matters. A ticket written in 2022 need not describe the product in 2026, but it can remain an accurate representation of the intent governing the 2022 change. Its persistence is then useful history rather than a synchronization burden. Maintenance uses the same pattern: an issue can preserve the discrepancy, intended correction, rationale, and acceptance evidence.

For many tickets, however, specifying the requested outcome is not the difficult part. The difficult part is diagnosis: determining why the observed behavior occurs, locating the responsible concepts and components, and identifying the system models and obligations that govern a correct change. This is especially apparent in defect repair, where root-cause analysis may dominate the work. MAGE's bidirectional traceability between models and implementation reduces that search problem. An observation in the realized system can be traced from code to the relevant architectural, behavioral, ownership, or other system models; reasoning over those models can in turn identify the implementation that realizes them. The ticket can therefore become a point of entry into the governed engineering environment rather than a standalone description from which an agent must reconstruct the system.

A ticket also reopens a particular region of the product to choice. Existing intent, architecture, contracts, and acceptance machinery may already settle most consequential questions, leaving an agent only implementation choices. Other changes expand the realization surface, expose a tacit obligation, or require new design judgment. Maintenance therefore does not simply consume a fixed stock of degrees of freedom; each change encounters and may alter the choice space defined by current obligations.

Figure F.4-1 brings these relationships together.

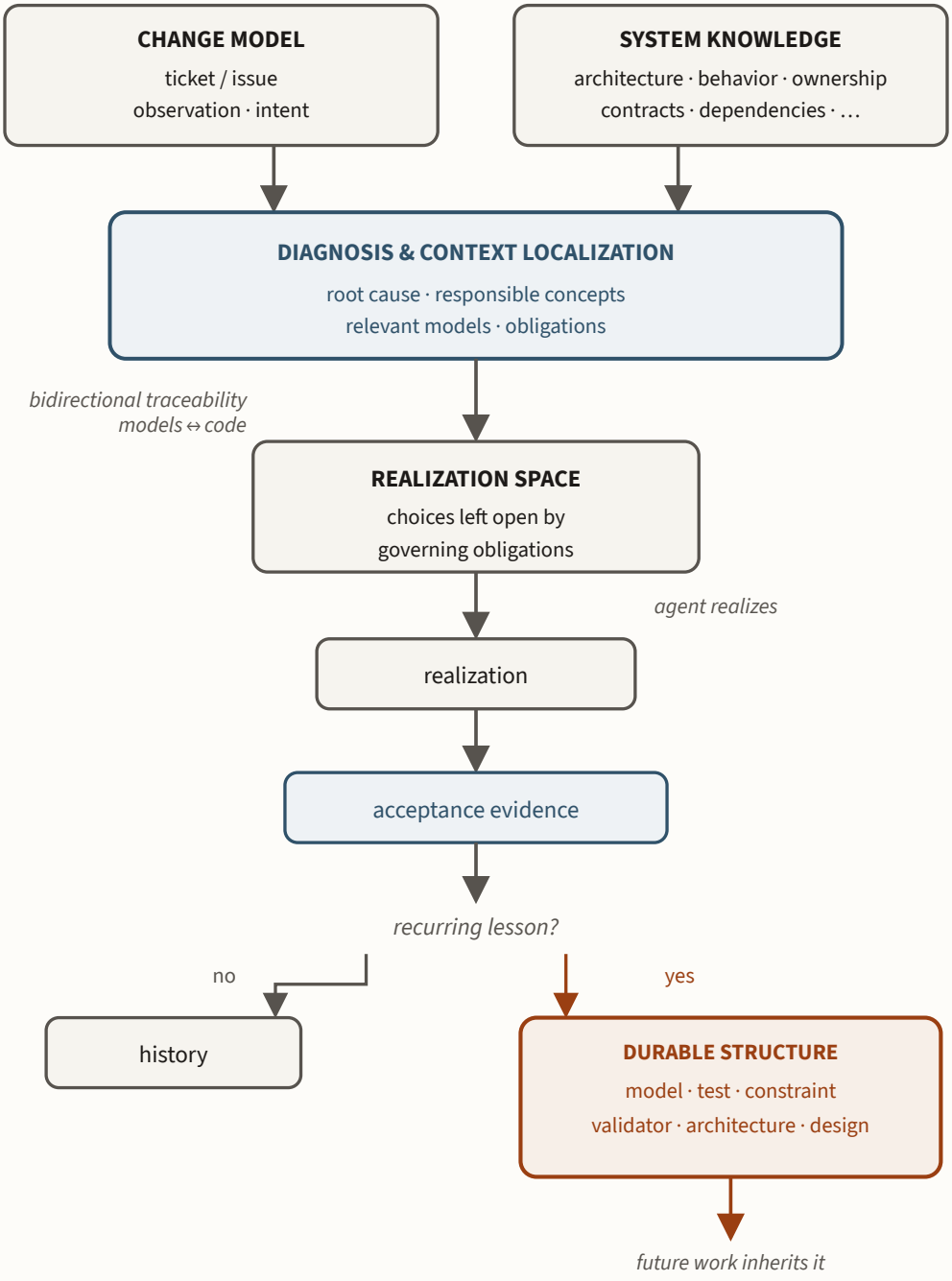


Figure F.4-1: Change-scoped models, degrees of freedom, and inheritance. *A ticket or issue provides an entry point into a change episode. Bidirectional traceability helps locate the responsible implementation and the models and obligations that govern it. Those inherited obligations bound the realization space while leaving some choices open. Acceptance evidence establishes whether the realized change satisfies its obligations; recurring or consequential lessons can then be converted into durable structure that future work inherits.*

Maintenance is not inherently more automatable than new engineering; its advantage is inheritance. A mature product may already provide the models, obligations, evidence machinery, traceability, and prior decisions that settle questions a greenfield change must answer. When they cover the requested change, realization becomes cheaper. When the ticket exposes a missing capability, tacit obligation, architectural conflict, or new design question, judgment returns.

Ticketing is an accessible MAGE surface because many organizations already preserve change-scoped intent. The useful move is not to enrich every ticket, but to connect consequential change intent to the longer-lived knowledge and evidence needed to govern the change.

MAGE profile.

- *Characteristic models.* Tickets and issues containing change intent, rationale, constraints, acceptance criteria, and links to longer-lived product and system models.
- *Lifetime.* Primarily change-scoped, with historical value after the change is complete.
- *Alignment posture.* Workflow state, acceptance evidence, regression tests, and admission gates, backed by the longer-lived obligations relevant to the change.
- *Role of autonomous reasoning.* Diagnosing reported problems, locating the relevant system context, interpreting change intent against it, and realizing choices that governing obligations leave open.
- *Determinization opportunity.* Recurring classes of changes, acceptance conditions, and repair obligations can migrate into deterministic machinery.
- *Degrees of freedom.* Choices left open by the obligations governing the particular change. A change can preserve, narrow, expose, or expand that choice space.
- *Smallest useful adoption.* Treat sufficiently structured tickets as change-scoped models, connect them through traceability to relevant persistent knowledge and acceptance evidence, and allow later work to inherit rather than reconstruct that context.
- *Lifecycle connections.* Product rationale and accepted intent from Discovery; persistent system models and their implementation from Engineering; corrective work originating in Operations; assurance obligations and evidence affected by the change.

F.5 Operations & Incident Response

Engineering question. What is the system doing, why is it doing it, and what should we learn from failure?

Operations works against the realized system. Its characteristic representations include deployment topology, service dependencies, runtime configuration, telemetry, resource relationships, runbooks, incident timelines, and operational policies. These models answer questions that source code alone often answers poorly: what is running where, which services depend on which others, what resources are shared, what happened before a failure, and what operators should do next. Because these models describe the running system, many require strong correspondence. A topology presented as current must describe the deployed system closely enough to support operational reasoning; telemetry must refer to identifiable entities and states; and a runbook intended to govern response must remain applicable to the system operators actually face.

Incidents expose where the governed environment was incomplete or wrong. Restoring service fixes the instance; root-cause analysis asks what condition made the failure possible. Bidirectional traceability can connect an operational observation to the responsible implementation and from there to the architectural, behavioral, ownership, resource, or other models that describe the affected concepts.

Explicit models can also reveal where else the same condition exists. Two failures need not share similar source code to share an engineering structure: components may occupy the same modeled role, implement analogous transitions, cross equivalent trust boundaries, share ownership relations, or depend on resources with the same lifecycle. Models therefore let engineers search for the class rather than only for syntactically similar code. The ambition is longstanding: repair the class rather than the instance.

Finding the class is only half the job. Governance conversion determines whether the lesson survives the incident: correct a wrong model, represent missing knowledge, or encode a stable obligation in a validator, constraint, policy, test, or gate. The strongest outcome is not merely a successful patch, but an environment in which future work inherits what the incident taught.

Figure F.5-1 shows the progression from one observed failure to a durable repair of the class.

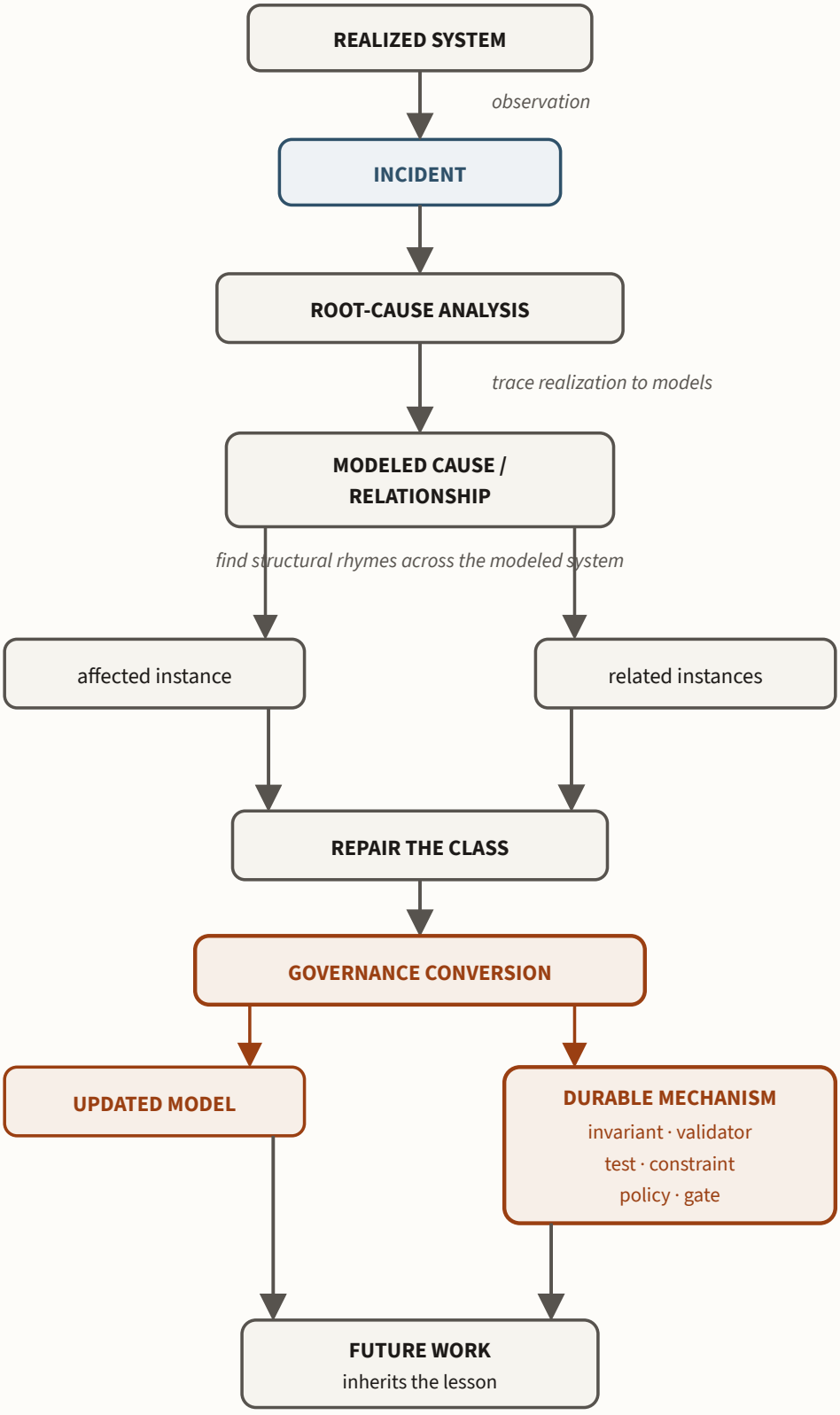


Figure F.5-1: From incident repair to governance conversion. *An incident provides evidence about the realized system. Traceability connects the failure to the models and obligations it realizes; relationships within those models can expose other instances of the same engineering condition even when their implementations differ. Repair can then address the class rather than only the observed instance, while governance conversion preserves the lesson in models or mechanisms that future work inherits.*

Not every incident generalizes, and repairing defect classes is not new. Root-cause analysis, defect prevention, static analysis, and related practices have long sought broader corrective action. MAGE adds an explicit representational substrate for finding analogous conditions and places to encode the resulting lesson when it can responsibly acquire authority.

Operations grows engineering capital when what one incident teaches changes what later engineering inherits.

MAGE profile.

- *Characteristic models.* Deployment topology, service and resource dependencies, runtime configuration, telemetry, operational state, runbooks, incident timelines, and operational policies.
- *Lifetime.* Mixed. Telemetry and incident evidence may be episode-scoped; topology, dependencies, policies, and runbooks are generally system-lived and require stronger synchronization with the realized system.
- *Alignment posture.* Strong where operational state or policy can be checked mechanically: deployment validation, health checks, policy enforcement, admission controls, automated response, and operational gates.
- *Role of autonomous reasoning.* Diagnosing incidents across runtime evidence, implementation, and system models; identifying root causes and structurally related instances; proposing corrective action; and recognizing lessons that should be converted into durable governance.
- *Determinization opportunity.* High for stable operational predicates and recurring failure classes once their governing conditions have been identified.
- *Degrees of freedom.* Operational and corrective choices left open by existing obligations. Incidents may reveal that an apparent freedom was actually constrained by an unmodeled or tacit obligation.
- *Smallest useful adoption.* Connect one recurring class of operational failure to the system models needed to diagnose it, then convert the resulting lesson into a durable representation or mechanism rather than repeatedly repairing individual instances.
- *Lifecycle connections.* Realized structure from Engineering; corrective work flowing into Maintenance; operational evidence supporting Assurance; recurring lessons converted into engineering capital for future work.

F.6 Assurance & Compliance

Engineering question. What must we be able to demonstrate about the product?

Assurance and compliance organize work around claims that must be established, not merely implementations that must be produced. Some claims are naturally evaluated over models; others concern the realized system; many require evidence connecting the two.

A security claim may be evaluated over a data-flow or access-control model; a memory-safety claim concerns the code that executes. Where implementation is not wholly generated from authoritative models, assurance must span both. Proving a property of an access-control model does not establish that the implementation realizes it, while inspecting implementation alone may obscure the intended policy. Traceability and Alignment connect the modeled claim to evidence from its realization and, where necessary, outward to requirements, operational measurements, and other lifecycle evidence.

Figure F.6-1 separates model evidence, realization evidence, and the traceability needed when a claim depends on both.

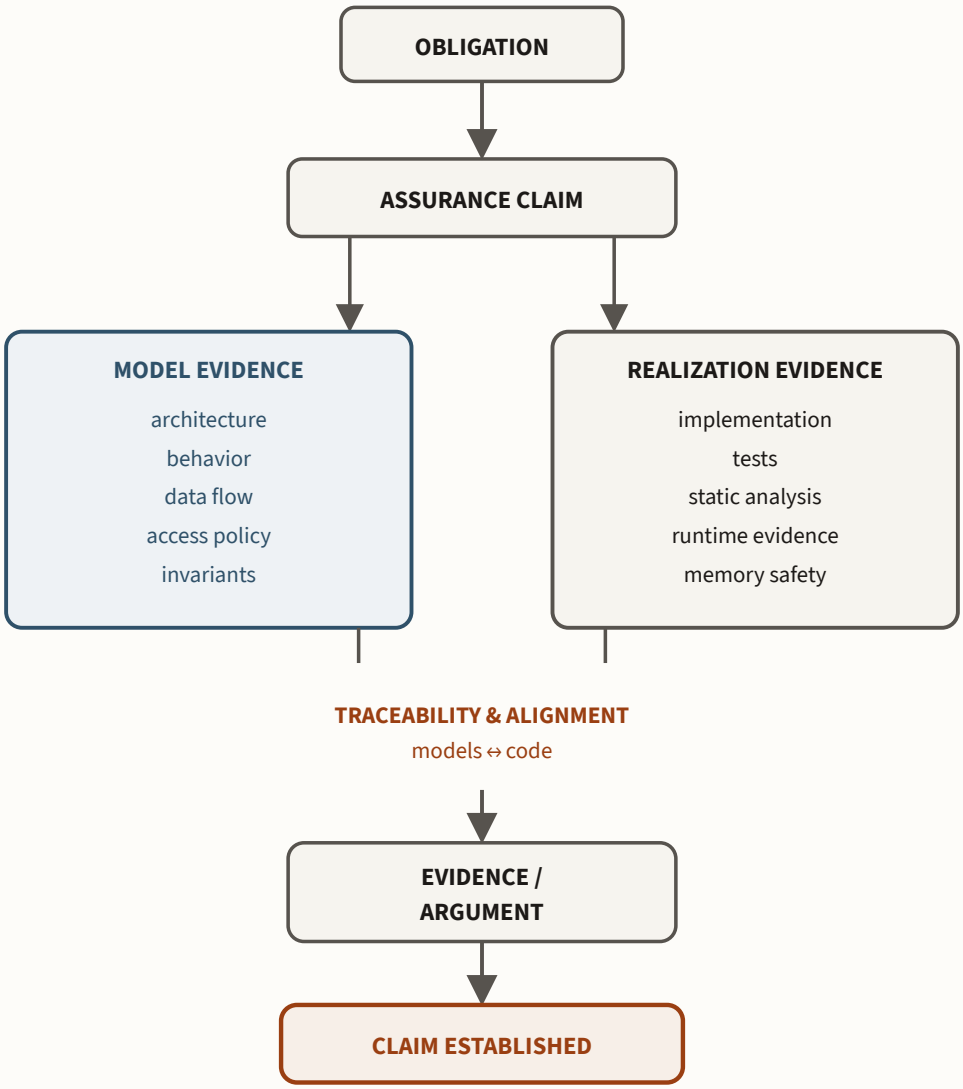


Figure F.6-1: Assurance across models and realization. *Claims may depend on modeled properties, realized properties, or correspondence between the two. Traceability and Alignment connect the required evidence.*

Assurance creates pressure to reduce degrees of freedom only where the return justifies it. Authoritative models and trustworthy transformations can move assurance toward the governing model and away from independently authored realization, but eliminating freedom costs modeling effort and may make later changes harder. Other properties are cheaper to establish over the realized system. The same tradeoff governs determinization: mechanize claims that can be decided responsibly; leave statistical, semantic, or normative judgment where it belongs.

Agentic realization may also shift verification away from lightweight code review toward more formal inspection.[39] Contemporary review assumes changes arrive slowly enough that peers can reconstruct intent, context, and obligations from diffs, comments, tests, and local judgment. Agent-generated changes stress that assumption. Inspection may therefore become a distinct activity organized around explicit obligations, models, traceability, procedures, and evidence rather than an informal second look at code.

Such inspection could begin from a ticket or assurance claim, traverse to the governing models, inspect their realization where necessary, and judge whether the evidence establishes both correspondence and satisfaction. The analogy to classical software inspection is structural rather than procedural: production and verification separate more clearly as realization becomes cheaper.

Assurance is therefore cross-cutting rather than a final lifecycle stage: its obligations constrain work elsewhere, and its evidence can originate anywhere in the lifecycle.

MAGE profile.

- *Characteristic models.* Requirements, policies, threats and hazards, invariants, traceability, assurance cases, claims, and evidence models.
- *Lifetime.* Usually system- or organizational-lived, although individual evidence can be change- or execution-scoped.
- *Alignment posture.* Strong. Assurance may require evidence over models, over their realization, and over the correspondence between them.
- *Role of autonomous reasoning.* Interpreting semantic obligations, reasoning across modeled and realized properties, maintaining traceability, assembling evidence, identifying gaps, and evaluating claims that machinery cannot settle.
- *Determinization opportunity.* High for decidable claims over either models or implementation; limited where interpretation or normative judgment remains necessary.
- *Degrees of freedom.* Realization choices left unconstrained because they are irrelevant to the assurance obligation or cheaper to inspect than to specify in advance.
- *Smallest useful adoption.* Select one consequential obligation, identify what must be established over models and realization, connect those artifacts through traceability, and automate evidence or admission where the claim is decidable.
- *Lifecycle connections.* Assurance carries obligations across the lifecycle and draws evidence from product decisions, engineering models, implementation, change history, and operational behavior.

[39] Formal software inspection has a substantial history. Michael Fagan's work at IBM distinguished inspection from informal review by treating it as a systematic verification activity with defined roles, procedures, and recorded results. The analogy here is structural rather than procedural: MAGE does not propose restoring the original inspection process, but predicts renewed pressure to formalize inspection as realization becomes cheaper and more abundant. See Fagan 1976[118] and Fagan 1986[119].

F.7 Differential Adoption

The five surfaces need not mature together. A startup may use stronger MAGE machinery only around payments or customer data; a service organization may have sophisticated operational models while leaving discovery informal; a regulated product may begin with assurance and traceability. These are not incomplete implementations awaiting product-wide rollout. They are rational local configurations when the investment pays.

The relevant question is what knowledge or obligations are consequential or expensive enough to deserve durable representation or authority. That answer changes over a product's life: repeated reconstruction may eventually justify an architectural model, a frequently used runbook may become deterministic automation, or an incident may expose an obligation previously left tacit. MAGE is therefore not a maturity ladder toward maximal modeling or determinization. Different surfaces legitimately preserve different mixtures of explicit representation, autonomous judgment, and machinery.

Broader value appears when independently useful investments begin to connect. A requirement represented for product reasons may later govern engineering; an architectural model may aid incident diagnosis; an operational failure may expose an invariant for future changes; assurance may reuse relationships created elsewhere. H.8 asks what emerges when these local investments begin to compose.

F.8 From Local MAGE to the Product GEE

When useful local representations connect across lifecycle surfaces, they can form a product-wide Governed Engineering Environment: product intent, system knowledge, operational experience, and assurance obligations become available across the life of the product. The product GEE emerges from connections among representations that remain suited to their own engineering purposes.

Each surface should retain representations suited to its own questions. What they can share is identity, relationships, provenance, temporal scope, and obligations. A requirement can concern a feature; a ticket can authorize a change to a component; a deployed service can realize that component; an incident can expose a violated assumption; the resulting invariant can constrain later engineering and support an assurance claim. Figure F.8-1 shows those connections.

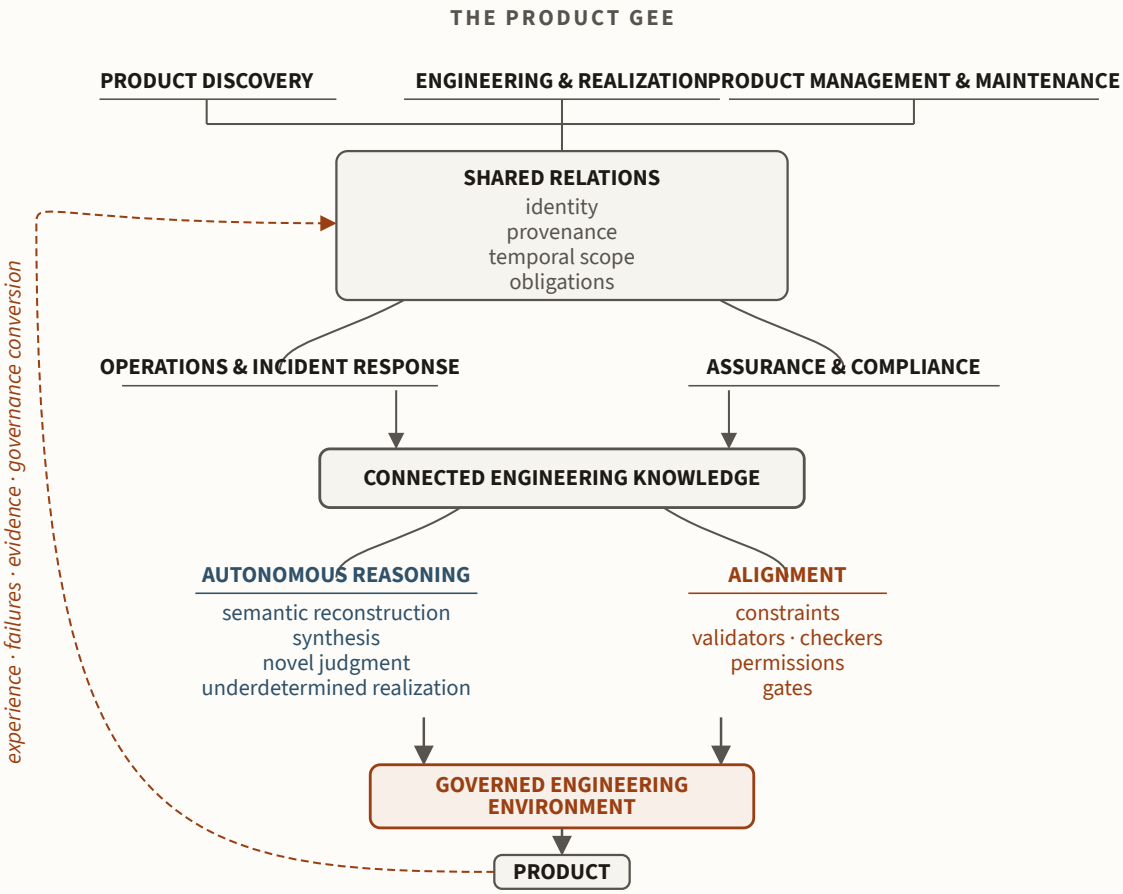


Figure F.8-1: From local MAGE adoption to the product GEE. *Each lifecycle surface develops representations suited to its own reasoning problems. Shared identity, relations, provenance, temporal scope, and obligations allow those heterogeneous models to compose without collapsing them into one universal model. Agents reason across the resulting knowledge where semantic or situational judgment remains necessary; Alignment carries obligations that can responsibly be made authoritative. Experience from the realized product feeds governance conversion, changing the models and mechanisms future work inherits.*

This need not be a lifecycle pipeline or a single knowledge system. Tickets, architecture registries, source repositories, telemetry platforms, requirements systems, and evidence stores can remain distinct as long as consequential relationships can be recovered reliably. Shared identifiers, typed relationships, provenance, indexes, APIs, or a knowledge layer can supply the joins.

The result changes what a short instruction or question can mean because the environment supplies context that would otherwise have to be reconstructed inside each reasoning episode:

- *Implement ticket X.* The environment can supply the product decision that motivated the change, the architecture it touches, the obligations governing those components, the incidents that made particular constraints important, and the evidence required before the change can be admitted.
- *Why does the system behave this way? Can we change it?* The environment can traverse from implementation through architecture and change history to product intent, then across dependencies, operational constraints, assurance obligations, tests, and migration history. The same connected knowledge can turn *What did this incident teach us?* into a question about which models and mechanisms future work should inherit.

Product-wide MAGE is a possibility, not a prerequisite. Each representation, relation, and mechanism must repay its carrying cost through better reasoning, less reconstruction, stronger evidence, safer autonomy, cheaper recovery, or another benefit to later work. If product-wide MAGE emerges, it should do so by connecting useful local investments—not by specifying the whole product in advance.

The same approach applies at larger scales. A team can connect local models to shared architecture; a product can connect models across lifecycle surfaces; and an organization can connect products through common policies, assurance obligations, infrastructure, or reusable engineering judgment. The representations need not collapse into one system. They need enough shared identity and correspondence for the relevant knowledge and obligations to travel between them.

F.9 Engineering Capital Across Time and Space

Engineering capital matters when an investment made in one place changes the economics of work elsewhere. Within a product, that effect is already visible: product decisions constrain later realization, engineering models support maintenance, incidents produce durable controls, and assurance evidence can accumulate rather than be rebuilt. A more conjectural question is whether such capital can also travel across products, teams, and organizations.

Figure F.9-1 depicts the two dimensions. Within each product, engineering structure accumulates selectively across episodes. Across products, some concerns and mechanisms may travel while others remain local. The figure is conceptual, not a maturity model or quantitative measure of capital.

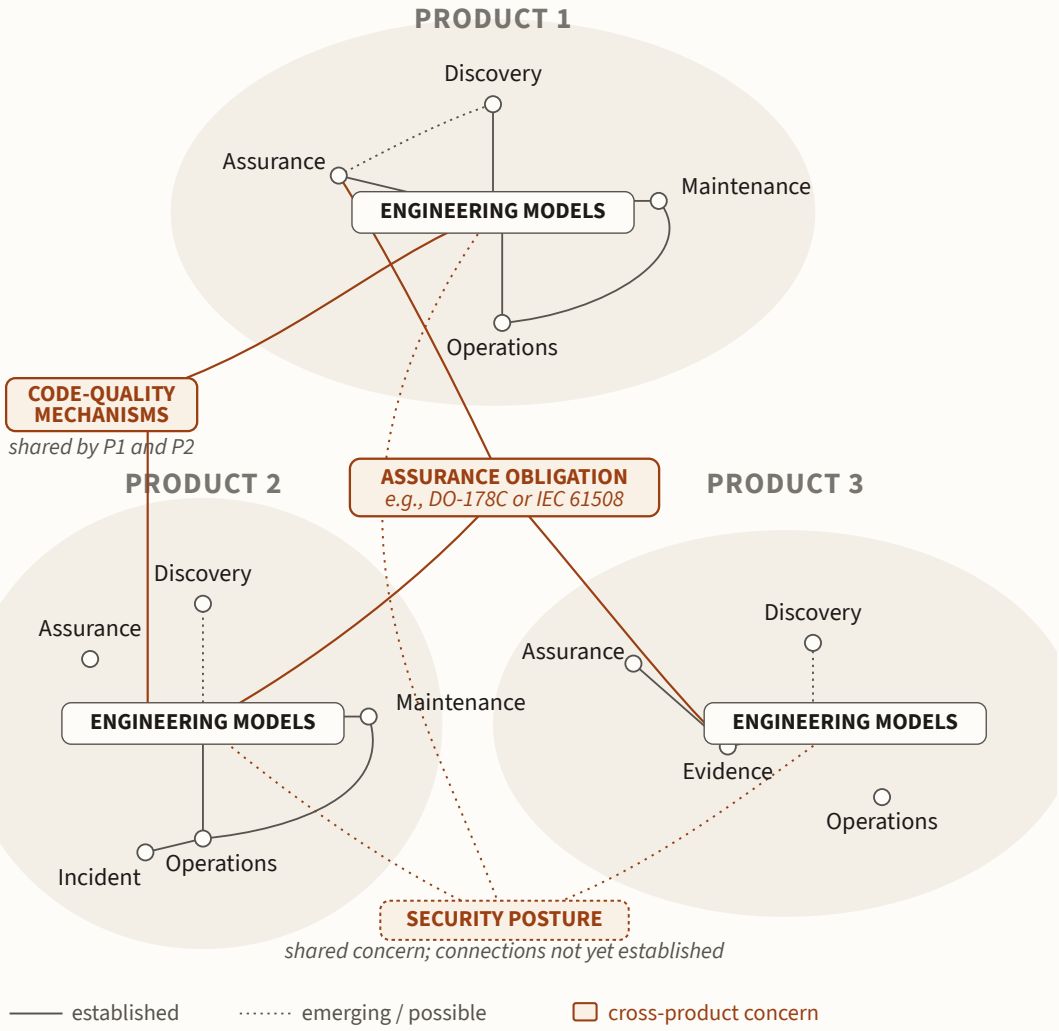


Figure F.9-1: Engineering capital across time and space. *A conceptual snapshot of three products shaped by successive engineering episodes. Within each product, useful engineering structure has accumulated selectively; the incomplete constellations represent neither a prescribed product model nor a maturity level. Across products, an assurance obligation connects all three, code-quality mechanisms are shared by two, and connections to a common security posture remain possible. Solid relationships are established; dotted relationships are emerging or possible.*

F.9.1 Across Time: Capital Across the Product Lifecycle

Engineering capital does not appreciate by eliminating every future choice. A useful model fixes what later work should not have to rediscover while leaving irrelevant choices open. Over-specification creates carrying cost and can make later change more expensive. The

valuable asset is therefore not maximal explicitness, but durable knowledge and authority over matters consequential enough to justify their cost.

Engineering capital can also gain new uses. An architectural model built to reduce reconstruction may later support incident diagnosis; maintenance traceability may support Assurance; an operational failure may produce an invariant useful to both Engineering and compliance. The value of the capital is therefore not the amount of structure accumulated, but the future work it changes.

F.9.2 Across Space: Reusing Engineering Capital

Some engineering capital is product-local: the rationale for one change, the state of one system, or one deployment topology. Other assets may apply across products: security policies, regulatory obligations, protocols, accessibility requirements, architectural conventions, assurance arguments, validators, or evidence procedures. The question is whether capital that pays repeatedly across time can also travel across products, teams, or organizations.

Software engineering has pursued versions of this promise before. Object-oriented and component-based development sought reusable implementations; design patterns sought reusable engineering judgment. In practice, the legacy was at least as much a change in engineering principles—encapsulation, interfaces, composition, separation of concerns, recurring patterns—as direct reuse. Application-specific context and judgment remained difficult to package.

Commodity intelligence changes one premise. General reasoning capacity can be supplied separately from task-specific context and judgment. Code can carry implementation; models can carry represented context; judgment as code can carry decisions about obligations, evidence, and acceptable action. The consumer of reusable engineering judgment need no longer be exclusively human.

This permits reuse without prescribing a common realization. A security obligation, protocol model, accessibility rule, or assurance procedure can govern different implementations while leaving other choices open. A library gives later engineering something already built; reusable judgment gives it something already learned.

Whether commodity intelligence makes engineering judgment substantially more portable remains an empirical question. Context still travels badly. A judgment separated from the assumptions that made it valid can be worse than no reuse at all. Cross-product reuse therefore needs enough scope, provenance, applicability, and dependency information to determine when an asset still applies. The spatial reach of engineering capital must be tested, not assumed.

F.9.3 Engineering Teams

Commodity intelligence and reusable engineering capital may also change how engineering responsibility is divided. An engineer supported by autonomous realization may own more

components or a larger subsystem. More speculatively, engineers may own cross-cutting concerns—an architectural relation, authorization model, lifecycle protocol, resource invariant, accessibility obligation, deployment policy, or assurance claim—whose realizations span many components. Component ownership need not disappear, but implementation boundaries may cease to be the only natural unit of responsibility.

Smaller teams create a second risk. Large projects pay coordination costs, but they also aggregate diverse judgment: different engineers notice different assumptions, challenge different decisions, and recognize different failure modes. If autonomous realization lets fewer people govern the same scope, coordination cost may fall along with that diversity. A GEE does not automatically replace it: models preserve what was represented, and Alignment governs obligations already identified.

The organizational problem may therefore shift from coordinating enough people to realize the system toward ensuring enough independent judgment over it. Teams may need deliberate substitutes for perspectives once supplied incidentally by scale: independent inspection, heterogeneous agents, adversarial reasoning, or independent evidence. If fewer people can govern more engineering, preserving diversity of judgment may itself become an engineering obligation.

Appendix G: Adopting GenAI in an Organization

Appendix G — Adopting GenAI in an Organization

This appendix is an organizational adoption guide, not a restatement of the MAGE method. It assumes the problem developed in Part I and draws most directly on the industrial evidence in Section 5.5 and the economics of explicit engineering in Section 6.3. Readers approaching MAGE primarily as an organizational problem can follow Part I → Section 5.5 → this appendix. The key MAGE concepts are briefly glossed here where needed; Parts II–IV develop the underlying engineering practices, and Part VI develops the full theory.

Organizations are already adopting GenAI through relatively weak forms of delegation: they give people access to capable models and let them use them inside existing work. An engineer asks an agent to draft a test, an analyst uses one to summarize a body of material, or a manager asks one to prepare a first version of a document. The existing worker still supplies much of the context, notices when the result is wrong, and decides what happens next. This can be highly cost-effective. The organization already employs people who possess the relevant knowledge and judgment, so useful machine intelligence can be added without first reproducing those capabilities in an engineered environment.

There is no single next step from this arrangement. An organization can invest independently in how strongly the work is modeled and how strongly its obligations govern the result. At the weak end, instructions, retrieved context, examples, and skills can make consequential knowledge available while leaving substantial interpretation to the reasoner. Stronger modeling can move recurring knowledge into structured representations that support more direct analysis. Similarly, guidance can improve the probability that an agent respects an obligation, while stronger alignment can give selected obligations authority through independent evidence, evaluation, constraints, gates, or expert judgment. Different engineering surfaces warrant different combinations; these are investment choices, not stages of maturity.

Human expertise is the general-purpose strong fallback. A capable expert can reconstruct context that was never adequately modeled, interpret obligations that remain informal or qualitative, inspect evidence, resolve exceptions, and reject work for reasons the surrounding machinery cannot express. Weak modeling and alignment can therefore work extremely

well while knowledgeable people remain around the agent: those people supply much of what the engineered environment does not.

The risk appears when organizations mistake machine productivity for independence from that human environment. The measured productivity of the agent-plus-engineer system can be incorrectly attributed to the agent alone. If the people who supplied implicit knowledge, review, exception handling, and accountability are removed, those functions do not disappear with them. Productivity gains may make a smaller engineering organization economically attractive, but reducing human capacity before replacing the functions it supplied can remove part of the environment on which the apparent productivity gain depended. An organization that intends to reduce its dependence on direct human involvement must first determine which of those functions the delegated work still requires and where they will reside. Some can move into models, evidence, validators, constraints, and procedures; others may continue to require expert judgment. As machine realization becomes cheaper and more abundant, the economic opportunity is to move recurring reconstruction and adjudication into reusable engineering structures—not to assume that productive agents have made those functions unnecessary.

A more consequential transition occurs when the organization delegates a unit of work. An agent may implement an issue, investigate an incident, reconcile records, remediate a document, or operate a service with substantial freedom over how to achieve the requested outcome. The person no longer supplies every intermediate judgment, so information and controls that previously lived in the worker's head or immediate workflow must move elsewhere. The organization needs to determine what the agent may inspect and change, what properties its work must preserve, what evidence will establish those properties, and where authority over consequential outcomes remains. Better models make larger units of work technically possible to delegate; they do not answer these organizational questions.

MAGE provides one way to reason about the environment that grows around such delegation. Modeling makes consequential knowledge and intent available without requiring each reasoner to reconstruct them from code, documents, conversations, and organizational memory. Alignment gives selected obligations authority through the mechanisms appropriate to them: a permission boundary may prevent an action, a validator may reject an artifact, instrumentation may produce evidence for later evaluation, and an expert may decide a property for which no adequate evaluator exists. As experience reveals missing knowledge or weak controls, governance conversion changes that environment so that later work inherits what earlier work learned. GenAI adoption then becomes more than repeated use of a capable model: the organization gradually becomes able to delegate larger or more consequential work because it has engineered the conditions under which that work occurs.

G.1 From Assistance to Delegation

Individual assistance is a useful place to begin because it exposes where GenAI actually helps. People discover which tasks are easy to describe, which require extensive context, which outputs are easy to check, and which mistakes recur. Little organizational machinery is required because the user remains inside the loop. If an assistant misunderstands a requirement, the engineer corrects it; if it proposes an implausible design, the engineer rejects it; if it needs a fact about the system, the engineer supplies the fact. Much of the engineering environment remains implicit in the person operating the tool.

Delegation changes that arrangement. The relevant unit is no longer a prompt but a work unit with an intended outcome, available context, permitted actions, acceptance conditions, and a boundary beyond which the agent may not proceed on its own. A useful first organizational step is therefore to identify work that already has a reasonably clear boundary. A maintenance issue with an executable regression test is easier to delegate than an open-ended product decision. A document transformation with explicit preservation requirements is easier to delegate than deciding whether the resulting document communicates effectively to its intended audience. The distinction is not whether the task is difficult. It is whether the organization can state enough about the work and its acceptance to govern the delegation.

This suggests expanding delegation along engineering surfaces rather than assigning an autonomy level to an entire organization or product. One service may have strong interface contracts, architecture rules, deployment checks, and operational evidence while another remains poorly understood. Within the same task, dependency policy may be mechanically governable while an architectural tradeoff remains judgmental. The appropriate boundary therefore depends on the particular obligations at issue: their consequence, recurrence, representability, available evidence, and the reliability with which the environment can evaluate them. Part VI developed this as an economic and governability question rather than a maturity ladder. Figure G.1-1 shows the transition as a moving boundary, not a ladder of levels.

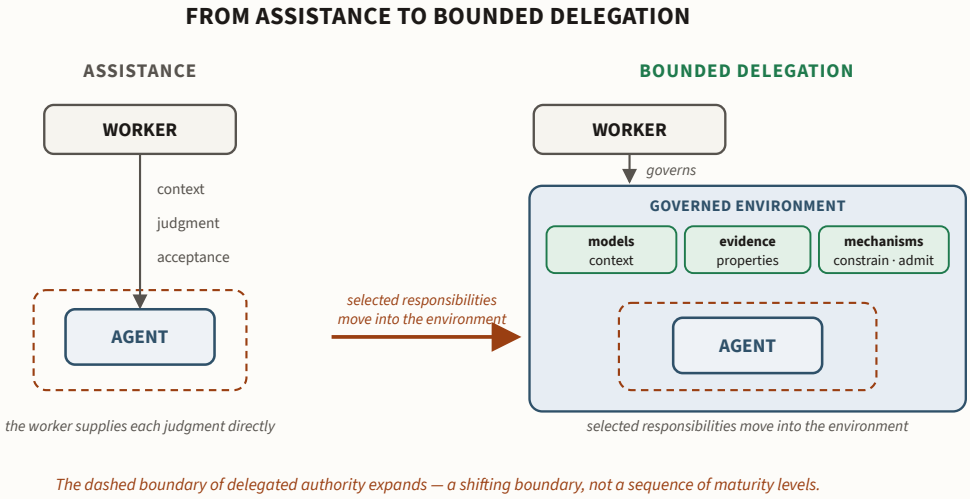


Figure G.1-1: From assistance to bounded delegation. *Under assistance, the worker supplies context, judgment, and acceptance directly around the agent. As work is delegated, selected responsibilities move into the environment — models carry context, evidence establishes properties, and mechanisms constrain and admit the result — and the boundary of what the agent may do on its own expands. The change is a shifting boundary, not a sequence of maturity levels.*

Early delegation should consequently favor work with short feedback loops and independent evidence. Tests, static analysis, schema validation, differential comparison, build results, simulations, and operational measurements can establish useful properties without asking the producing agent whether its own work is correct. The point is not to eliminate human review. It is to learn which parts of review repeatedly establish the same facts and which parts require judgment that the environment does not yet adequately support. That distinction supplies the raw material for the next stage of adoption.

G.2 From Context Engineering to a Governed Environment

As delegation expands, organizational knowledge becomes part of the engineering problem. A person embedded in a system may know which service owns a capability, which dependency direction is intentional, which migration is underway, which test is authoritative, and which apparent inconsistency is deliberate. An agent arriving at a work unit does not share that history. It can often reconstruct some of it from source and documents, but repeated reconstruction consumes reasoning capacity and can produce different interpretations from one task to the next. The same problem appears between people, although persistent teams and organizational memory have traditionally hidden some of its cost.

A natural early response is context engineering: improve what the reasoner can see when it makes a decision. Organizations can collect instructions, conventions, architecture documents, service catalogs, prior decisions, examples, skills, and other durable knowledge, then retrieve the relevant subset for each task. Graph engineering strengthens this approach by connecting entities and relationships so that retrieval can follow the structure of the system rather than depend only on textual similarity. These are important engineering improvements. They move knowledge out of individual memory and transient conversations into organizational assets that later reasoners can reuse.

In MAGE terms, however, much of this remains a relatively weak form of representation. A natural-language description of a dependency rule makes the rule explicit, but the producing agent must still interpret the description, determine how it applies to the current change, and realize an acceptable result. A graph can improve this substantially by identifying the relevant services, owners, dependencies, policies, or prior decisions. But if the meaning of its nodes and edges ultimately arrives as prose for an agent to interpret, the final engineering judgment still rests largely with the reasoner.

That is not a defect. Weak representations can produce large gains at low cost. Better context can reduce search, prevent repeated reconstruction, improve consistency, and raise the probability that an agent makes a satisfactory decision. For many engineering concerns, that may be enough. The mistake is to confuse making knowledge available to a reasoner with making the represented property independently analyzable.

Modeling strengthens the representation when the recurring engineering question warrants the investment. A service catalog can become a typed dependency model; prose about ownership can become explicit ownership and transfer relations; operational experience can become a measurement model with a declared bound; a workflow description can become a state machine. The relevant progression is not from “bad context” to “good model.” It is from representations that require more interpretation to representations whose semantics let additional questions be answered directly by people, agents, or tools.

The same distinction applies to control. Natural-language instructions, examples, policies, and skills can strongly influence autonomous work. They can make an obligation explicit and improve the probability that the producing agent respects it. But the producing agent is still being trusted to interpret the obligation, apply it correctly, and often assess its own

realization. Part IV called this guidance. Authority begins when the result no longer depends entirely on that cooperation: a constraint can make an action unavailable, a sensor can produce independent evidence, a validator can evaluate that evidence, or a gate can prevent unacceptable work from advancing.

Context engineering and graph engineering therefore occupy an important region of the MAGE design space. They can reduce reconstruction without eliminating interpretation, and improve realization without independently deciding whether an obligation was satisfied. This often makes them economical early investments in organizational adoption. They are also natural substrates from which stronger models and controls can later grow.

Figure 0.3-1 named the two recurring costs as reconstruction and repeated adjudication. Organizational adoption makes them operational: context and graph engineering can reduce the first cheaply, while stronger models and authority become attractive where the residual cost justifies further investment.

The reason to strengthen them is churn. If agents repeatedly retrieve the same prose and reach different architectural conclusions, the organization is still paying for interpretation. If reviewers repeatedly reject work for the same policy violation, the organization is still paying for adjudication. If an agent repeatedly reconstructs a relationship already implicit in a graph, the representation may not yet expose the semantics needed by the recurring engineering question. Governance conversion asks what the organization has learned from that recurrence and what future work should inherit.

The response should be proportionate. A recurring ambiguity may earn clearer guidance. Repeated reconstruction may earn a structured representation. A stable relationship may earn a typed model. A recurring judgment may earn an evaluator. A sufficiently stable and evaluable obligation may earn a constraint or admission gate. Each move requires construction and carrying cost. The reason to make it is that later work may inherit what earlier work established. Where the avoided reconstruction, adjudication, rework, or risk exceeds that cost, the asset becomes engineering capital and its return compounds across subsequent work.

This is how engineering capital compounds. The organization need not begin with rich executable models or comprehensive governance. It can begin cheaply, observe where probabilistic reasoning performs well enough, and invest more deeply where recurring cost or consequence justifies it. Stronger models are warranted not because structured representations are inherently more mature, but because they can retire work that weaker representations leave to repeated interpretation. Stronger authority is warranted not because every rule should become mechanical, but because some obligations are too stable, recurrent, or consequential to leave to repeated voluntary compliance.

The resulting pattern is to start with the cheapest adequate environment, observe the residual churn, and strengthen representation or authority independently where the expected return justifies it. This is not a maturity ladder. Different obligations can stop at different points. A natural-language skill may remain the right engineering asset for a contextual procedure. A service graph may need only to support navigation. Another graph may warrant

typed relationships because architecture analysis depends on them. A security boundary may justify structural prevention from the beginning. Organizational adoption consists of making these investments selectively, surface by surface.

An explicit representation can also serve several consumers at once. A service graph can help a reasoner locate relevant code, let an architect inspect a proposed dependency, and, where its semantics are sufficiently strong, let a validator reject an edge the organization has declared forbidden. A provenance record can explain an automated change to a reviewer and also support a mechanical check that every change was accounted for. The representation becomes organizational engineering capital when later work can begin from what earlier work established rather than reconstructing it.

Representation alone, however, does not make a rule binding. If a dependency policy matters, the organization must decide what happens when proposed work violates it. Some obligations can be made impossible to violate through interfaces, permissions, schemas, or other constraints. Some can be evaluated before work is admitted. Others become observable only during execution or after an artifact has been assembled. Alignment chooses mechanisms appropriate to those boundaries and gives selected obligations authority over the work. A rule that exists only in guidance remains advice; a rule connected to a mechanism can constrain what the organization accepts or permits.

This does not imply that every important obligation should become a deterministic check. Part VI used historic preservation and accessibility regulation to make the contrary point: an obligation can be consequential, explicit, and legally binding without admitting a complete formal evaluator. An organization may be able to enforce file-format validity mechanically while requiring an expert to judge whether a transformed document remains useful; it may verify structural properties of a design while retaining expert review of contextual fit. The appropriate response is to mechanize the parts for which adequate evaluators exist and improve the representations, evidence, procedures, and review available for the rest.

This matters for adoption because consequence and mechanizability are different dimensions. A highly consequential judgment may deserve substantial investment precisely because it cannot be handed to a cheap checker. The organization can make the governing obligation explicit, gather better evidence, provide precedent and structured criteria, preserve the rationale for earlier decisions, and require an appropriately capable expert to decide the remaining question. As models, measurements, and artificial reasoners improve, that boundary can move. Expert here names the reasoner capable of making the judgment to the standard required by its consequences; it does not require that the reasoner remain human. What matters is that the authority delegated to the reasoner does not exceed the organization's basis for trusting the judgment. The value-versus-governability tradeoff developed in Section 6.3 applies directly. Choose where to invest surface by surface. Movement toward stronger representation or authority is justified by the engineering return, not by a general notion of organizational maturity.

Figure G.2-1 maps where that investment can go.

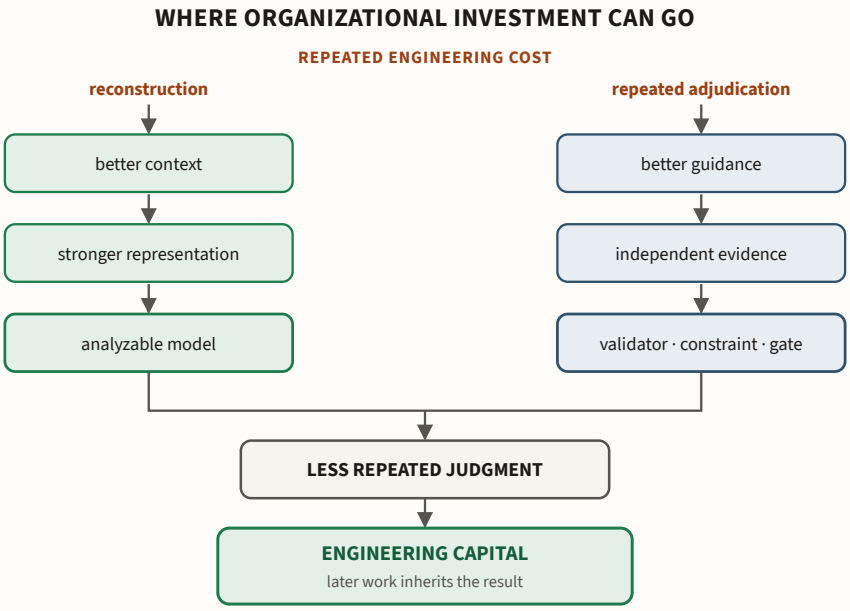


Figure G.2-1: From recurring cost to engineering capital. *Context and guidance can cheaply reduce reconstruction and improve probabilistic judgment. Where residual churn or consequence warrants further investment, organizations can strengthen representation, evidence, evaluation, or authority so that later work inherits more of what earlier work established. The paths are investment choices, not maturity levels.*

This appendix also operationalizes four figures the book has already developed; Figure G.2-2 collects them.

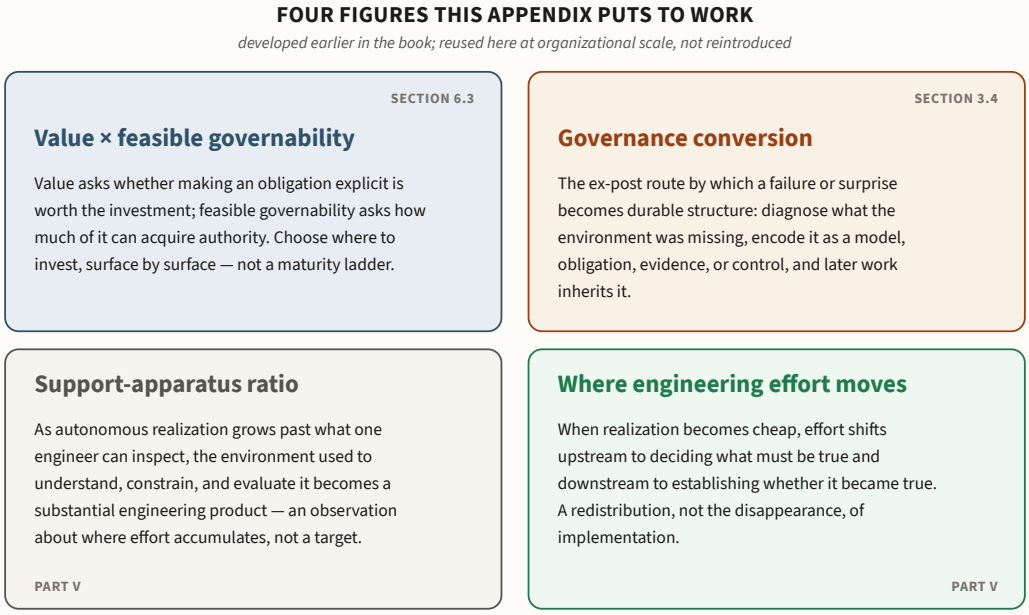


Figure G.2-2: Four figures this appendix puts to work. *Each was developed earlier in the book and is reused here at organizational scale, not reintroduced: the value of explicit engineering against feasible governability (Section 6.3), the governance-conversion loop (Section 3.4), the support-apparatus ratio, and the redistribution of engineering effort (both in Part V). The appendix refers back to these rather than re-deriving them.*

The result is a Governed Engineering Environment rather than merely an inventory of AI tools. The GEE contains the representations through which work is understood, the constraints within which it may proceed, the evidence by which results are evaluated, and the mechanisms that determine what happens next. Agents operate inside that environment alongside people and conventional software. The organization's adoption problem is then less dependent on the behavior of any particular model: a more capable model may enlarge what can be attempted, while the environment determines what may be accepted.

G.3 Learning from Delegated Work

No initial environment will capture everything that matters. Early agents will misunderstand local conventions, cross boundaries that were obvious to experienced engineers, satisfy tests while violating an unstated requirement, or produce results that expose weaknesses in the tests themselves. Human reviewers will continue to catch important problems. These interventions are useful not only because they repair the immediate work, but because they reveal where the environment failed to carry knowledge or authority that the work required.

The first response to a failure is often local: correct the artifact and continue. Sometimes that is enough. A one-off misunderstanding does not justify a new model, validator, or organizational rule. Repeated failures deserve a different diagnosis. If several work units reconstruct the same architectural fact incorrectly, the problem may be missing representation. If an obligation is repeatedly stated but violated, the environment may lack authority. If a validator reports success while reviewers consistently reject the result, the evaluator may not measure the property the organization actually cares about. If an agent cannot determine which rule applies, the relevant knowledge may exist but be unavailable at the point of reasoning.

Governance conversion is the process of turning such recurring discoveries into changes to the engineered environment. A correction may become a model entry, invariant, schema constraint, test, validator, permission rule, measurement, operating procedure, or reusable skill. The choice depends on the failure. The objective is not to mechanize every review comment, but to stop paying repeatedly for engineering knowledge that the organization has already acquired. When the same judgment is likely to recur and can be represented usefully, later work should inherit it. This is the governance-conversion loop developed in Section 3.4 (Figure G.2-2), read here at organizational scale: delegated work produces evidence and failures, diagnosis identifies what the environment was missing, a representation or authority change encodes it, and later delegated work inherits the result.

This is where adoption begins to compound. A conventional productivity rollout asks whether the next model, prompt, or tool lets a worker complete today's task faster. Governance conversion asks whether today's work leaves the environment better prepared for tomorrow's. The distinction is important because model capability is externally supplied and will continue to change. An organization that depends mainly on prompting skill must repeatedly reconstruct its local practices around each new tool. An organization that has encoded consequential knowledge, evidence, and authority into durable engineering structures can expose those structures to new reasoners as they arrive.

The resulting assets also have different lifetimes. A test tied closely to one implementation may depreciate during a rewrite. An architectural boundary, ownership rule, accessibility obligation, provenance requirement, or deployment invariant may survive several implementations. Some assets can be shared across a product; others remain local to one component or workflow. Adoption therefore creates an ordinary maintenance problem: engineering capital has carrying cost, can become stale, and must remain connected to the reality it claims to describe. More machinery is not automatically better. Models that

no longer answer useful questions and controls whose expected benefit no longer exceeds their cost should be revised or retired.

The DocAble case illustrates the scale this supporting environment can reach without establishing a general target. At the final measured snapshot, approximately 491,000 lines of production source were accompanied by about 1.50 million lines classified as support apparatus, a ratio of 3.06 to 1; the ratio had been higher during the preceding hardening period. Those counts include models, tests, validation, control machinery, and related support code, and are tabulated in the evidence ledger (Appendix I). They describe one repository under unusually intensive agentic development, not an optimal ratio for other organizations. The useful observation is structural: as autonomous realization expanded beyond what one engineer could inspect conventionally, substantial engineering effort accumulated in the environment used to understand, constrain, and evaluate that realization. Part V records the same shift in where engineering judgment went.

G.4 Expanding Delegation

Direct human review can compensate for weak modeling and alignment while knowledgeable reviewers remain available and the volume of delegated work remains manageable. A person can inspect an agent's output, reconstruct the relevant intent, notice an unexpected choice, and request a correction. That approach scales poorly when review demand grows with cheap machine realization: the cost of review continues to grow with the amount of realized work. Oversight therefore moves toward the models, evidence, validators, constraints, and gates that govern many realizations at once. Experts still review consequential work, but more of their attention can go to whether the right obligations were chosen, whether the representations remain faithful, whether the evidence is sufficient, and whether an exception is acceptable. The object of review moves; accountability does not. Part VII develops this change in sign-off as a professional consequence of agentic engineering.

This shift changes what limits scale. If every generated change requires line-by-line inspection by the person who requested it, autonomous implementation can increase output without proportionally increasing accepted engineering work. The relevant quantity is not simply how much agents produce, but how much expert attention is required to understand and govern that production. Better representations can reduce repeated reconstruction; independent evidence can settle routine questions; constraints can remove invalid choices before review; validators and gates can reject covered failures without consuming reviewer attention. The remaining review can then concentrate on obligations for which judgment still adds value. The support-apparatus ratio recorded in Part V (and recapped in Figure G.2-2) is the visible trace of this: as delegated realization grows, support apparatus can become a substantial engineering product in its own right — which is an observation about where effort accumulates, not a claim that a larger ratio is better.

This also changes where engineering effort is likely to move. When realization is expensive, substantial effort naturally accumulates in implementation. When realization becomes cheaper, the economics favor greater investment upstream in deciding what should be true and downstream in establishing whether it became true. Requirements, architecture, models, constraints, evaluation, evidence, operations, and the mechanisms that connect them become relatively more important. Implementation does not disappear, and people may still write code where doing so is useful. The organizational change is that implementation effort no longer provides a good proxy for engineering effort. Part V traces this as a redistribution of engineering effort, not the disappearance of implementation.

The boundary of safe delegation can then expand as the environment improves. A work unit that initially required an engineer to reconstruct several facts manually may become delegable once those facts are represented. A migration that once required exhaustive inspection may become easier to govern once architectural dependencies and differential evidence are available. A recurring qualitative review may acquire a structured rubric and better evidence even if it remains judgmental. Engineering capital can therefore move the delegation boundary by reducing the uncertainty or review cost attached to later work.

That boundary should not be confused with model capability. An agent may be technically capable of taking an action that the organization has no adequate basis for permitting autonomously. Conversely, a modest model can sometimes receive substantial delegated authority inside a tightly constrained environment because its possible actions are bounded and independently checked. The useful unit of adoption is consequently capability within an engineered boundary, not the advertised capability of the underlying model.

This is also why organizations should expect adoption to remain uneven. Security boundaries may justify strong authority early because their consequences are high and many relevant properties are mechanically evaluable. Product discovery may remain much more judgmental. A mature service may accumulate enough architecture and operational evidence for substantial autonomous maintenance while a rapidly changing prototype does not justify comparable machinery. Scientific software may govern provenance and reproducibility strongly while leaving scientific validity to domain experts. The book's theory predicts these mixed profiles rather than convergence toward one universal autonomy level, as Part VI argued.

G.5 Organizational Adoption

An organization does not need to adopt MAGE across an entire product before receiving value from it. The natural unit is a recurring engineering surface where delegated work is already useful or likely to become useful: dependency changes, test construction, accessibility remediation, deployment, incident investigation, API evolution, data migration, security policy, or another bounded concern. Starting from a real surface keeps the engineering proportional to an observed problem and provides evidence about which representations and controls are worth carrying.

Local environments can later compose. A dependency model built for one service may become part of a product-wide architecture model. A validator created after one incident may protect every service that shares the same obligation. A provenance convention developed for one autonomous workflow may become the organization's normal record for machine-generated changes. The product GEE emerges from these useful local structures when their scope genuinely crosses organizational boundaries; it should not be designed in advance as a comprehensive central model of everything.

This incremental route also reduces the risk of confusing standardization with governance. A central AI platform can provide model access, authentication, logging, cost controls, approved tools, and common execution infrastructure. Those are useful platform capabilities, but they do not by themselves encode what a particular engineering system means or what properties its work must preserve. The teams closest to a system still have to identify its consequential obligations and representations. Central infrastructure can make those artifacts easier to build, discover, evaluate, and enforce without pretending that one platform team can specify every domain.

Organizational roles may change accordingly. Domain engineers remain important because they know which distinctions matter and which tradeoffs are acceptable. Platform engineers can provide common machinery for context retrieval, agent execution, evidence collection, validation, permissions, provenance, and policy enforcement. Security, safety, compliance, accessibility, and other cross-cutting groups can express obligations in forms that product environments can consume rather than relying exclusively on prose guidance and episodic review. Engineering leadership decides which work may be delegated, what evidence is required for consequential decisions, and where responsibility remains. The division of labor is less about who writes a particular piece of implementation and more about who establishes the knowledge, authority, and evidence under which implementation is accepted.

That change does not remove accountability. Part VII argues that professional authority rests on competence and responsibility, not on personally performing every underlying action. An organization may therefore delegate substantial realization while retaining identifiable people and institutions responsible for the systems produced. Where consequential authority is delegated to an artificial reasoner, the organization still needs a defensible account of why that delegation was appropriate, what bounded the decision, what evidence supported it, and who had authority to establish those terms. A machine can participate

in expert judgment as its capabilities improve; it does not thereby become the terminus of organizational responsibility.

Resilience introduces a separate consideration. An organization that moves substantial engineering capability into external machine intelligence acquires a dependency on that intelligence. For ordinary work, the productivity gain may easily justify the dependency. For critical engineering functions, the organization should also ask what happens if the relevant service becomes unavailable, materially degrades, changes terms, becomes untrustworthy for the task, or cannot legally or operationally be used. The appropriate response need not be to preserve the old workflow unchanged. It may be to retain enough human competence, alternative tooling, models, documentation, and operating procedures to recover essential engineering capability when the preferred intelligence is unavailable.

The same issue applies to the diversity of judgment inside engineering organizations. Smaller teams supported by autonomous realization may incur less coordination cost, but large teams have historically supplied multiple perspectives as a side effect of having many people involved. Different engineers notice different assumptions and recognize different failure modes. A GEE preserves what has been represented and governs obligations that have been identified; it does not automatically supply perspectives that nobody brought to the problem. If fewer people can govern a larger system, organizations may need deliberate substitutes such as independent inspection, heterogeneous agents, adversarial analysis, or independently produced evidence. The product-lifecycle appendix (Appendix F) identifies this as a possible shift from coordinating enough people to build the system toward ensuring enough independent judgment over it.

G.6 A Practical Adoption Procedure

The preceding argument can be turned into a procedure. It is deliberately organized around one engineering surface at a time rather than around an enterprise-wide maturity level.

1. **Choose a recurring unit of work.** Start where GenAI can already perform useful work and where the organization can observe the result. Identify the artifact or system being changed, the outcome expected, and the boundary of the work unit.
2. **Write down the consequential obligations.** Ask what must remain true if this work is delegated. Include functional requirements, architectural boundaries, security and privacy rules, operational limits, provenance requirements, and qualitative obligations that matter to acceptance. Do not require every obligation to be mechanically decidable.
3. **Identify what the reasoner currently has to reconstruct.** Record the system facts, organizational knowledge, prior decisions, and relationships repeatedly rediscovered from source, documents, or people. Externalize the consequential recurring parts into representations appropriate to the questions being asked.
4. **Separate evidence from production.** Determine how the organization can establish important properties without relying solely on the producing agent's account of its own work. Reuse existing tests, analyses, measurements, simulations, and review procedures where they are adequate; add evidence where important claims remain unsupported.
5. **Assign authority to selected obligations.** Decide which properties should prevent an action, reject an artifact, require additional evidence, or escalate the work. Use the weakest mechanism that provides adequate assurance for the consequence at stake. Preserve expert judgment where no adequate evaluator exists rather than disguising a heuristic as a proof.
6. **Delegate within the resulting boundary.** Give the agent the work unit, relevant models, permitted actions, and acceptance path. Record enough provenance to reconstruct what happened and why. Keep escalation available for situations the environment does not adequately cover.
7. **Study interventions and failures.** When a person corrects the work, ask whether the correction is local or reveals missing organizational structure. Repeated reconstruction suggests a missing model; repeated violation suggests missing authority; repeated disagreement with a checker suggests an inadequate evaluator; repeated escalation may indicate either an appropriate judgment boundary or an opportunity for better support.
8. **Convert recurring lessons into engineering capital.** Update the representation, evidence, mechanism, procedure, or skill that would let later work inherit the lesson. Check that the new asset has a consumer and that its expected value justifies its carrying cost.
9. **Expand delegation where the evidence supports it.** Increase the scope or consequence of delegated work only where the surrounding environment provides an

adequate basis for doing so. Evaluate individual engineering surfaces rather than declaring the product or organization autonomous.

10. **Revisit responsibility and resilience as the boundary moves.** Determine who remains answerable for consequential decisions, what competence they need to exercise that responsibility, and which capabilities the organization must retain if its preferred machine intelligence is unavailable or inappropriate.

This procedure can begin with one team and one workflow. The first useful result may be modest: a dependency rule that stops being rediscovered, a test that becomes an admission gate, a provenance record that makes an automated change inspectable, or a recurring review criterion that becomes a structured rubric. The relevant measure of progress is not how many tasks have been handed to an agent. It is whether the organization can delegate useful work while retaining a sound account of what the work must satisfy, how those obligations are evaluated, and where authority over the result resides.

As capability improves, that account becomes more rather than less important. Some work that currently requires direct expert judgment will acquire better evaluators; artificial experts may become trustworthy for judgments that organizations currently reserve for people; and cheap realization will make still larger spaces of alternatives practical to explore. Other obligations will remain contextual, contested, or expensive to evaluate. Adoption therefore has no fixed endpoint at which the organization has "automated engineering." It is an ongoing allocation of reasoning and authority among people, artificial reasoners, and engineered mechanisms, revised as their relative capabilities and costs change.

The practical objective is correspondingly narrower than maximum autonomy. An organization should delegate work when doing so creates value and when the surrounding engineering provides adequate knowledge, evidence, authority, and accountability for the consequences involved. MAGE supplies structures for making those conditions explicit and for improving them through use. The result is an organization that can take advantage of increasingly cheap intelligence without requiring either that people inspect everything it produces or that the organization simply trust what the intelligence decides.

Appendix Part II — Evidence

The Empirical Record

This part collects the evidence behind the book's empirical claims. The appendices that follow separate observation from interpretation and make the evidentiary basis for the argument available for inspection.

Table 17.0-1: The evidence appendices — each beside the question that sends you to it.

If you are asking…	Go to
What was that external organization doing, and how does MAGE read it?	Appendix H — Field Guide
What measurements support that DocAble claim?	Appendix I — Evidence Ledger

Appendix H: Field Guide

Appendix H — Field Guide

Eight organizations, seven kinds

Part V (§5.5) asks whether engineering structures found in DocAble recur in systems developed independently under different technical and organizational pressures. This appendix presents the evidence behind that comparison.

These are reconstructions, not additional case studies. Public accounts reveal selected mechanisms and design decisions, not the engineering history available for DocAble. They can show that a comparable structure exists, demonstrate another way to realize it, or mark a limit on the MAGE interpretation. They cannot establish causation, and silence in a public source does not establish that an organization lacks a capability (see §6.1.5 and §5.5).

Each kind therefore gets a two-page spread, built on its clearest exemplar. The verso stays close to the public record: what the organization says it built, the evidence available for that account, and the boundary of what those sources establish. The recto interprets the same evidence through MAGE: Modeling, Alignment, the Governed Engineering Environment, governance conversion, and engineering capital.

Read the diagrams as mappings, not ratings. A strongly marked relation is directly supported by the cited account; a partial relation is a MAGE interpretation. An unmarked region means only that the source does not establish it. No card claims that an organization implements MAGE as a whole.

The organizations enter from different directions: Cloudflare from institutional policy; Spotify from fleet-scale change; Shopify from organizational knowledge; Docker from delegated runtime authority; Siemens from model-first engineering; Zenseact from distributed autonomous work; and Uber from factory-scale economics. GitLab enters that last direction from the opposite motive—trust rather than cost—so it shares Uber's spread as the sibling account of the same kind. The question is which engineering moves survive those differences.

How to read the recto diagrams

Every recto uses a simplified Figure 0.1-1: an observed engineering pressure enters from the left and passes through the mechanisms supported by the evidence to the part of MAGE it most directly illuminates.

The overlay uses exactly three epistemic states:

Mark	Meaning
▇▇▇▇ solid	directly supported by the public account
• ▸ – dashed	MAGE interpretation / partial support
▒▒▒▒ pale	part of MAGE, but not established by this source

The diagrams are neither MAGE audits nor a maturity ordering. Pale means not established, not absent.

Cloudflare
Internet infrastructure · policy-first governance at organizational scale

Verso — Evidence

What the public record shows. Cloudflare describes an engineering-standards system in which requirements originating in RFCs are extracted into structured, machine-readable form. Requirements that can be checked mechanically can become custom checks; obligations requiring semantic judgment remain review questions. This separates the authority to establish policy from the machinery that applies it repeatedly.[62]

The important move is not AI review itself, but turning policy into an artifact that later work inherits.

Boundary of the evidence. The public account is much stronger on representation and enforcement of obligations than on explicit executable models of the software being governed. Nor does it show repeated failures being converted into new environmental structure over time. The evidence establishes policy → representation → mechanism, but not failure → diagnosis → environmental adaptation.

Portable lesson. A policy decision gains multiplicative reach when the environment can apply it without requiring another person to reconstruct it.

Recto — MAGE Interpretation

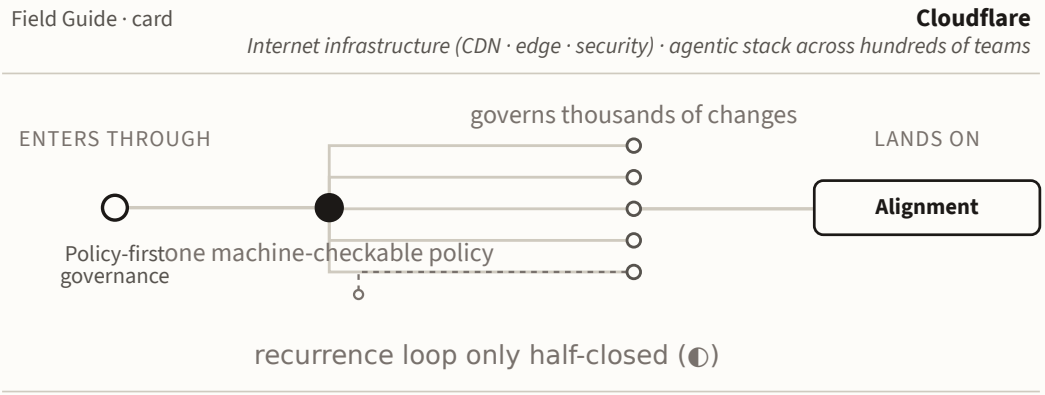


Figure H-1: Cloudflare projected onto MAGE. *Public evidence strongly supports structured policy, machine-checkable enforcement, and organizational-scale reuse; the governance-conversion loop stays interpretive.*

MAGE reading. Cloudflare is strong evidence for Alignment, with a modest Modeling move: obligations that once lived in RFC prose acquire structured identities that machinery can consume. Once represented mechanically, one policy decision can govern many later changes.

Interpretive boundary. This is an Alignment-heavy example with a modest Modeling component, not evidence for the complete MAGE loop.

Spotify
Audio streaming · fleet-scale autonomous migration

Verso — Evidence

What the public record shows. Spotify's Honk system moves large-scale code migration from distributed manual implementation toward centrally scoped, fleet-executed work. A migration that could previously involve hundreds of teams over weeks can instead be scoped by one engineer over a few days; tooling identifies and schedules targets, agents execute changes concurrently, and the engineer supervises the fleet and handles exceptions.[63]

Spotify's surrounding engineering estate also matters. Its System Model and Backstage catalog carry service identities, ownership, dependencies, endpoints, and lineage, giving fleet tooling a representation through which migrations can be targeted rather than requiring each agent to rediscover the repository landscape.[40]

[40] Spotify System Model and Backstage service catalog, as recorded in the §5.5 reconstruction (service identities, ownership, dependencies, endpoints, lineage).

Boundary of the evidence. Reported activity increases — including the 76% PR-frequency figure[41] — measure implementation activity, not durable throughput. The stronger evidence is how the work changes: agents implement while human judgment moves upstream to scoping and supervision.

Portable lesson. Scale implementation; concentrate human judgment on what to change and where.

Recto — MAGE Interpretation

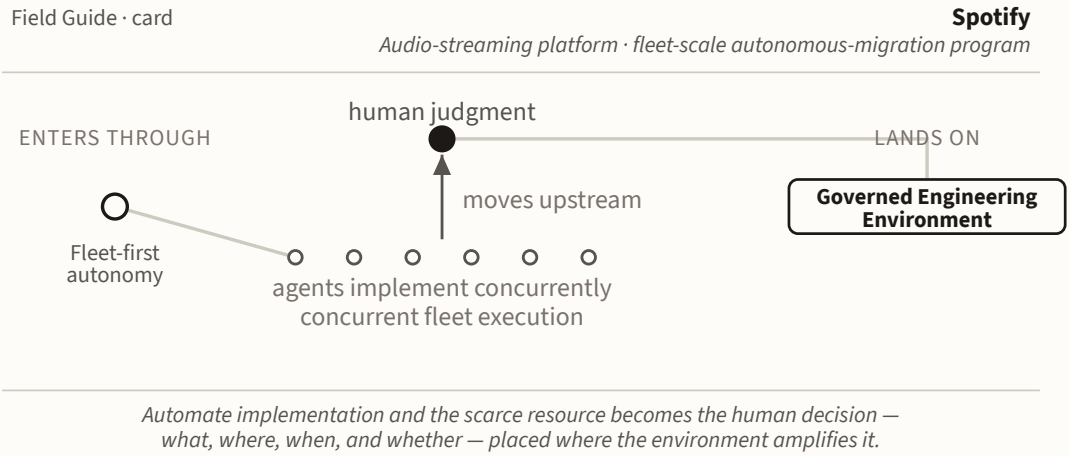


Figure H-2: Spotify projected onto MAGE. *Persistent estate representation feeds targeting, concurrent execution, and fleet-level supervision; the reading is judgment moving upstream, not the PR count.*

MAGE reading. Spotify shows the model and governed environment working together. A model of the service estate helps determine where work belongs; verification and admission machinery determine whether generated changes can proceed; fleet execution makes one engineer's upstream decisions consequential across many repositories.

Interpretive boundary. The evidence does not show that every Spotify migration is model-driven or automatically admitted; the reading applies only to the mechanisms described publicly.

Shopify

Commerce · shared organizational context for agent work

Verso — Evidence

What the public record shows. Shopify's public account describes agent work occurring in shared, observable channels rather than disappearing into private sessions. The record gives a one-month snapshot of **59,918 sessions across 5,170 channels involving more**

[41] 76% PR-frequency figure as reported in the Honk account; cited here strictly as an activity measure, not as evidence of durable throughput.

than 7,000 people, together with the characterization of the system as "multiplayer by construction."[120] Shared sessions become searchable organizational artifacts; repeated knowledge can then be extracted into skills, conventions, and defaults that later agents can retrieve. This is a soft form of externalization: retrievable prose and skills rather than a structured executable system model.

Boundary of the evidence. The evidence establishes `session → reusable knowledge`, but not `session → skill → engineering capital`. MAGE supplies the further interpretation that recurring judgment, once captured in reusable skills, may become engineering capital.

Portable lesson. Experience cannot compound organizationally while it stays private. Make work observable before trying to make its lessons reusable.

Recto — MAGE Interpretation

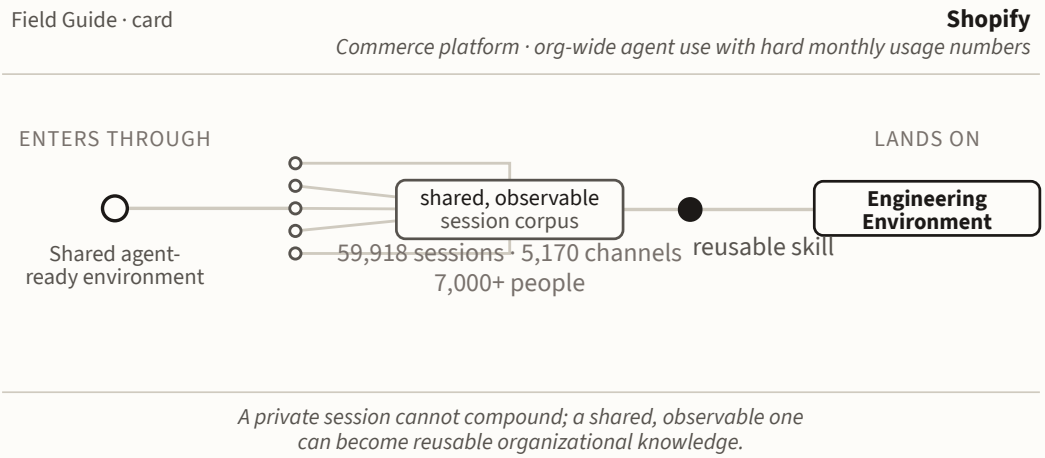


Figure H-3: Shopify projected onto MAGE. *The source supports shared externalized knowledge and reusable context; conversion into durable capital is the interpretive step, not an automatic one.*

MAGE reading. Shopify sharpens a condition on engineering capital: experience must become observable before it can become durable. Shared sessions can be searched, corrected, mined, and — where a lesson deserves reuse — converted into skills or defaults. This is Modeling at the knowledge end of the spectrum, not the executable-system-model end.

Interpretive boundary. Observability makes compounding possible; it does not make every observed session engineering capital.

Docker

Developer tooling · delegated autonomy with explicit authority boundaries

Verso — Evidence

What the public record shows. Docker's system separates delegated roles, tool permissions, implementation, and evaluation. The reconstruction identifies **seven permission-backed roles** and an **independent reviewer using a separate model invocation**, while

deterministic tests supply repeatable evidence. Final admission remains human: the system deliberately does not mechanize every consequential judgment.[121]

Docker also resists a crude reading of MAGE as "automate every gate." A human admission boundary may be exactly right where a decision stays semantic or consequential.

Boundary of the evidence. The public account covers the shortest period of the six, so it cannot establish governance conversion over time or returns on engineering capital. It does establish bounded authority, separation of generation from evaluation, deterministic evidence where appropriate, and deliberately retained human judgment.

Portable lesson. Mechanize what is adequately decidable; retaining human authority over the residue can be an engineered boundary rather than unfinished automation.

Recto — MAGE Interpretation

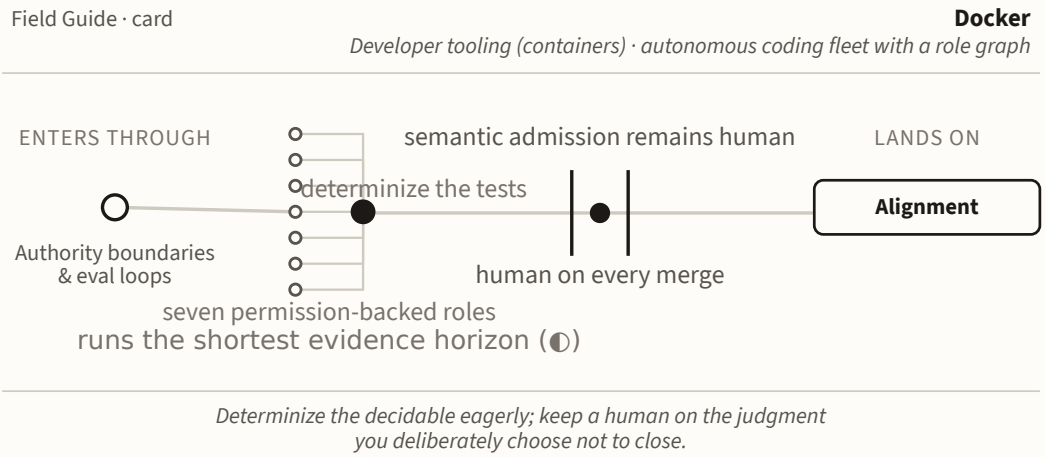


Figure H-4: Docker projected onto MAGE. *Strong direct Alignment with deliberately residual human authority; the unhighlighted Modeling and conversion regions mark source-silence, not absence.*

MAGE reading. Docker shows that strong Alignment need not depend on rich explicit system models. Permissions can bound action, tests can produce repeatable evidence, and an independent reviewer can separate generation from evaluation. Where the remaining admission decision is judgment-laden, Docker keeps a person in authority. That is not failed Alignment; it is an explicit decision to retain human judgment.

Interpretive boundary. Docker supports the Alignment reading, not governance conversion. The unhighlighted Modeling and conversion regions mean not established, not absent.

Siemens

Industrial and model-based engineering · persistent models as development surfaces

Verso — Evidence

What the public record shows. Siemens provides a comparison from outside software-first engineering. Persistent engineering representations carry requirements, behavioral structure, simulation, traceability, and other system semantics; software generation can occur downstream of those representations. The example establishes that the strong end of MAGE's Modeling is not novel: rich, executable representations are already ordinary practice in model-based engineering.[122]

SysML and CAD models, simulation, virtual ECUs, and bill-of-materials structures can serve as engineering surfaces, with code and verification downstream.[122]

Boundary of the evidence. The sources examined here do not establish a general admission mechanism that rejects changes when declared model↔implementation correspondence is violated. As throughout this appendix, source silence does not imply organizational absence.[122]

Portable lesson. Software need not be the primary surface of engineering reasoning. Mature engineering disciplines routinely place richer representations upstream of realization.

Recto — MAGE Interpretation

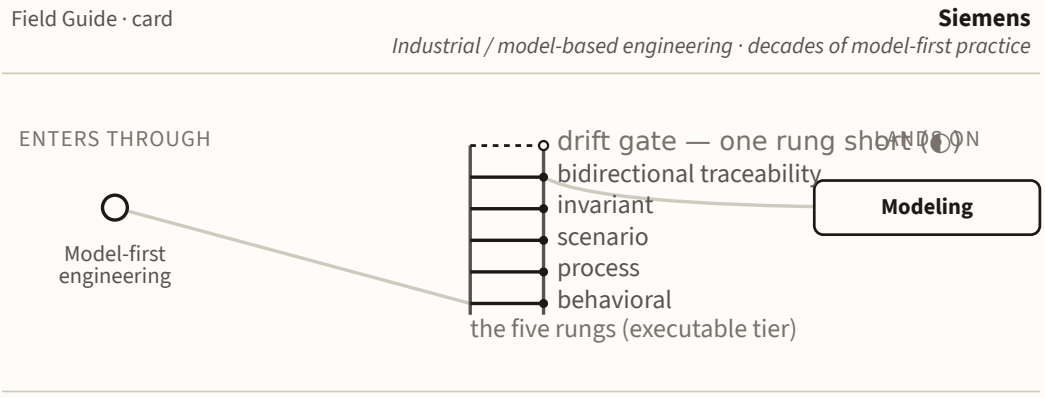


Figure H-5: Siemens projected onto MAGE. *Strong evidence for rich Modeling; evidence for analysis and verification; no claim from source silence about generalized model↔code admission.*

MAGE reading. Siemens shows the strong end of Modeling in established engineering practice: engineers reason through persistent semantic representations and treat implementation as downstream realization. MAGE did not invent model-first engineering; its contribution is to bring that engineering instinct into agentic software engineering and connect representation explicitly to the authority surfaces of the governed environment.

Interpretive boundary. The missing model↔code admission gate means not established, not absent.

Zenseact

Autonomous-driving software · centralized substrate, distributed domain ownership

Verso — Evidence

What the public record shows. Zenseact describes an organizational split. A platform team owns common substrate — authentication, sessions, execution, and safety — while domain teams retain their own agents, tools, instructions, and domain expertise. The arrangement expresses organizationally a principle seen technically elsewhere: centralize mechanisms that should be shared; leave domain judgment with the people who hold the domain knowledge.[1][2][3]

Zenseact also illustrates *selective context*. Routing machinery selects task-relevant expertise instead of loading the whole organizational corpus into every reasoning window. A central runtime and safety envelope coexist with domain expertise captured in reusable skills and surfaced on demand.[1][2][3]

Boundary of the evidence. The sources show how Zenseact divides the platform, routes context, delegates work, and assigns ownership. They do not establish the resulting productivity, safety, or returns on engineering capital.

Portable lesson. Build the common substrate once; distribute the engineering judgment to the teams that possess the knowledge it requires.

Recto — MAGE Interpretation

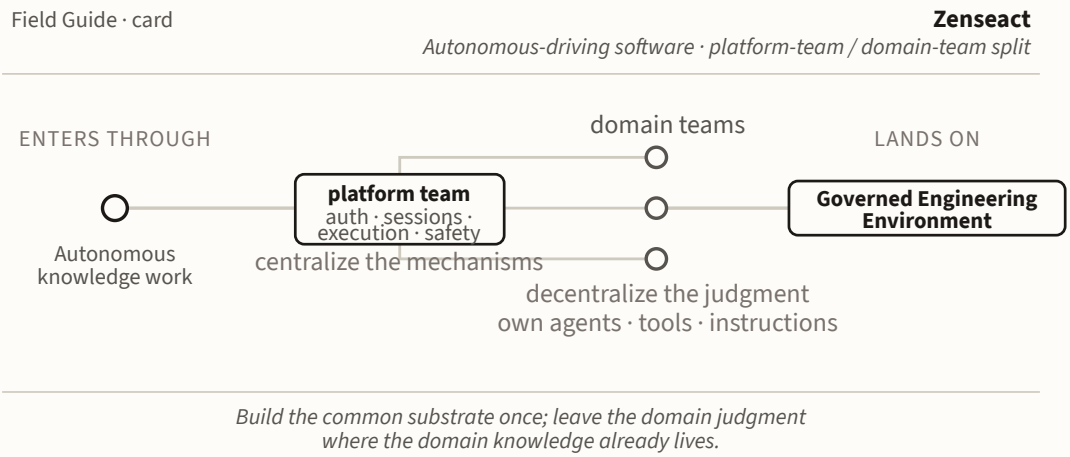


Figure H-6: Zenseact projected onto MAGE. *Shared mechanisms centralize while domain knowledge and judgment remain distributed — a concrete organizational answer to scaling both context and authority.*

MAGE reading. Zenseact combines Modeling and Alignment organizationally. Externalized domain knowledge and selective context help finite reasoners see what a task requires; shared authentication, execution, and safety machinery place common authority in one substrate. The organization does not centralize the judgment itself — domain teams keep the expertise that defines their agents and tools. The result is an organizational realization

of the Governed Engineering Environment: common mechanisms without a central knowledge bottleneck.

Interpretive boundary. The platform/domain split illustrates the Governed Engineering Environment; the evidence does not establish a closed governance-conversion loop or complete Modeling and Alignment.

Uber

Ride-hailing and logistics platform · a durable environment around a replaceable reasoner (economics motive)

Verso — Evidence

A shared kind, two motives. Two organizations build the same durable environment around a replaceable reasoner and reach it from opposite motives. GitLab starts from trust: as code gets cheap, the scarce problem becomes trusting it, so GitLab wraps the model in a durable layer of context, verification, governance, identity, and provenance, kept independent of any one model or agent so the organization's controls persist as reasoners change.[76] Uber starts from economics: with agents already writing most of its code, it treats the surrounding environment as an object of continuous measurement. Both keep the reasoner swappable and the environment permanent, and both stop at a knowledge-and-context graph rather than an executable model of behavior. Uber gives the fuller, independently measured account, so the deep treatment follows it.

What the public record shows. Uber describes AI tools operating throughout the software lifecycle: more than 70 percent of pull requests are attributed to local or cloud agents, engineers have created more than 3,600 agent skills, and those skills execute more than 30,000 times per day.[77] Managed agents perform code review, CI repair, end-to-end changes, alert triage, debugging, and maintenance. The interesting structure is not the volume of agent use but that Uber engineers the environment around the agents: real workloads become benchmarks; models are selected against cost, quality, and reliability; context and tool access are compiled down to reduce unnecessary reasoning; recurring workflows harden into reusable skills; and agents are evaluated in units such as cost per merged pull request, review, or alert.

Boundary of the evidence. The account is self-reported operational data, not a controlled comparison. It is strong on context, skills, routing, benchmarks, and cost, and much thinner on behavioral, process, or invariant models serving as the primary reasoning surface — its substrate is a knowledge-and-context graph, not an executable model of the software. GitLab's account is thinner still: a vendor architectural vision rather than a measured deployment.

Portable lesson. When code is abundant, the durable engineering object is the environment around the reasoner — the context it reads, the tools it may call, the cost it is measured against — not the reasoner itself.

Recto — MAGE Interpretation

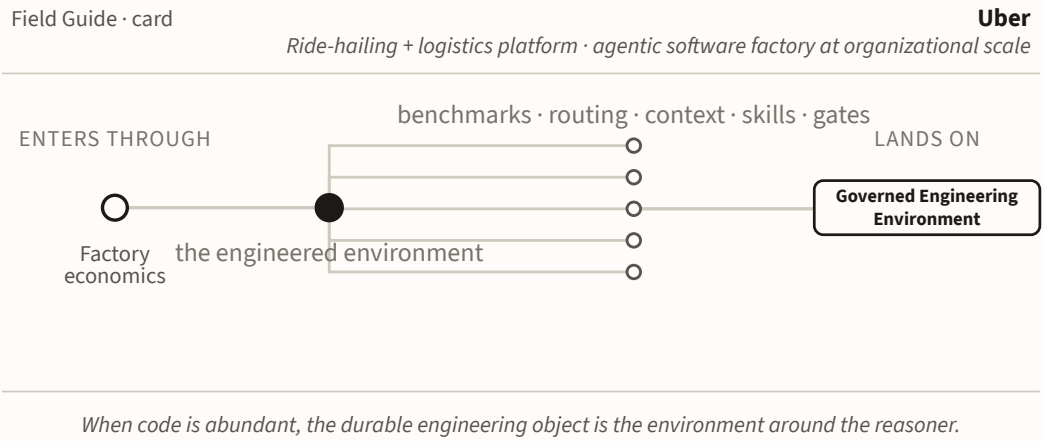


Figure H-7: Uber projected onto MAGE. *Public evidence strongly supports engineering the environment around a replaceable reasoner; the executable-model tier stays out of reach.*

MAGE reading. Uber is strong evidence that the durable engineering object is the Governed Engineering Environment rather than the model. It engineers the reasoning horizon by compiling context and tools down to what a task needs, externalizes organizational knowledge into a queryable graph, and converts recurring workflows into reusable skills — engineering capital that outlives any one model.

Interpretive boundary. MAGE reads much of Uber interpretively. The reasoning-horizon and externalized-knowledge moves are directly supported, but the account stops at a knowledge graph, with no executable behavioral model as the reasoning surface. GitLab enters the same kind from the trust motive and is cited as the sibling, not developed separately.

Appendix I: Evidence Ledger: The DocAble Case

Appendix I — Evidence Ledger: The DocAble Case

G.1 How to Read This Ledger

Part V presents the DocAble case over time. This appendix carries the quantitative and repository-level receipts behind its numerical claims.

Each section gives the measurement, its source or counting procedure, the observation window where one applies, the resulting values, and the main limit on interpretation. Unless stated otherwise, these are within-case descriptive measurements. They do not establish causal effects, comparative productivity, or portable target values.

The measurements fall into six groups: build scale and repository motion; model correspondence and drift; model coverage; representation and navigation cost; development and processing costs; and the relationship between measurement and authority.

The ledger exists for traceability. A number summarized in Part V should be recoverable here, together with the assumptions that bound it.

G.2 Build Scale and Repository Motion

G.2.1 Weekly Commit Volume

Weekly commit volume rose sharply as the agent fleet expanded, exceeded 1,000 commits per week during sustained high-volume periods, and briefly exceeded 3,000. Volume declined through the interval Part V identifies as hardening, then rose again.

During hardening, a larger share of classified commits concerned models, validation, tests, and control machinery. Commit counts do not establish why volume changed or whether productivity rose or fell.[42] Figure I-1 plots the weekly series.

[42] On some of these weeks, the author purportedly went on vacation.

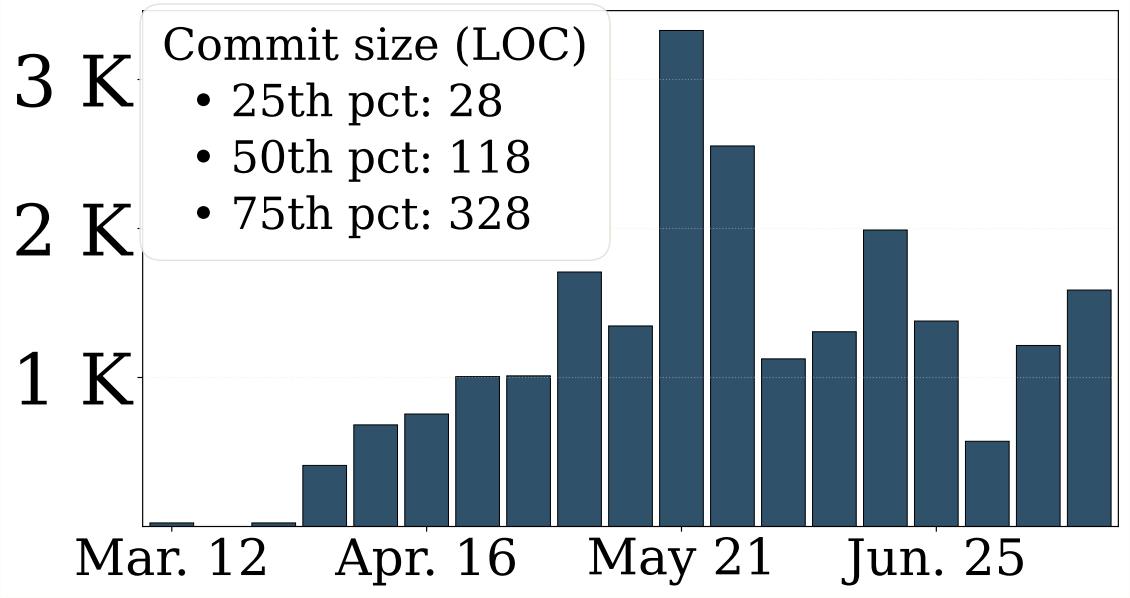


Figure I-1: Weekly Commit Volume. *Commits per week across the project history. Bar height measures repository activity, not engineering productivity; interpreting the hardening interval requires classifying the work represented by those commits.*

G.2.2 Support-Apparatus Ratio

Production and support-apparatus source were counted at four dated repository states across the seven primary source roots. The counting procedure fails if an expected root is absent. These counts are the source for the support-ratio curve in Part V.

Window	Production LoC	Support LoC	Support ratio
Prototype — Apr. 9	26,956	22,908	0.85×
Mechanization — May 31	302,844	751,050	2.48×
Hardening — Jun. 30	337,905	1,244,194	3.68×
Final snapshot — Aug. 3	491,090	1,501,907	3.06×

Support-apparatus source began below parity with production source, crossed it by the mechanization snapshot, peaked at 3.68× during hardening, and stood at 3.06× at the final snapshot.

These counts describe how source was distributed in this repository. The ratio is not a measure of engineering value, engineering-capital return, or a recommended target for another project.

G.2.3 Product-Path Line Motion

Lines added and deleted were counted over four windows for the two principal product paths:

- `web/` — the Python service and worker.

- `backend/` — the C# tool and rule engine.

Source: `git numstat` over the dated commit windows.

Window	web/ added	web/ deleted	backend/ added	backend/ deleted
Prototype	22,539	7,717	48,636	10,166
Mechanization	371,855	161,044	941,120	286,378
Hardening	179,649	33,983	109,188	3,767
Loop management	96,825	14,332	116,313	9,708

The mechanization window contains the largest add-and-delete volume, particularly in `backend/`. Deletions fall sharply in the later windows, and both paths become strongly net-additive. Figure I-2 plots these counts.

These counts are a repository-motion proxy, not the theoretical concept of churn used elsewhere in the book. Generated bundles and vendored trees are included where they occur; Part V's accounting note describes that bounded inflation.

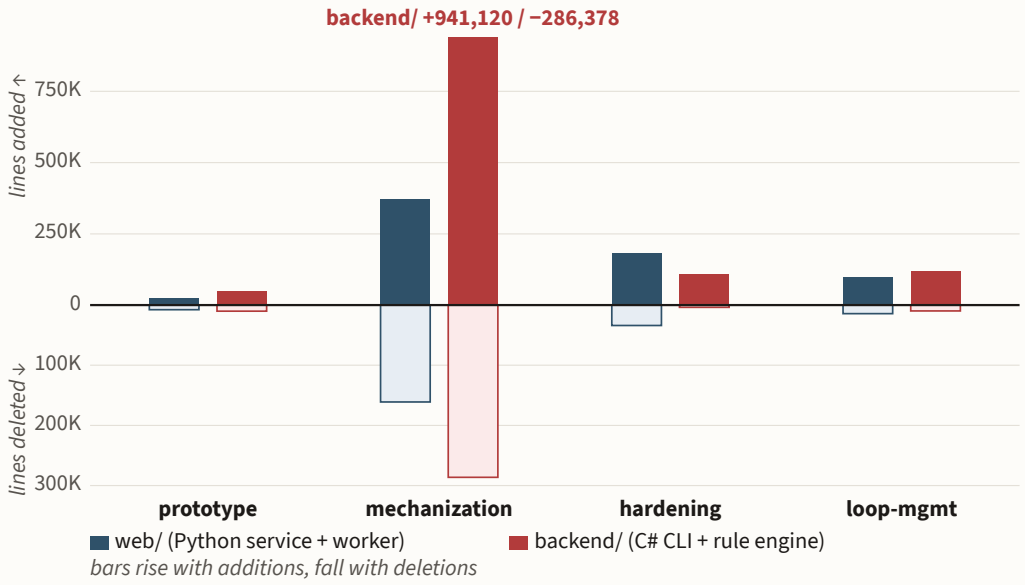


Figure I-2: Product-Path Line Motion. *Lines added above the baseline and deleted below it, by path and study window. Mechanization contains the largest observed line motion. The later reduction in deletions is consistent with less structural rewriting but does not establish its cause.*

G.2.4 Growth of Countable Controls

Project-specific lint files and gate scripts were counted at four repository states, using `git ls-tree` at the corresponding window SHA.

Window	Lint files	Gate scripts
Prototype	0	0
Mechanization	336	20
Hardening	595	76
Final snapshot	747	102

Both counted surfaces begin at zero in the prototype snapshot. At the final snapshot, the 747 lint files contain 993 registered lint specifications.

Two further repository counts show that at least some controls were created in response to observed failures:

- **Paired fix-and-lint tags** — 208 commits carry one.
- **Incident-named lints** — 27 lints name a specific dated incident in their text; spot checks confirmed that the sampled cases linked an observed failure to the resulting control.

These counts show both the growth of project-specific control machinery and a subset of cases where failures led to new controls. They do not show what fraction of all controls originated in failures rather than being designed prospectively, and raw control counts do not measure their value.

G.3 Model Correspondence and Drift

The measurements in this section concern model↔code correspondence only. Documentation drift is counted separately in G.3.5 and is excluded from the model-sync claim.

The underlying question is narrow. After explicit models became reasoning surfaces, did model↔implementation drift occur, could derived checks detect it, and did the observed class recur after close?

G.3.1–G.3.4 Correspondence Results

Claim	Measurement	Result	Limitation
Model↔code drift existed before the derived floor	Re-run each closed Epic's own lints against its closed state; classify findings by hand	Approximately 27 genuine model↔code drifts, including a production-blocking pointer drift and a fully typed function with zero consumers	Manual classification; no independently specified oracle

Claim	Measurement	Result	Limitation
Derived checks catch fresh drift	Re-run the derived floor at HEAD	6 genuine catches: three traceability failures and three stale-anchor or stale-test failures	Small N; measures only governed surfaces
Post-close recurrence of modeled, mechanically decidable drift	Census over 56 cumulative Epic closes	0 observed across 56 cumulative Epic closes	Finite observation window; does not cover semantic or unmodeled drift
Checks were exercised during continuing model-bridge change	`git numstat` over one week for the query/reactor, governance-graph, and frontend-build models	+8,970 / −173 lines across 63 commits	Line motion is a change-load proxy, not a measure of semantic difficulty

Manual review classified all approximately 27 pre-floor findings as genuine model↔code drift. Because that classification relied on human judgment rather than an independent criterion, the result establishes that drift existed before the floor, not the detector's precision.

The six HEAD catches were an unregistered model consumer, a missing component entry, a service-call-graph mismatch, and three stale-anchor or stale-test cases. A symbol-anchored drift lint, a consumer-registry-freshness check, and a service-call-graph drift lint detected them.

Taken together, these observations establish a bounded within-case sequence: model↔code drift existed before the derived checks; the checks caught six fresh instances; and the modeled, mechanically decidable class did not recur across 56 closes during the measured window. They do not establish that the class was eliminated or that the mechanism prevents model drift.

G.3.5 Documentation Drift — Excluded from the Model-Sync Claim

Stale headers and stale prose numbers are documentation drift, not model↔code drift. They are counted separately because folding them into the correspondence measurements would inflate the model-sync evidence.

Documentation drift	Refresh-window count	Detected by	Resolution
Status header frozen at a pre-close phase	11	Reading by a person or capable model	Close tooling rewrites the status atomically
Stale prose number after the implementation moved past it	9	Model re-deriving the number from code	Routed one-line fix or audit finding

These observations concern prose that no derived correspondence check parses. They therefore say nothing about whether a model remains synchronized with implementation.

G.3.6 Scope of the Model-Sync Claim

The correspondence mechanisms cover only modeled, mechanically decidable relationships, and the observed N is small. A semantic mismatch whose anchors still resolve may remain judgment-dependent, and an unmodeled region has no model-correspondence check at all.

The results are therefore field observations about one governed surface, not a general catch rate or a proof of model correctness. Three cases stay distinct:

- **Derived correspondence** — machinery can re-establish it from current artifacts.
- **Semantic correspondence** — it may still require judgment.
- **Unmodeled regions** — no model claim exists for them yet.

G.4 Model Coverage

The traceability tracer measures the fraction of exercised code with no corresponding model claim. Repeated runs identified portions of the exercised surface that stayed outside the explicit model structure. Across the recorded sequence, the unmodeled fraction fell from 56% on the first run to 7.89% on the ninth.

Run	Unmodeled exercised surface	Note
First	56%	Majority of exercised code traced to no model claim
Intermediate runs	declining	Successive model-loop Epics targeted large remaining orphan clusters
Ninth	7.89%	Remaining orphan surface was small and explicitly identified

Three qualifications matter.

- **Coverage, not quality.** The metric measures coverage, not model quality. Code may trace to a model claim that is itself incomplete or poorly chosen.
- **Operator-directed decline.** Each model-loop Epic targeted a large remaining orphan cluster. The decline reflects deliberate modeling work, not autonomous convergence.
- **Not all orphans are debt.** An unmodeled exercised symbol may be a genuinely missing model, a missing anchor on an existing model, or implementation detail below the grain the model should represent. Backward tracing from exercised symbols to expected model edges distinguished these cases; only the first two necessarily call for more modeling. The unmodeled fraction measures absence of explicit representation, not engineering deficiency: the remainder may include unknown obligations, tacit obligations, and deliberately preserved degrees of freedom. MAGE does not prescribe zero as the target.

G.5 Representation and Navigation Cost

A small exploratory pilot tested whether a model-derived navigation surface reduced the context an agent consumed while determining where to look in the repository.

Across four tasks, model-guided navigation reduced reconstruction-token cost by a median of roughly 35% relative to the from-scratch condition, with no observed loss of task-level correctness.

Measure	Baseline	Model-guided	Interpretation
Tasks	4	4	N too small for generalization
Reconstruction-token cost	Full baseline	~35% lower median	Directional evidence of reduced reconstruction effort
Recorded correctness	Reference condition	No observed loss	The existing pilot does not support a general equivalence claim

The tasks were not independently sampled, and N=4 is too small to estimate a general effect. Within this small pilot, the result is consistent with the proposed mechanism: an explicit representation may reduce how much lower-level structure an agent must reconstruct before acting.

G.6 Cost and Scale Receipts

These quantities establish orders of magnitude relevant to the Part V discussion. Their units, scopes, and cost categories differ; they are not entries in a comparative cost model.

Quantity	Observed or estimated value	Basis
Accessibility-checker findings in a representative deck	42	One graduate instructional deck evaluated with the built-in accessibility checker
Manual remediation, one teaching load	≈ $20,000 estimated faculty labor	Findings/deck × minutes/finding × decks/course × loaded hourly rate
Vendor remediation	$3–$40 / page	Market range collected during the study; not staffed for graduate-level subject annotation
Automated processing, one representative deck	≈ 1 minute; ≈ $1 direct processing cost	Warm-start service near the end of the study period
Direct development cost	≈ $60,000	Roughly 20-week study; majority salary

The manual-remediation estimate and automated-processing figure use different units and cost categories. One estimates faculty labor across a teaching load; the other records direct processing cost for one representative deck. They indicate scale but are not a controlled cost comparison.

The vendor range is a market observation, not a quality-adjusted comparison with DocAble. The development-cost figure is an order-of-magnitude direct-cost estimate, not audited project accounting.

G.7 Measurement Without Authority

A measurement can be useful before it deserves authority.

G.7.1 Provisional Cost-and-Time Model

A per-chunk worst-case cost-and-time estimate was recorded as a timestamped provisional model rather than embedded as a permanent code constant.

Current instrumentation can report when an operation would exceed the provisional budget. Production admission does not depend on that estimate, because the observation base is not yet strong enough to justify a blocking threshold.

The provisional bound need not be correct to be useful. The system keeps the stages separate: **observation → representation → reporting**. Because the evidence is insufficient, the estimate does not become a gate.

G.7.2 Cold-Start Contrast Case

The contrasting case did justify an engineering decision.

The measured request-level cold-start value was 4,057 ms. That observation fed a topology model representing the longest cold-start path and motivated an architectural change. The resulting warm floor was 109 ms. The sequence is **request-level measurement (4,057 ms) → cold-start topology model → architectural change → 109 ms warm floor**.

The contrast does not imply that every measured quantity should eventually become a gate. Authority requires evidence adequate to the decision being made. The provisional cost model remained report-only; the stable, structurally interpretable cold-start observation justified architectural action.

G.8 What This Ledger Does Not Measure

Most quantities in this appendix are activity, structural, or within-case mechanism measures:

- commits and line motion;
- source distribution;
- control counts;
- model coverage;

- correspondence catches and reopens;
- exploratory navigation cost;
- selected financial and runtime observations.

These are not direct estimates of the broader outcomes needed to compare MAGE with another engineering process: durable throughput, defect escape, human-attention burden, or total cost of ownership. The case follows one production system, one primary engineer directing an agent fleet, and one contemporary model ecosystem; it did not collect those outcomes under a controlled counterfactual. The measurements therefore establish what happened within the case and which mechanisms were exercised, not the effects another organization should expect from adopting MAGE.

The ledger's narrower purpose is traceability: the quantitative claims in Part V can be inspected independently of the surrounding argument.

Appendix J: Crazy Ideas

Appendix J — Crazy Ideas

A temporary holding area — not part of the argument

Editorial status. This appendix is a temporary holding area for research directions generated while developing the manuscript but extending beyond the claims the book needs to make. Nothing here is essential to MAGE. The notes preserve conjectures, study designs, and possible paper agendas so they can be developed separately rather than enlarging the book's argument. Remove this appendix before publication.

J.1 Commodity Intelligence as an Experimental Instrument for Software Engineering

Commodity intelligence may create an unusual empirical opportunity for software-engineering research. Studies of human engineering practice contend with substantial variation in experience, familiarity, reasoning style, and local practice. Agentic systems remain stochastic and evolve across releases, but more of the reasoning substrate can be held approximately fixed: model version, repository state, tool surface, prompt, sampling procedure, task family, and engineered environment.

Repeated trials can therefore estimate stochastic variation while one representation, mechanism, or authority policy changes. Some system-scale questions become more directly interventional: not only *do teams using practice X report better outcomes?*, but *what changes when representation A is replaced by representation B, or admission policy A by policy B, under otherwise matched conditions?*[43]

This suggests an experimental science in which the engineered environment itself becomes an independent variable. Cross reasoning engines with environments. Hold the task family fixed while changing the available models. Add or remove a validator. Vary correspondence strength. Compare thin and richly represented repositories. Measure durable throughput, reconstruction effort, defect escape, human intervention, and the distribution of engineering work.

The opportunity is broader than MAGE. MAGE supplies one theory about which environmental variables should matter; other architectures should supply competing theories. The methodological conjecture is that commodity reasoners make some previously organizationally confounded software-engineering questions experimentally tractable enough for stronger causal study.

The methodological opportunity is not merely repeatability. It is factorization. Human-subject studies of software engineering often struggle because practitioner skill, local familiarity, tool use, organizational practice, and problem-solving strategy move together. A commodity reasoner does not eliminate those confounds, but it can make more of them explicit experimental factors. Reasoning engine, environment, task, representation, tool surface, authority policy, and sampling procedure can be crossed rather than allowed to vary silently together.

That creates a research program around the environment–reasoner interaction. Does a richer environment substitute for raw model capability, allowing weaker reasoners to close the gap? Does it complement capability, so stronger reasoners benefit disproportionately from better representations and mechanisms? Are there environments that help one model

[43] The existing productivity record already runs both ways: a field experiment across thousands of developers measured a real rise in completed tasks[124], while experienced developers in familiar repositories ran measurably slower with early tooling even as they believed themselves faster[125]. The spread motivates the design: agent effects are context-dependent, so "AI use" is too coarse a variable to test on its own.

family and hinder another? At what point does environmental structure cease to reduce reasoning cost and instead become another context and maintenance burden?

The dependent variables should also move beyond task completion. A useful experiment would distinguish raw implementation activity from durable throughput, defect escape, reconstruction cost, human intervention, and the amount of work that can be delegated at matched quality. It could also measure where human effort moves. An environment that accelerates source production but increases review and recovery is different from one that removes repeated reasoning across several stages of the engineering loop.

Several study designs follow naturally. A crossed design can vary reasoning engine and engineered environment over a shared task family. A stepped intervention can add a representation, validator, or policy to one subsystem and use its earlier behavior as a baseline. Ablation can remove one environmental capability from a mature system. Longitudinal experiments can ask whether an intervention continues paying after the immediate task, which matters if the claim concerns durable engineering capital rather than one-shot performance.

The hard measurement problem is environment quality. Mechanism count, repository size, or model volume are poor proxies. Candidate dimensions include representation fidelity, obligation coverage, synchronization strength, retrieval/reconstruction cost, governance friction, and the fraction of repeated judgment displaced by durable structure. A research program should treat those as separable constructs rather than collapse them into a maturity score.

The design also creates threats that should be made explicit. Foundation models change rapidly; repeated trials are not independent if service-side behavior shifts; task suites can become contaminated; environments may be tuned to the particular reasoner used to evaluate them; and controlling the repository too tightly can make the study unlike real engineering. The methodological claim is therefore not that commodity agents create laboratory-perfect software engineering. It is that they may make stronger interventions over engineering structure possible than studies in which the reasoning substrate is an uncontrolled population of human practitioners.

Candidate research questions.

* How much productive capability is attributable to the engineered environment rather than the reasoning engine? * When do environment quality and model capability substitute for one another, and when are they complementary? * Which environmental interventions persist as durable gains rather than moving effort downstream? * How do representation fidelity, authority, and evidence independently affect agentic performance? * What experimental controls are sufficient for causal claims when the reasoner itself remains stochastic and versioned?

Possible paper seed: *Software Engineering With a Standardized Reasoner: Commodity Intelligence as an Experimental Instrument.*

J.2 Model Induction from Realized Work

Explicit models need not always precede realization. Repeated structure in realized work may supply evidence from which useful engineering concepts can be induced. The central move is not clustering for its own sake. Weak detectors or retrieval surface candidate regularities; comparison establishes whether instances share more than superficial form; engineering judgment decides whether the recurring shape denotes a concept worth naming; and the resulting vocabulary becomes available to future reasoning.

DocAble supplies a grounded software instance. Primitive-density analysis exposed repeated low-level structures. Most were legitimate implementation detail. Some recurring patterns across related components expressed concepts the system was using without naming, and those concepts became explicit types and relations.

The broader conjecture is that the same move may recur outside software. Repeated entities and relations in documents or organizational records may expose latent domain vocabularies. Repeated geometric arrangements, interfaces, and parameter bundles in CAD artifacts may expose candidate components or design concepts. In each case, realized artifacts supply candidate structure, but they do not decide which distinctions matter.

This suggests a research problem distinct from ordinary retrieval: when can an agent move from retrieving repeated instances to proposing a durable abstraction that makes future work cheaper to reason about? Useful evaluation would have to measure not merely whether an induced concept matches a latent cluster, but whether adopting the abstraction improves later reasoning, consistency, search, assurance, or reuse.

The research problem therefore has at least three separable stages. Detection asks whether realized artifacts contain recurring structure worth comparing. Abstraction asks whether several instances can be explained by a common concept rather than merely grouped by surface similarity. Adoption asks whether naming and institutionalizing that concept improves later engineering work. A system can succeed at one stage and fail at the next: a perfect cluster can still correspond to an accidental implementation convention, and a semantically valid abstraction can still cost more to maintain than it saves.

This makes model induction different from ordinary concept mining. The output is not merely a label or cluster. The induced concept may acquire fields, relations, invariants, ownership, correspondence rules, or authority. Once future work begins depending on it, the abstraction becomes part of the engineering environment and incurs the same obligations as any other model: it can drift, overfit its originating examples, suppress useful variation, or survive after the concept has ceased to matter.

The central evaluation question is downstream utility. Given the same future tasks, does an induced representation reduce reconstruction, inconsistency, or repeated judgment? Does it improve retrieval because related instances join through a shared concept? Does it permit checks or transformations that were impractical over raw artifacts? Can engineers predict which changes will be affected by modifying the model? And does the benefit persist long enough to repay the cost of establishing and maintaining the abstraction?

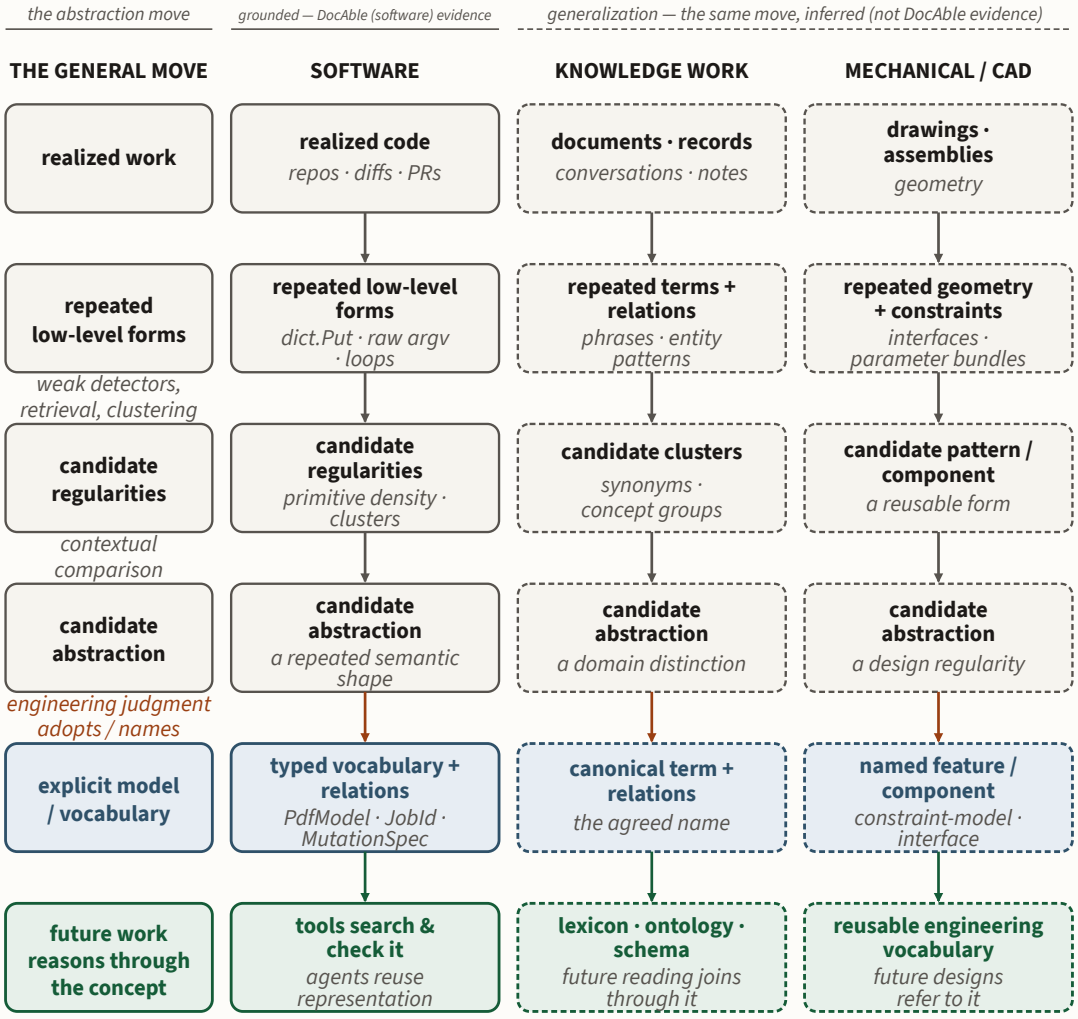
Brownfield evolution provides a particularly useful experimental setting. Mine candidate concepts from an initial history, introduce selected representations, and then evaluate later work prospectively rather than scoring the induced abstraction against the same artifacts from which it was inferred. That design distinguishes an abstraction that merely compresses yesterday's implementation from one that actually improves tomorrow's engineering.

Human judgment is not merely a fallback stage in this process; it supplies the intent test that repetition cannot. Two structures may recur because of copy-and-paste history, framework convention, or coincidence rather than because the domain contains a stable concept. Conversely, a consequential concept may have several superficially different realizations. The useful agentic system therefore needs to propose and compare candidate abstractions while exposing the evidence from which each was inferred, leaving adoption to accountable engineering judgment until stronger criteria are available.

Candidate research questions.

* Which forms of repetition predict a reusable engineering abstraction rather than accidental implementation similarity? * How should agents propose relations, invariants, and boundaries once a candidate concept has been identified? * What evidence should accompany an induced abstraction so an engineer can judge whether it deserves adoption? * How can induced models be evaluated prospectively on later work rather than retrospectively against their training artifacts? * When does introducing a model reduce degrees of freedom productively, and when does it prematurely freeze an ontology? * How should an induced model evolve, split, merge, or retire as realized work changes?

Figure J.2-1 traces that general move, from realized work to a named model, and instantiates it three ways. Its software, knowledge-work, and mechanical-CAD columns are what make this a research direction rather than a claim the book establishes.



Model induction converts repeated form into reusable engineering vocabulary.

The agent finds repetition, compares, and proposes structure;
the engineer decides which distinctions matter and names the concept.

Figure J.2-1: Model induction. *The same move runs in three settings: realized work yields repeated low-level forms, weak detection surfaces candidate regularities, and engineering judgment names the concept that survives comparison. The left rail states the move; the three columns instantiate it in software, knowledge work, and CAD. Software is the grounded case; the extension to knowledge work and CAD is a generalization, not evidence DocAble supplied.*

Possible paper seed: *From Repetition to Representation: Inducing Engineering Models From Realized Work.*

J.3 Model-Based Agentic Engineering Beyond Software

MAGE is framed as a software-engineering method because software supplies the evidence developed in this book. Its underlying arrangement may have a broader domain of application: a capable autonomous reasoner acts through explicit representations of an engineered object, operates under selected constraints and evidence requirements, and leaves consequential judgment with accountable engineers.

That arrangement is not inherently specific to source code. Mechanical design already has persistent representations of geometry, interfaces, tolerances, requirements, and physical behavior. Electrical and systems engineering have schematics, architecture models, simulations, requirements, and assurance arguments. Knowledge-intensive organizational work has weaker formal structure but often possesses recurring entities, relations, workflows, policies, and evidentiary obligations. In each case, autonomous realization raises a similar question: what environment should be engineered around the reasoner so that consequential properties remain explicit, observable, and governable?

The conjecture should not be that every discipline will converge on MAGE. Different realization substrates impose different economics and authority regimes. Physical fabrication remains costly; safety cases may be legally constrained; some domains already have mature model-centered practice; others depend primarily on social or scientific judgment that resists mechanization. The research question is whether the same architectural variables —representation, correspondence, evidence, authority, residual degrees of freedom, and durable engineering capital—predict where autonomous engineering succeeds.

Comparative work could therefore start from established engineering workflows rather than importing software machinery directly. Observe how autonomous tools operate in CAD, EDA, systems engineering, or scientific-modeling environments that already contain explicit models. Ask which representations agents can use without reconstruction, which obligations can be delegated to existing analysis machinery, which new correspondence problems autonomous work creates, and where human responsibility remains irreducible.

The strongest test would be comparative rather than rhetorical. If a common architecture exists, independently evolved agentic engineering systems should exhibit homologous structures despite different artifacts: purposeful representations, machine-readable constraints, independently produced evidence, allocated authority, and mechanisms that turn recurring judgment into reusable capacity. If those structures do not recur—or if software's unusually symbolic realization substrate is essential to the effect—the generalization should fail.

Possible paper seed: *From Agentic Software Engineering to Agentic Engineering: A Comparative Theory of Governed Autonomous Realization.*

Back Matter

Colophon — Dogfooding MAGE

[44]

In a typical colophon, I would tell you about the font. The book was set in Source Serif, with Source Sans for headings. Tradition satisfied.

This book was built and revised using some of the same engineering ideas it describes.

During editing, important properties of the manuscript were represented explicitly rather than left entirely in prose or editorial memory. Models tracked properties and obligations such as chapter identity, the argument spine, claims, terminology, evidence relationships, and book structure. Generated projections and consistency checks helped keep those representations synchronized as the manuscript changed.

The point was not to mechanize editorial judgment. It was to make consequential editorial knowledge explicit so changes could propagate coherently and decidable inconsistencies could be caught before publication.

The models and checks behind the manuscript

Each model made one aspect of the manuscript explicit, and checks kept it synchronized as the prose changed. Table 21.1-1 shows each model and the failure its checks were designed to prevent.

Table 21.1-1: The manuscript's models. *Each structured model alongside the manuscript, the facet of the argument it carried, and the failure a consistency check on it prevented.*

Model	Purpose	Prevented failure
Argument-spine model	The book's numbered claims, their order, and the chapters that advance them	A claim advanced nowhere, duplicated across chapters, or spread thinner than it is taught
Chapter-shape model	Each chapter's opening and closing graded against the editorial discipline	Structural drift — an opening that never names the failure, a closing that only restates

[44] *A colophon is the note a book keeps about its own making — traditionally set near the end, where the typeface, the press, and the hands that set the type are recorded.*

Model	Purpose	Prevented failure
Concept-and-vocabulary map	One canonical phrase for each of the book's central ideas	Terminology drift — the settled *structured model* slipping back toward *typed model*
Claims model	The load-bearing propositions, each with an audit predicate identifying prose that would contradict it	An unsupported claim, or one the later text quietly contradicts
Literature-positioning model	One record per intervention, each carrying its established-lineage → current-frontier → narrow-move frame	A citation floating loose, or the book mis-positioned against neighboring work
Substantiation ledger	Each factual number bound to the claim it backs, its source, and its limit	A claim about the world with no data and no literature behind it
Flagship-case model	The DocAble case organized into catalogue-entry packages, one row per architectural part	A case-study description drifting away from the catalogue it draws on
Design tokens	One source for the faces, the accent, and the type scale, projected to every output	Visual drift between the web edition and the print edition

Structural change, semantic change

Two edits during the final pass show what the models could do — and where they stopped.

The first was structural. During the final edit, Part VI was split into a theory Part and a professional-synthesis Part, and chapters moved between them. Their reader-facing locations changed, but their stable identities did not. Because other manuscript models referred to those identities rather than to transient filenames, the reorganization propagated without manually repairing every dependent representation.

The harder edits were semantic. During the final theory pass, the relationship between Modeling and Alignment changed. The old account made them causally dependent; the final theory treats them as independent activities addressing different problems, with Modeling as the principled route to greater semantic reach. The role of a sensor changed too: instead of deciding that a violation occurred, it observes and produces evidence for a validator. Existing references still resolved; nothing was mechanically "broken." But some declared statements now represented an obsolete theory.

A model can be structurally valid and semantically wrong.

Those changes required editorial judgment. Once the corrected meaning was encoded in the model, its consequences became easier to propagate and later drift easier to detect. But machinery could not decide what the corrected theory should be. Mechanical checks can establish that a reference resolves. They cannot establish that the idea it points to is

still right. The manuscript needed a separate model-to-prose semantic audit for exactly this reason: a join could pass and still carry stale meaning.

What the models did — and did not do

Explicit models make important properties available to machinery, and machinery can enforce relationships that are sufficiently decidable. But representation does not abolish judgment. A manuscript can satisfy every structural join and still say the wrong thing.

The models did not supply the editorial judgment. They made selected properties explicit, allowed selected relationships to be checked, and helped propagate consequential decisions once those decisions had been made.

This is a reflexive demonstration, not independent evidence for MAGE. It is a fitting final example: represent what matters, give machinery authority over what it can honestly decide, and leave semantic judgment where it belongs.

About the Author

James C. Davis is an assistant professor of software engineering in the Elmore Family School of Electrical and Computer Engineering at Purdue University. His work appears at venues such as ICSE and IEEE S&P and has been recognized with three ACM SIGSOFT Distinguished Paper Awards. He received the NSF CAREER award in 2026. His lab is supported by the NSF, Google, Rolls-Royce, Cisco, Socket, and OpenAI. He is a Senior Member of the IEEE and co-editor of the column “Practicing AI” in IEEE Computer.

Before entering academia, Davis worked as a software engineer at IBM (2012–2015), where he wrote roughly a hundred thousand lines of Perl and C/C++ test infrastructure, trained engineers at international sites, and broke his team’s record for most defects opened in a year. He earned his PhD from Virginia Tech in 2020.

Bibliography

1. Nosta, J. Intelligence as a Commodity. https://www.psychologytoday.com/us/blog/the-digital-self/202603/intelligence-as-a-commodity (2026).
2. Ling, D. The $0.50 Barrel of Intelligence. https://goodai.substack.com/p/the-050-barrel-of-intelligence (2026).
3. Shafto, P., Ono, K. & Kominers, S. D. What Happens When the World Is Run on Code No One Understands?. *Time* (2026).
4. Amdahl, G. M. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. in *Proceedings of the AFIPS Spring Joint Computer Conference* 483–485 (Atlantic City, NJ, 1967).
5. Goldratt, E. M. & Cox, J. *The Goal: A Process of Ongoing Improvement*. (North River Press, Great Barrington, MA, 1984).
6. Liu, B. *et al.* LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. (2023) doi:10.48550/arXiv.2304.11477.
7. Packer, C. *et al.* MemGPT: Towards LLMs as Operating Systems. (2023) doi:10.48550/arXiv.2310.08560.
8. Besta, M. *et al.* Graph of Thoughts: Solving Elaborate Problems with Large Language Models. in *Proceedings of the AAAI Conference on Artificial Intelligence* vol. 38 17682–17690 (2024).
9. Zhang, F. *et al.* RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* 2471–2484 (Association for Computational Linguistics, Singapore, 2023).
10. Yang, J. *et al.* SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. in *Advances in Neural Information Processing Systems 37* (2024).
11. Zhang, Y., Ruan, H., Fan, Z. & Roychoudhury, A. AutoCodeRover: Autonomous Program Improvement. in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)* (Association for Computing Machinery, New York, 2024). doi:10.1145/3650212.3680384.
12. Rich, C. & Waters, R. C. The Programmer's Apprentice Project: A Research Overview. *Computer* **21,** 10–25 (1988).
13. Zhong, H. & Zhu, S. AI Harness Engineering: A Runtime Substrate for Foundation-Model Software Agents. https://arxiv.org/abs/2605.13357 (2026).
14. Young, J. Effective Harnesses for Long-Running Agents. https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents (2025).
15. Tan, G. Thin Harness, Fat Skills. https://github.com/garrytan/gbrain/blob/master/docs/ethos/THIN_HARNESS_FAT_SKILLS.md (2026).

16. Hsieh, C.-P. *et al.* RULER: What's the Real Context Size of Your Long-Context Language Models?. https://arxiv.org/abs/2404.06654 (2024).
17. Liu, N. F. *et al.* Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics (TACL)* **12,** (2024).
18. Sourcegraph. Why Coding Agents Fail in Large Codebases. https://sourcegraph.com/blog/why-coding-agents-fail-large-codebases (2026).
19. Zave, P. & Jackson, M. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology* **6,** 1–30 (1997).
20. Nuseibeh, B. & Easterbrook, S. Requirements Engineering: A Roadmap. in *Proceedings of the Conference on the Future of Software Engineering* 35–46 (ACM, New York, 2000). doi:10.1145/336512.336523.
21. Lamsweerde, A. van. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. (Wiley, Chichester, 2009).
22. International Council on Systems Engineering. *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*. (Wiley, Hoboken, NJ, 2023).
23. ISO/IEC/IEEE. *ISO/IEC/IEEE 42010:2022, Software, Systems and Enterprise — Architecture Description*. (2022).
24. Fairbanks, G. *Just Enough Software Architecture: A Risk-Driven Approach*. (Marshall & Brainerd, Boulder, CO, 2010).
25. Brooks, F. P. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer* **20,** 10–19 (1987).
26. Davis, R., Shrobe, H. & Szolovits, P. What Is a Knowledge Representation?. *AI Magazine* **14,** 17–33 (1993).
27. Harel, D. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* **8,** 231–274 (1987).
28. Baier, C. & Katoen, J.-P. *Principles of Model Checking*. (MIT Press, Cambridge, MA, 2008).
29. Gray, C. G. & Cheriton, D. R. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* 202–210 (ACM, New York, 1989). doi:10.1145/74850.74870.
30. Lampson, B. W. Protection. *ACM SIGOPS Operating Systems Review* **8,** 18–24 (1974).
31. Lazowska, E. D., Zahorjan, J., Graham, G. S. & Sevcik, K. C. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. (Prentice-Hall, Englewood Cliffs, NJ, 1984).
32. Bertsch, A., Pratapa, A., Mitamura, T., Neubig, G. & Gormley, M. R. Oolong: Evaluating Long Context Reasoning and Aggregation Capabilities. https://arxiv.org/abs/2511.02817 (2025).
33. Zhang, A. L., Kraska, T. & Khattab, O. Recursive Language Models. https://arxiv.org/abs/2512.24601 (2025).

34. Yao, S. *et al.* ReAct: Synergizing Reasoning and Acting in Language Models. in *International Conference on Learning Representations* (2023).

35. Wang, X. *et al.* Executable Code Actions Elicit Better LLM Agents. in *Proceedings of the 41st International Conference on Machine Learning* vol. 235 50208–50232 (2024).

36. Zhang, A., Li, Z. & Khattab, O. The Mismanaged Geniuses Hypothesis. (2026).

37. Zhang, A. L. & Khattab, O. Language Model Harnesses Are Compositional Generalizers. (2026).

38. Murphy, G. C., Notkin, D. & Sullivan, K. J. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. in *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering* 18–28 (ACM, New York, 1995). doi:10.1145/222124.222136.

39. Sangal, N., Jordan, E., Sinha, V. & Jackson, D. Using Dependency Models to Manage Complex Software Architecture. in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* 167–176 (ACM, New York, 2005). doi:10.1145/1094811.1094824.

40. Passos, L., Terra, R., Diniz, R., Valente, M. T. & Mendonça, N. C. Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE Software* **27,** 82–89 (2010).

41. Leveson, N. G. *Engineering a Safer World: Systems Thinking Applied to Safety*. (MIT Press, Cambridge, MA, 2011).

42. Snyk. The Agentic Development Lifecycle. https://snyk.io/blog/agentic-development-lifecycle/ (2026).

43. Cockroach Labs. CockroachDB AI Agents: CLI Database Automation. https://www.cockroachlabs.com/blog/cockroachdb-ai-agents-cli-database-automation/ (2026).

44. Saltzer, J. H., Reed, D. P. & Clark, D. D. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems* **2,** 277–288 (1984).

45. Kelly, T. P. Arguing Safety—A Systematic Approach to Managing Safety Cases. (1998).

46. Adelard. The Adelard Safety Case Development Manual. (1998).

47. Pierce, B. C. *Types and Programming Languages*. (MIT Press, Cambridge, MA, 2002).

48. Bodden, E. Self-Adaptive Static Analysis. in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results* 45–48 (ACM, New York, 2018). doi:10.1145/3183399.3183401.

49. Bodden, E. *Self-Optimizing Static Program Analysis (SOSA)*. (2023).

50. Lehman, M. M. & Ramil, J. F. Software Evolution—Background, Theory, Practice. *Information Processing Letters* **88,** 33–44 (2003).

51. Shingo, S. *Zero Quality Control: Source Inspection and the Poka-Yoke System*. (Productivity Press, 1986).

52. *Resilience Engineering: Concepts and Precepts*. (Ashgate, 2006).

53. *Site Reliability Engineering: How Google Runs Production Systems*. (O'Reilly Media, 2016).
54. Hutchins, E. *Cognition in the Wild*. (MIT Press, Cambridge, MA, 1995).
55. Zhang, J. & Norman, D. A. Representations in Distributed Cognitive Tasks. *Cognitive Science* **18,** 87–122 (1994).
56. Argyris, C. & Schön, D. A. *Organizational Learning: A Theory of Action Perspective*. (Addison-Wesley, 1978).
57. Reason, J. *Human Error*. (Cambridge University Press, Cambridge, 1990). doi:10.1017/CBO9781139062367.
58. Lehman, M. M. & Belady, L. A. *Program Evolution: Processes of Software Change*. (Academic Press, London, 1985).
59. Red Hat. Why Developer Portals Matter More in the Age of AI Agents. https://www.redhat.com/en/blog/why-developer-portals-matter-more-age-ai-agents (2026).
60. Pólya, G. *How to Solve It: A New Aspect of Mathematical Method*. (Princeton University Press, Princeton, NJ, 1957).
61. Lakoff, G. & Johnson, M. *Metaphors We Live by*. (University of Chicago Press, Chicago, 1980).
62. Reimann, T. How Cloudflare Enforces Engineering Standards Using AI. https://blog.cloudflare.com/engineering-standards-enforcement/ (2026).
63. Charas, M. & Bruggmann, M. Honk: Autonomous Code Migration at Spotify. https://engineering.atspotify.com/ (2026).
64. Barr, E. T., Harman, M., McMinn, P., Shahbaz, M. & Yoo, S. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* **41,** 507–525 (2015).
65. Zamudio Amaya, J. A., Smytzek, M. & Zeller, A. FANDANGO: Evolving Language-Based Testing. in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2025)* (ACM, New York, NY, 2025). doi:10.1145/3728915.
66. Sillito, J. & Kutomi, E. Failures and Fixes: A Study of Software System Incident Response. in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)* 185–195 (2020). doi:10.1109/ICSME46990.2020.00027.
67. PagerDuty. Supercharging Incident Response with Runbook Automation. https://www.pagerduty.com/blog/supercharging-incident-response-runbook-automation/ (2021).
68. LinearB. The Engineering Productivity Gap: How Elite AI Teams Are Pulling Away from the Rest. https://linearb.io/resources/ai-engineering-productivity-gap (2026).
69. Ousterhout, J. Always Measure One Level Deeper. *Communications of the ACM* **61,** 74–83 (2018).
70. Atlassian. AI Agents for Jira Engineering Maintenance. https://www.atlassian.com/blog/development/ai-agents-jira-engineering-maintenance (2026).

71. Twilio. Why Connect Twilio AI Skills. https://www.twilio.com/en-us/blog/developers/best-practices/why-connect-twilio-ai-skills (2026).

72. NVIDIA. NVIDIA Verified Agent Skills Provide Capability Governance for AI Agents. https://developer.nvidia.com/blog/nvidia-verified-agent-skills-provide-capability-governance-for-ai-agents/ (2026).

73. Carlson, L. Higher Ed Accessibility Lawsuits, Complaints, and Settlements. https://www.d.umn.edu/~lcarlson/atteam/lawsuits.html (2026).

74. Ringer, T., Palmskog, K., Sergey, I., Gligoric, M. & Tatlock, Z. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends in Programming Languages* **5,** 102–281 (2019).

75. Gamma, E., Helm, R., Johnson, R. & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. (Addison-Wesley, Reading, MA, 1994).

76. Staples, B. When Code Is Abundant. https://about.gitlab.com/blog/when-code-is-abundant/ (2026).

77. Medisetty, U. K. Running a Software Factory Efficiently at Uber Scale. https://www.uber.com/us/en/blog/efficient-software-factory/ (2026).

78. Conant, R. C. & Ashby, W. R. Every Good Regulator of a System Must Be a Model of That System. *International Journal of Systems Science* **1,** 89–97 (1970).

79. Bollinger, T. B. & McGowan, C. L. A Critical Look at Software Capability Evaluations. *IEEE Software* **8,** 25–41 (1991).

80. Riggins, J. AI Coding Got Faster. Why Didn't Engineering?. https://thenewstack.io/ai-productivity-measurement-gap/ (2026).

81. Storey, M.-A. From Technical Debt to Cognitive and Intent Debt: Rethinking Software Health in the Age of AI. *Queue* **24,** (2026).

82. Winters, T., Manshreck, T. & Wright, H. *Software Engineering at Google: Lessons Learned from Programming over Time*. (O'Reilly Media, Sebastopol, CA, 2020).

83. OpenAI. Why SWE-bench Verified No Longer Measures Frontier Coding Capabilities. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/ (2026).

84. Smith, E. K., Barr, E. T., Le Goues, C. & Brun, Y. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)* 532–543 (2015). doi:10.1145/2786805.2786825.

85. U.S. Department of the Interior, National Park Service. The Secretary of the Interior's Standards for Rehabilitation. (2024).

86. U.S. Department of Justice. Nondiscrimination on the Basis of Disability; Accessibility of Web Information and Services of State and Local Government Entities. (2024).

87. Conway, M. E. How Do Committees Invent?. *Datamation* **14,** 28–31 (1968).

88. Hakim, S. B., Adil, M., Velasquez, A. & Song, H. H. Neuro-symbolic Agentic AI: Architectures, Integration Patterns, Applications, Open Challenges and Future Research Directions. *Computer Science Review* **60,** 100902 (2026).

89. Allamanis, M., Brockschmidt, M. & Khademi, M. Learning to Represent Programs with Graphs. in *International Conference on Learning Representations* (2018).

90. Luo, L. *et al.* G-Reasoner: Foundation Models for Unified Reasoning over Graph-Structured Knowledge. in *International Conference on Learning Representations* (2026).

91. Kumar, S. *et al.* Time Warp: The Gap Between Developers' Ideal vs Actual Workweeks in an AI-Driven Era. in *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* 12–22 (2025). doi:10.1109/ICSE-SEIP66354.2025.00007.

92. Meyer, A. N., Barr, E. T., Bird, C. & Zimmermann, T. Today Was a Good Day: The Daily Life of Software Developers. *IEEE Transactions on Software Engineering* **47,** 863–880 (2021).

93. Butler, J. *et al.* Eight Myths on Software Engineering and GenAI. *Queue* **24,** (2026).

94. Ralph, P. The Sensemaking–Coevolution–Implementation Theory of Software Design. *Science of Computer Programming* **101,** 21–41 (2015).

95. Ralph, P. Software Engineering Process Theory: A Multi-Method Comparison of Sensemaking–Coevolution–Implementation Theory and Function–Behavior–Structure Theory. *Information and Software Technology* **70,** 232–250 (2016).

96. Reeves, J. W. What Is Software Design?. *C++ Journal* https://www.developerdotstar.com/mag/articles/reeves_design.html (1992).

97. Ruan, H., Zhang, Y. & Roychoudhury, A. SpecRover: Code Intent Extraction via LLMs. in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering* (2025). doi:10.1109/ICSE55347.2025.00080.

98. Tang, H., Key, D. & Ellis, K. WorldCoder, a Model-Based LLM Agent: Building World Models by Writing Code and Interacting with the Environment. in *Advances in Neural Information Processing Systems 37* (2024).

99. Appel, A. W. *Modern Compiler Implementation in ML*. (Cambridge University Press, Cambridge, 1998).

100. Friedenthal, S., Moore, A. & Steiner, R. *A Practical Guide to Sysml: The Systems Modeling Language*. (Morgan Kaufmann, 2014).

101. Brambilla, M., Cabot, J. & Wimmer, M. *Model-Driven Software Engineering in Practice*. (Morgan & Claypool, 2017).

102. Bosu, A., Greiler, M. & Bird, C. Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. in *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)* 146–156 (IEEE, 2015). doi:10.1109/MSR.2015.21.

103. Greiler, M. From Peer Review to Self-Review: The Epistemic Risk of Agentic Development. https://www.michaelagreiler.com/code-reviews-from-team-to-individual/ (2026).

104. National Society of Professional Engineers. Responsible Charge. (2024).
105. National Society of Professional Engineers. Licensure Exemptions. (2023).
106. Bugliarello, G. The Social Function of Engineering: A Current Assessment. in *Engineering as a Social Enterprise* (National Academy Press, Washington, DC, 1991).
107. Future of Life Institute. Pause Giant AI Experiments: An Open Letter. (2023).
108. Herbert, F. *Dune*. (Chilton Book Company, Philadelphia, 1965).
109. National Academies of Sciences, Engineering, and Medicine. *Building Resilience into the Nation's Medical Product Supply Chains*. https://doi.org/10.17226/26420 (2022) doi:10.17226/26420.
110. Presidential Commission on the Space Shuttle Challenger Accident. *Report of the Presidential Commission on the Space Shuttle Challenger Accident, Volume 1*. (1986).
111. Columbia Accident Investigation Board. *Columbia Accident Investigation Board Report, Volume 1*. (2003).
112. U.S. Department of Justice, Office of Public Affairs. Volkswagen Engineer Pleads Guilty (2016) and Is Sentenced to 40 Months in Prison (2017) for His Role in the Conspiracy to Cheat U.S. Emissions Tests. (2017).
113. U.S. Senate Committee on Commerce, Science, and Transportation. Protecting Kids Online: Facebook, Instagram, and Mental Health Harm. (2021).
114. U.S. Senate Committee on the Judiciary. Social Media and the Teen Mental Health Crisis. (2023).
115. Vella, A. & Blincoe, K. The Impact of AI Coding Assistants on Software Engineering: A Longitudinal Study. https://arxiv.org/abs/2605.23135 (2026).
116. ABET Engineering Accreditation Commission. Criteria for Accrediting Engineering Programs, 2025–2026. https://www.abet.org/accreditation/accreditation-criteria/criteria-for-accrediting-engineering-programs-2025-2026/ (2025).
117. ABET Computing Accreditation Commission. Criteria for Accrediting Computing Programs, 2025–2026. https://www.abet.org/accreditation/accreditation-criteria/criteria-for-accrediting-computing-programs-2025-2026/ (2025).
118. Fagan, M. E. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal* **15,** 182–211 (1976).
119. Fagan, M. E. Advances in Software Inspections. *IEEE Transactions on Software Engineering* **SE-12,** 744–751 (1986).
120. Moreno, J. & Libbey, B. Under the River. https://shopify.engineering/under-the-river (2026).
121. Peña, M. de la. A Virtual Agent Team at Docker. https://www.docker.com/blog/ (2026).
122. Siemens Digital Industries Software. A3E: Autonomous, Agentic Assistance for Engineering. https://blogs.sw.siemens.com/ (2026).

123. Dufwa, P. & Luvö, T. A Platform for Scalable Enterprise AI Agents. https://zenseact.com/news/a-platform-for-scalable-enterprise-ai-agents/ (2026).

124. Cui, Z. (K. *et al.* The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers. *Management Science* https://doi.org/10.1287/mnsc.2025.00535 (2025) doi:10.1287/mnsc.2025.00535.

125. Becker, J., Rush, N., Barnes, E. & Rein, D. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. https://arxiv.org/abs/2507.09089 (2025).